ABSTRACT

TrollCreek is a CBR- tool developed at our group recognised for it's ability to reason in weak theory domain, explanatory facilities, and sustained learning. Artificial Neural Networks provide a method for analyzing complex data without any a-priori knowledge of their possible interactions.

This work combines classical methods and ANN with CBR by extending the reasoning capabilities of our group's TrollCreek to handle real- time processing of massive, inter-correlated datasets. These is done by pre- selecting only relevant attributes from the Measurement While Drilling data using a regression tool and analyze these using ANN. The result is reducing 33 real- valued variables to one, problem relevant signal, readily processed by TrollCreek.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Professor Agnar Aamodt first and foremost for believing in my ideas, sharing my enthusiasm for this work and for his catching, genuine interest for the AI discipline. This has periodically been an independent study but I have flourished professionally as a result of his confidence and reassuring feedback. As a teaching supervisor he possesses impressively extensive professional knowledge within his domain, but also provides invaluable knowledge about the scientific work process.

I would also like to thank Professor Harald Valaas for his guidance in selecting and applying an expedient statistical method for this work.

Furthermore, I would like to thank Professor Kesheng Wang at the Institute of Product and Quality Engineering, NTNU for his practical approach in providing ideas our discussions selecting candidate AI methods for this work.

Finally I sincerely thank my parents for invaluable practical, moral and financial support during the last year of this work.

LIST OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Abbreviations

CHAPTER 1

INTRODUCTION

The vision of this work is to utilize computational intelligence in preventing and mitigating unwanted events in the oil-drilling domain, not limited to avoid mistakes happening or merely operationalizing existing knowledge. The reasoning system should exhibit a minimum of explanatory functionality, not merely stating that a situation is interpreted as dangerous, but also being able to explain why. Finally, the reasoning capabilities should be extended to sustainable learning from mistakes to avoid repeating experienced unwanted events.

Obtaining these paramount goals involves achieving several high- level objectives which include methods that will enable::

1. continuous background monitoring and giving qualified warnings accordingly
2. explaining why the system interprets the ongoing operation as dangerous
3. making justified recommendations primarily how to avoid the confirmed unwanted event in progress and secondarily how to mitigate an expected unwanted event
4. continuous utilization, maintenance and update of a lessons learnt knowledge base

Chapters 1-2 show that existing research and work on virtual intelligent prediction and learning in this domain fails to adequately address the objectives listed above, and it seem necessary to combine classical transparent methods and machine learning techniques to achieve the project vision outlined.
After an introduction to the selected problem domain, the project high- level objectives are formalized into the goals of this work.

Problem domain studied

There are numerous possible unwanted events being candidates for a prediction and mitigation feasibility analysis. The focus of this work ended up on prediction and mitigation of stuck pipe, despite it being a somewhat declining problem with respect to frequency. Daily drilling rig rental has increased proportionally to the oil price and presently, off- shore rig day rates might exceed 160000 US$/day [Office Of Statistical and Economic Analysis (2005)], and a severe stuck pipe incident might take several days to resolve. Equally important is the availability, frequency and quality of the relevant data subject for analysis and the absence of a commonly accepted computationally intelligent system minimizing these problems. Furthermore, classification of a possible training set is feasible since there exist an industry accepted database on how to map stuck- pipe decisions [Bailey L., Jones T., Belaskie J., Ortban J., Sheppard M., Houwen O., Jardine S. and McCann D. (1991)]. This does not mean, however, that this work is limited to the problem of stuck pipe, other unwanted drilling events are possibly susceptible for such analysis. using the existing dataset. Moreover, data collection enhances both quantitatively and qualitatively as a consequence of increasingly complex drilling operations and regulatory demands, continuously expanding the plausible scope of this work.

1.1 Research goals and hypotheses

The following sections is the guiding documentation for further analysis:

1.1.1 Goal 1: Real- time, computationally efficient prediction

Predicting unwanted events on complex offshore development well drilling operations by finding a reliable problem indicator or an interpretable unwanted event signal within readily available digital data.

Hypothesis 1: It is possible to develop a method predicting unwanted events real time from MWD data

- **Sub- Hypothesis 1.1**: There is sufficient explanatory strength within the available rig site data to get a consistent, reliable problem signal for real- time prediction of unwanted events.
- **Sub- hypothests 1.2**: Given the right predictive system, the explanatory strength within these data might provide an adequate problem signal at relative low computational cost.

1.1.2 Goal 2: Provide justified explanations, recommendations and learning capabilities

This is the final process of providing explanations and justifications for alarm states, justified recommendations for preventive measures and built- in capabilities of retaining lessons learnt.
As Chapters 2-3 will show, Goal 2 has long been a subject of research within our AI- group, but in this context goal 2 also depends on the validity of Sub-H1.1 and Sub-H1.2.

Depending on the success of goal 1 alone, this research might differ *qualitatively* from existing predictive systems. Goal 2, however, separates this work *methodically* from any existing real- time monitoring system by elevating the target of problem handling from merely indicating an alarm state to providing explanations for the alarm, suggesting pre-emptive measures and learning from the alarm incident.

1.2 High- level, preliminary design of the proposed solution

In the following sub- chapter, an analysis of the goals leads to a formulation of functional requirements, explaining the functionality required by the system in obtaining the goals of this work and an introduction to the proposed solution.

1.2.1 Functional requirements

The system proposed should document ability to

1. continuous background monitor an ongoing operation at low computational cost, providing appropriate warning signals in a timeframe allowing avoidance of the reported unwanted event.
2. explaining why the system interprets the ongoing operation as dangerous
3. making justified recommendations how to primarily avoid the confirmed unwanted event in progress and secondarily how to mitigate an expected unwanted event
4. continuous and proactive utilization, maintenance and update of a lessons learnt knowledge base

The cumulative requirements include both data intensive predictive power and extensive knowledge representation and reasoning as explanations and justifications require both causal and structural knowledge.
This exclude both purely instance- based, statistical and ANN (Artificial Neural Nets) approaches. A version of CBR (Case Based Reasoning) learners, TrollCreek addresses the knowledge intensity embedded in the complex and inadequately modeled problem domain studied. Meanwhile, the problem to be addressed exhibits both data intensive and knowledge intensive characteristics. A version of Case Based Reasoning, TrollCreek, might address goals 1 & 2. However, Chapter 3 shows that representational, reasoning and computational problems when processing numeric, real- time data makes TrollCreek an implausible solution in meeting goal 1.
A consequence of the above is that realizing both goals necessitate modulating the system as no known, single machine learning technique adequately accommodates all functional requirements

1.2.2 Preliminary design

Figure 2-1 shows a high level design of the proposed extension to the existing decision support and learning system. All process flowcharts in this thesis, data or data storages are represented as circles, processing elements as boxes. The processing element in grey (green) indicates that parts of the functionality required are present in an existing system.

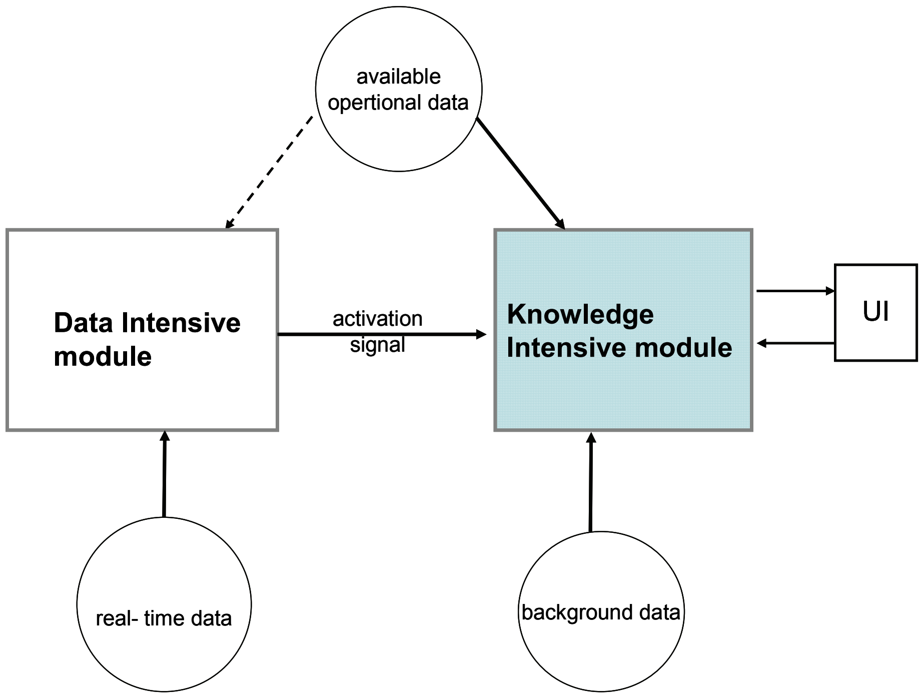


Figure 1.1: Abstract view of the proposed system architecture accomplishing the thesis goals

Data intensive processing

The process element "data intensive processing" performs the real time, data intensive prediction, presupposing this is performed by a trained module, requiring only the less processing expensive querying step. The main source of data is any operational real time data, typically updated every 5 seconds, but might make use of supporting data from more static data sources if needed.

Knowledge intensive processing

When an alarm signal is generated, the control is passed to the "knowledge intensive processing" module which accepts or dismisses this trigger event, based on the embedded domain knowledge and experience. Depending on the outcome of this module's alarm signal evaluation, user interaction or an explained unwanted alarm is communicated to the user which acts as the evaluator of the system's overall performance. Regardless of signal type, alarm or solution correctness, prospective new experience is justifiably stored or dismissed as important learning steps in the process chain.

The details of the individual processing elements, data descriptions, control and message exchange is the essence of this work and will be the subject of the succeeding chapters.
However, the core of this thesis work will focus on the ability to make predictions from real-time numerical data and the way such a module integrates in the total architecture.

1.3 Structure of this work

A broad area of methods is discussed in this thesis, thus leaving less latitude for in- depth elucidation of each. However, the target is primarily to provide sufficient theoretical background to appreciate the purpose, reasoning and outcome of each chapter, secondarily to provide pointers to more elaborated material.

The rest of this chapter summaries both "thesis philosophy" and "thesis anatomy". The first is how to achieve the goals and requirements stated, the latter how this document is organized.

1.3.1 How to achieve the goals

Goal 2 essentially describes our group's Case Based Reasoning (CBR) system TrollCreek. This contribution condenses into analyzing and patching TrollCreek's shortcomings in reaching the thesis goals, and to develop amendments as needed.
A significant obstacle in reaching Goal 2 using the existing system is handling real- time operational data. A major amendment produced by this work is data intensive prediction as described in Goal 1, thus relieving TrollCreek of operating on these data.
In searching for a method at the data intensive stage, the nature of this thesis is explorative. The boundaries given by the functional requirements narrows the scope of possible methods down to eager learners[1], but this still leaves fuzzy logic and the entire ANN (Artificial Neural Network) family of plausible methods. The data description further narrows down the scope of feasible methods of analysis, but for the unknown data intensive module, the "thesis philosophy" becomes an empiric one.

[1] Among their distinctive characteristics, eager learner executes the computationally expensive steps prior to deployment. Classification, or prediction in this context, thus becomes inexpensive[Mitchell, T. M. 1997] pp. 230-244

Introducing FuzzyLogic would improve the existing reasoning on MWD parameters within the existing version of TrollCreek. However, the nature of the data as shown in this chapter and discussed in Chapter 3: Discretization -and using rules on these real- valued parameters makes reliable real- time prediction infeasible, which leaves only the ANN family of methods for testing.

Empiric search supported by statistical regression

Testing Artificial Neural Networks in general, and especially Standard Backpropagation -ANN (SBP-ANN), is a highly empiric exercise having several tuning variables, no modeling and few tuning heuristics. Moreover, there is a considerable data load, with expect to frequency, parameter quantity -and range. This leaves a considerable search space for an optimal solution. Regression was used to aid reducing the complexity of the neural nets by selecting only variables relevant to the problem studied.
As interpretable patterns started to form, additional tools were added successively as for instance noise- reduced normalization and function smoothening.
The work procedure might be summarized to *successively explore each candidate method, test the remaining plausible ones and iteratively develop the SBP-ANN.*

1.3.2 Thesis structure

Software system development is not top- down and is a major element in this work. The nested dependencies between analysis of a broad area of methods, iterative development and testing constitute a challenging documentary exercise. This is necessarily reflected in the structure of the presented thesis.

This chapter has outlined the desired goal state and a high- level design of the system proposed to achieve this state.

Chapter 2 provides a thorough data analysis, enabling formalization of functional requirements of the data intensive module and a detailed system design.

Given the modularity of the proposed system, the three succeeding chapters are somewhat independent studies of the individual modules, each having separate results, discussion and conclusion parts. This is advantageous and expedient as each module is assigned separate tasks, procure separate data and produce separate outputs. Meanwhile, these modules are interconnected by their supereminent objectives and message exchanges. These dependencies are attended to successively and in the common discussion and conclusion chapter.

In Chapter 3 the first of these modules are analyzed and is the system already present: TrollCreek. The conclusions of this analysis either confirm or lead to modification of the functional requirements given the data intensive module, developed through the next two chapters:

A statistical tool is presented and utilized in Chapter 4, significantly reducing the data extent by including only relevant parameters, thus increasing feasibility of the real- time prediction.
Chapter 5 constitutes the majority of this work and is composed of an analysis of Artificial Neural Nets (ANN); identifying a suitable neural net, leading to the development of an extended Backpropagation ANN. An empiric analysis of the significant drilling parameters identified in Chapter 4 concludes this chapter and also concludes the modular analyses.

In the final chapter, the conclusions of the modular analysis in Chapters 3-5, are cumulated, discussed and measured towards the thesis goals. Moreover, the important sub- chapter "Future Work" outlines development steps necessary to either integrate into, or create an interface from the data intensive module and TrollCreek.

Practical aspects

Abbreviations and petroleum specific terminology is either explicitly explained at the first occurrence of this term in a chapter, or a pointer to an explanation is provided. In addition, all abbreviations used, independently of domain, are included in the list of abbreviations with a pointer to an explanation.

CHAPTER 2

METHODS AND DESIGN

The former chapter described the high- level goals, functional requirements and constraints.
This chapter describes the status of current research with regards to this work. A detailed data description –and analysis lead to functional requirements and design criterion for the data intensive, predicting module. This narrows the scope of plausible prediction tools down to Artificial Neural Networks (ANN) and brings forth a more fine- grained, overall design.
Consequently, this chapter becomes the final, purely theoretical design iteration prior to in- depth analysis, testing and development of the AI- methods constituting the proposed system.

2.1 Related research

Much effort has been put into utilizing Artificial Intelligence as a tool to prevent unwanted events in the petroleum industry by attempting to model this complex domain by statistical and analytical methods, by enhancing the focus and knowledge of rig personnel (pedagogical, motivational approaches) For some very limited, simple and repetitive well operations, basic machine learning approaches has been promisingly utilized.
The next sub- chapters mention some published tools related to this work.

2.1.1 ANN utilization

Neural networks have been utilized to predict formation characteristics such as porosity and permeability [Mohaghegh, S., Arefi, R., Bilgesu, I., Ameri, S. 1995], field development [Doraisamy, H., Ertekin, T., Grader, A., 1998] and completion analysis [Shelley, R., Stephenson, S., Haley, W., Craig, E., 1998] from data where transparent methods performed poorly.
Recent examples are the use of ANN's in data mining of huge databases to optimizing entire oilfield production [Salehi, I (2004)] or predicting ultimate drilling and completion practices for a well construction project [L., V. (2004)]

2.1.2 CBR methods

Schlumberger Cambridge Research built a Web-based system to diagnose the causes of stuck pipe during drilling and then to suggest remedies [Arango, G., Colley, N., Connely, C., Durbec C., 1997]. This works by initiating a case- base, interviewing the user and implementing lessons learnt. However this is a knowledge poor data- mining tool not monitoring the operation autonomously thus depending on the user detecting the problem and guiding the system through the knowledge database.

2.1.3 Commercially developed tools

There are commercial actors that specialize in the field of AI applied to oil and gas production, one of which is Intelligent Solutions Inc [Intelligent Solutions, 2005]. This company claim to be the most experienced actor in aapplication of artificial intelligence to all aspects of petroleum and natural gas engineering.

Schlumberger is the largest service provider to the oil and gas industry and possesses a vast inventory of drilling related software, one of which is designated to predicting stuck pipe; Osprey Risk [Osprey Risk, 2004], another example is their Petrel Classification and Estimation tool [Petrel Classification, 2005] utilizing ANN's to predict reservoir characteristics

Their software includes services as reservoir modeling, simulation and intelligent best practices analysis. However, these analyses is limited to pre- operational prediction with regards to design and post- operational analysis and learning.

2.1.4 Summary

There is extensive commercial and academicals research and deployed commercial software utilizing machine learning techniques aiming at predicting and preventing non- productive time during complex off- shore well construction. There are systems methodologically similar, both on CBR and ANN. Presently however, there seem to be no released or published material from research groups adequately covering neither goal 1 nor 2 individually and consequently no documented system covering the overall system functionality.

2.2 Data description

In many aspects the real- time data, their nature and intensiveness dictate the functional requirements for the tools in which they are to be processed.

There are three main categories of data available during a typical offshore drilling operation:

Category 1 data "Real Time Data": In this work these data has been exclusively MWD (Measurements While Drilling) data, but this category constitutes any readily available, automatically and frequently recorded, non- symbolic data that consequently need no interpretation.

Category 2 data "Routinely Recorded Data" are digitally recorded at least daily as an integral part of the drilling operation. Their source is mainly the Daily Drilling Report[2] which is a requirement of the NPD[3] including section size, drilling fluid data (type, specific weight, viscosity etc., daily mud losses, and equipment.

Category 3 data "Background Data": These data are manually recorded, being symbolic, numeric or textual descriptions requiring interpretation. These data need not necessarily be available digitally or at the operations control site. Data acquisition is in the boundary line of knowledge acquisition (see for instance [Luger, G. F (2002) pp. 250-256, Gruber, T.R (1996)]). This must not, however, be confused with knowledge acquisition in the sense of domain modeling and achieving sustained, robust and adaptive knowledge as described in e.g. [Aamodt, A (1995)]. Background data is field specific information giving a large scale picture of operations parameters such as reservoir data, known HTHP (High Temperature High Pressure) zones, water/oil/gas ratio, formation specific data (e.g. lithology through possible core samples) log data (samples/previous exploration wells) and wireline data.

A fourth category including e.g. general domain knowledge and a case base could be introduced here, but once elevated to the knowledge intensive level beyond the mere predictive part of the system, one of the learner's tasks is to retrieve relevant *cases* (Chapter 3) and compare an evolving case- base for matching. Depending on context these cases might play the role of being mere data, information or even knowledge [Aamodt, A, Nygård, M, 1995], consequently there is no data category at this level.

2.2.1 Category 1 data - "Real Time Data"

If not continuously covered, all referred and significant abbreviations used on drilling parameters in this chapter are described in Appendix 9.1: "Statistically significant MWD variables".

Category 1 data are mainly collected for generating alarm events invoking the knowledge-intensive part of the system, but those alarm signals might also be carriers of selected and prepared data vital for the further reasoning process.

[2] Detailed drilling story", a textual summarization of main activities, current operations, events and deviations with an update frequency at best every 30'th minute.

[3] Norwegian Petroleum Directorate

The data used for alarm signal generation in this work are exclusively MWD (measurement while drilling) data collected real time and sampled every 5 seconds. Below is an example of some raw data recorded:

Table 2.1: Extract from MWD data. 9 of 33 recorded parameters are shown and 8 recordings constitute 40 seconds of operational time.

Depth	UnixTime	BitMDepth	BlockPos	AvgROP	AvgHookld	CalcHookl	AvgWOB	AvgTorq	...
4703	766766755	4281.651	2.656	30.354	316363.625	407314.08	-680290.79	25971.79	...
4703	766766760	4309.813	2.647	4.796	530843.688	407314.08	-123529.6	1922.005	...
4703	766766760	4309.821	2.68	0.828	1125535.88	407314.08	-128293.9	1918.35	...
4703	766766760	4309.738	2.78	0.828	1131686.88	407314.08	-128293.9	1918.35	...
4703	766766760	4309.705	2.797	0.828	1093755.88	407314.08	-128293.9	1918.35	...
4703	766766760	4309.821	2.797	0.828	1075302.88	407314.08	-128293.9	1918.35	...
4703	766766760	4309.821	2.797	0.828	1041472.5	407314.08	-128293.9	1918.35	...
4703	766766760	4282	2.805	0.828	935880.688	407314.08	-128293.9	1918.35	...
...	...	...	...	...	...	...	...	...	...

The purpose of Table 2.1 is to give an overview of the data, not providing an insight of relevant parameters at this stage. Each row in this table represents a data instance (sample) having 33 parameters, of which 9 are shown in the table. These real- valued parameters have varying reliability and relevancy to the selected unwanted event; stuck pipe, and a summarization of the collected data material characteristics are:

Massive input data

Real- time data mean data need processing at the speed of which they enter the system. Having a somewhat representative data distribution of a normal vs. abnormal drilling situation is a trade-off between, on one hand; expected precision gain following increased data intensiveness vs. training cost.

Poor target estimates

The dependant variable studied here; stuck pipe (See Appendix 9.3: "Mechanisms causing the unwanted event Stuck Pipe") is dicotome, that is, the value is either

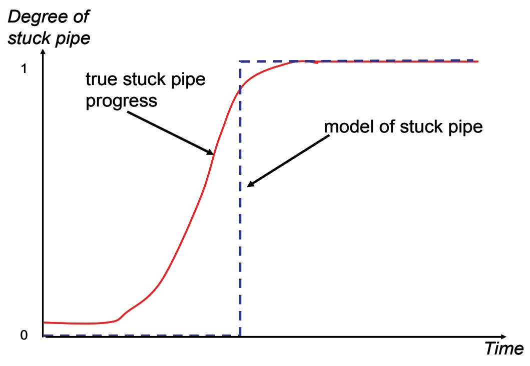


Figure 2.1: model vs. true stuck pipe progress

stuck or not stuck, 0 or 1. This is a rough model of the actual stuck pipe event, which typically is a result of cumulative effects from varying problems. A real valued continuous function is, in lack of a transparent, analytical model, simplified to a step- function in the data representation.

Extreme data:

There are few "true", *independent* training examples: MWD data for development wells including a single, clearly identifiable stuck pipe incident are not readily available as these surveys almost exclusively are outsourced to service companies and these data are subject to company restrictions. In this work, 7 wells has been the basis for training and validation. Each of these training examples (in the strict interpretation of the term) is, however, represented by up to 17000 instances. The challenge becomes handling few independent, but severely data intensive, training examples.

Skewed distributed target value

The dependant variable (target value) of each instance; the values subject to prediction, represent a severely skewed distribution. Each independent training example has 2500-6000 negative instances per 100 positive instances and all negative instances are sequentially listed prior to the 100 positive samples.

Dispersed value ranges and inter- correlated parameters

All parameters values are given in SI units (see Appendix 9.1: "Statistically significant MWD variables") having observed value spans differing greatly and regardless of their relative predictive significance.

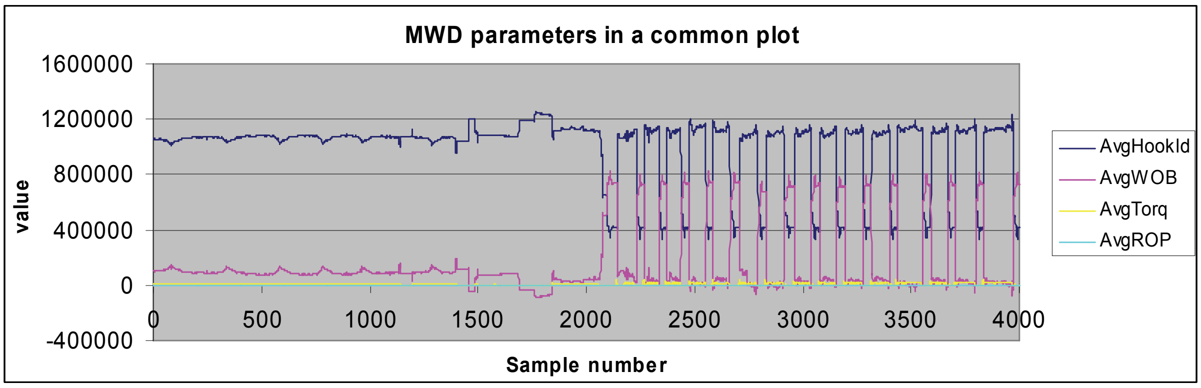


Figure 2.2: Example of value dispersion in MWD data

Examples from figure 2-3 are ranges of MWD parameters $V_{AvgROP} = [0,1.7\rangle$ and $V_{AvgWOB} = [-150000,150000\rangle$. AvgROP is intuitively at least as significantly correlated to stuck pipe as AvgWOB. Their measuring units and their amplitudes do not necessarily reflect their relative correlation strengths towards stuck pipe. Figure 2-3 also show examples of two severely inter-correlated variables, AvgWOB and AvgHookld: The former essentially expresses the inverse of the latter.

Noisy data

Several of the measurements are based on less than reliable instruments and are error prone. Figure 2 shows $V_{Hookload} = [-600, 200000]$ which is one of example of observed severe noise. The relative weight of the drill string in drilling fluid cannot be less than zero unless there is an explicitly noted, severe gas influx into the wellbore. Another example is ROP values up to 1500 while a typical $ROP_{max} < 5$.

In the following sub- chapter, these data descriptions are summarized and set into the goal-achieving context.

2.2.2 Data descriptions summarized into functional requirements

The data intensive, knowledge poor part need to handle:

1. massive amounts of data, especially when trained as real- time data need reliable prediction on the fly
2. a poorly classified training database
3. few truly independent training cases – 7 wells and each training case having massive amounts of training examples
4. the training examples' classifications are severely skewed distributed
5. training examples' features are somewhat inter- correlated

6. noisy data having dispersed value spans

In the next section a system is suggested accommodating both goals and functional requirements.

2.3 System design

This is the final design stage prior to testing and development, narrowing the scope of possible prediction methods as a result of the cumulative functional requirements.

2.3.1 Data intensive, real- time prediction

Earlier discussion has excluded purely instance- based[4] and lazy learners for this stage. Remaining, considered eager learners are FuzzyLogic [Luger, G. F. 2002] pp. 323-328 and ANNs.

Preliminary analysis [Valaas, I, 2004] showed that a pure "Black Box Training" strategy, utilizing all available MWD parameters gave no interpretable results, regardless of ANN method, topologies and their pertinent tuning parameters (Section 5.4.2 "Training Algorithms"). A method of pre- selecting only relevant features proved necessary.
Two strategies are employed:

1. In an earlier work [Valaas, I., 1997] analytical real- time processing of the MWD data was tested in a promising attempt to create a function modeling a targeted stuck pipe mechanism called "hole cleaning problems" (see Appendix 9.3: "**Error! Reference source not found.**" section 9.3.2).
   In developing the "Prediction"- module, a second attempt is made to create this model using the Radial Basis Function Networks' inherent function interpolation ability (Section 5.4) and utilizing the analytical and domain knowledge embedded in the original data preparation (see Appendix 9.2: "Dataset derived from applicable analytical models applied to relevant, available MWD data").
2. Selected training data including all MWD parameters (features) are presented to the Multivariate Logistic Regression (MLR) module which analyses and ranges these features according to problem relevance. Parameters found relevant according to a performance criterion (5% level of significance) constitute the input variables at the next level of prediction i.e. the Standard Back- Propagation ANN analysis(Chapter 4: "Attribute Selection By Statistical Analysis").

[4] Instance based learners are "lazy learners", performing all computationally expensive stages until a sample arrives and need classification (prediction in this context) [Mitchell, T. M. 1997] pp. 230-244

Details of the internal processing of these data are covered in Chapter 5: "Artificial Neural Networks".

2.3.2 Overall design – second iteration

Figure 2.3 shows the proposed system architecture attempting to accommodate both the goals and the functional requirements described above.

Boxes represents processing modules of the system each having well- defined responsibilities that cumulatively addresses Goal 1 and.2. Circles represent databases both static and dynamic and finally; dotted ovals show message exchange between the modules.

The static domain knowledge acquired and the dynamic case base embedded in the KI CBR (Knowledge Intensive Case Based Reasoning) module is not shown in the figure. This knowledge is, as discussed previously, elevated beyond the definition of data and is considered an integral part of the Deep Reasoning module. Below is a brief summary of each processing module:

- The processing sub- module "Analytical Methods" is a data pre- processing method supporting the next processing element. As discussed in the previous sub-chapter, two possible analytical methods are studied to qualitatively improve and reduce the training data: Domain knowledge supported transparent methods[5] and regression analysis.
- The "ANN" sub- module is trained on the resulting dataset and performs real- time prediction.
- The TrollCreek module has already been sketchy described and will be elaborated in Chapter 3: "Case Based Reasoning and TrollCreek"

[5] Transparent method: Exact mappings between input vectors and their classifications. "functions" and "equations".

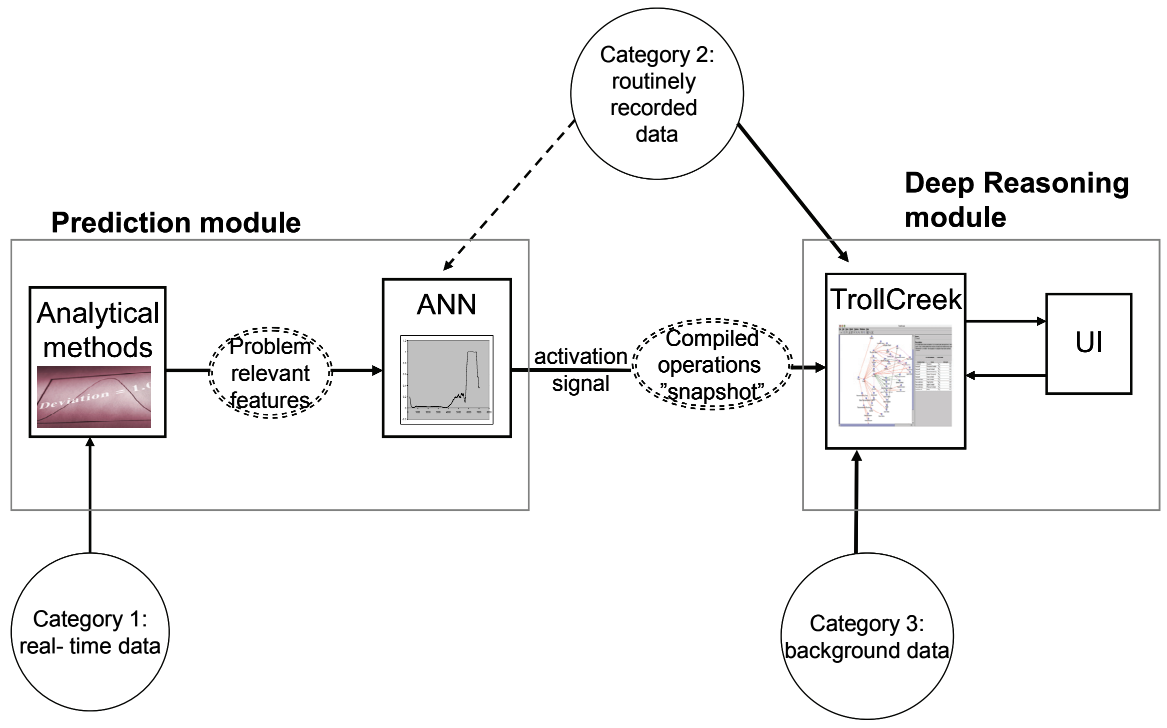

Category 2: routinely recorded data
Deep Reasoning module
Prediction module
Analytical methods
Problem relevant features
ANN
activation signal
Compiled operations "snapshot"
TrollCreek
UI
Category 1: real- time data
Category 3: background data

Figure 2.3: System design – second iteration

The next section gives an overview the functionalities, messages exchange and control flow of the proposed system.

2.3.3 System functionality and interaction

Prediction

Control input to this module is either start of operations or a dismissed, earlier warning signal from TrollCreek. If it is the latter, an alarm threshold needs to be incrementally increased. Output of this module is an alarm signal generated whenever a problem indicator exceed the alarm threshold This signal, comprised of the real valued amplitude exceeding the threshold, and possibly along with selected and relevant operational data, is passed to the next module in the problem solving chain.

After the prediction stage, control is passed to TrollCreek.
Details of the internal processing of steps Re is covered in Chapter 3: "Case Based Reasoning and TrollCreek". Below is an informal summary :

Explain

At this stage the alarm signal is interpreted and translated and a search for relevant, similar cases is triggered using the reasoner's semantic network. If one or several comparable unwanted event-cases are retrieved by direct matching, this might be explanation sufficient. If a case is somewhat similar, this partial match need be explained.. The output of this process is one or several justified problem hypotheses presented to the user through the user interface, which helps the system through the next phase:

Either the system dismisses the warning internally ot the user interacts with the system in either acknowledging or rejecting the proposed hypothesis after having completed possible missing or inadequate information in the knowledge base.

Recommend

Further steps in the KI reasoning process depends on the outcome of the former phase. If the user ultimately dismisses the proposed unwanted event hypothesis, this step is skipped and focus is directed at the "Learn" phase. In the event of an acknowledged warning, however, the system retrieves for instance detailed drilling reports, including problem solving steps (Category 2 data, DDR) in the retrieved, confirmed relevant cases, as recommendations primarily how to avoid and secondarily how to resolve the unwanted event.

Learn

Regardless of the "Acknowledge" outcome, the learning step is mandatory and vital for maintaining the case- base and tuning of the alarm signal module. If the user dismisses the proposed warning, and this is confirmed by no such unwanted event occurring, relevancy values following the paths that lead to the retrieval of the perceived relevant cases need be weakened, or this case is stored as a negative example (positive being an unwanted event case). Furthermore, the alarm threshold by which the "predict" signal passed need to be incrementally adjusted whenever such erroneous alarms occur.
If the justified warning is acknowledged, a similar but opposite case base update process is executed and the "predict" phase alarm threshold remains unchanged.

A more fine grained explanation of both modules is given in Chapters 3-6

CHAPTER 3

CASE BASED REASONING AND TROLLCREEK

TrollCreek is the system already present, represents the most extensive reasoning part and becomes the anchor point in this work, from which the others develop. It is therefore naturally discussed first amongst the processing elements of the theoretical design.
This chapter and the development stage in Chapter 5 "Artificial Neural Networks" is mutually dependant as the outcome of this analysis potentially changes the design criteria for the ANN-module. Furthermore, as discussed in the previous chapter, these modules have mutual functional dependencies.

The purpose of this chapter is neither to give an in- depth review of Case Based Reasoning, hereby called CBR, nor an elaborate description of the implemented CBR system, TrollCreek. The intention is to provide an overview of the reasoning paradigm and its implementation adequately to apprehend the weakness of the current system, thus placing this thesis' contribution in context.

3.1 Introduction

The motivation for using Case Based Reasoning is its inherent potential for problem solving in weak theory domains (see for instance [Aamodt, A., 1994]). The oil drilling domain is a wide-spread engineering discipline each unfeasibly modelled analytically and impossible to model jointly. This makes the oil drilling domain a typical candidate domain to which CBR might be applied.

At the highest level, CBR is a machine learning technique utilizing past experience in the process of solving new problems. There are several reasoning systems utilizing analogical reasoning and acquired knowledge through past experience. The next section is an attempt to clarify in which context our system resides.

3.2 CBR theory

The following section gives some pointers to the development of the CBR field

3.2.1 The CBR paradigm – background

The history and development of the CBR paradigm is found for instance in [Watson, I., 1997], chapter 2.2, a systematic overview of earlier systems leading up to the current system can be found in [Aamodt 1991] chapter 4 or [Aamodt, A., Plaza, E. 1994] (Part 2: "History of the CBR field"). The latter also describes and distinguishes this CBR system from other, similar analogical reasoning systems (Part 3: "Main types of CBR methods").

To summarize: CBR is an instance based, lazy learner: Among their distinctive characteristics, lazy learners postpone the computationally expensive steps upon deployment. Classification, or prediction in this context, thus becomes the computational endeavoring part [Mitchell, T. M. 1997] pp. 240-244 Furthermore, the paradigmatic CBR method contains a somewhat complexly internally organized case base comprised by information enriched cases. General background knowledge, varying both in extent and role in the reasoning process, support the specific collective knowledge represented by these cases. This background knowledge might support any or all phases of the CBR process and is typically utilized in the retrieval and possible modification of past solutions to fit new problems.

The next section narrows down the scope of the CBR system we use and elaborates on the essential CBR processes and essential sub- processes involved:

3.2.2 Incremental learning through the CBR cycle

CBR as described by [Aamodt, A., Plaza, E. 1994] is a cyclic process constituted by four main processes depictured in Figure 3.1

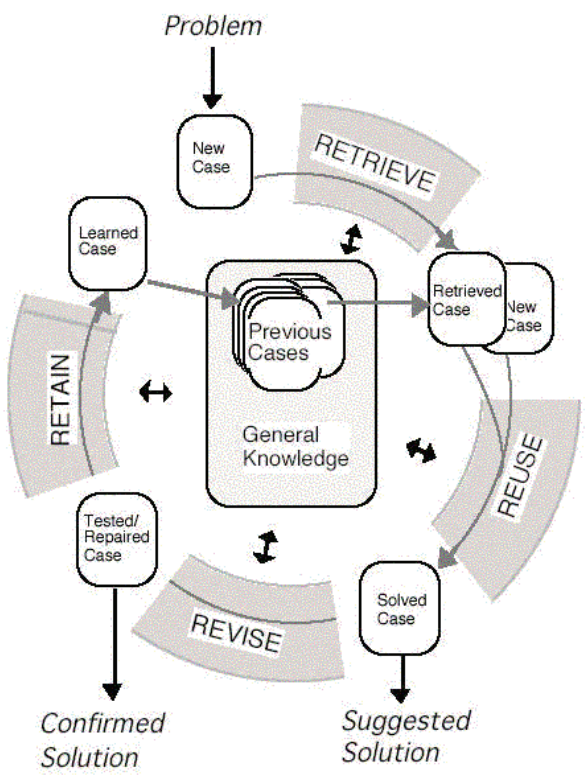


Figure 3.1: The 4 processes constituting the CBR cycle [A. Aamodt, E. Plaza, 1994]

The figure shows all key processes involved in solving a problem (Retrieve, Reuse and partly Revise) and the learning step (partly Revise and Retain). Furthermore, the cumulative system knowledge is represented as the static General Knowledge and the dynamic Case- base. Below are the main reasoning processes and their utilization of system knowledge further elaborated.

3.2.3 CBR reasoning steps

Presupposing that the system has been used sufficiently to have an adequate case base and general knowledge implemented, the following scenario explain the main reasoning steps involved:

Retrieve

A problem occurs, is structured and presented to the system as a new case. This initiates the first step of the reasoning process, the retrieval of similar cases from the case base. Unless there are directly matched cases with sufficient explanatory strength[6], general knowledge is guiding the search for similar cases and ranking the retrieved cases by relevance measures

[6] for similarity assessments, see for instance [Sun, Z., Finnie G. and Weber K., 2003]

Reuse

The retrieved case(s) now represent plausible solutions to the input case and within the reuse process, adaptation of the retrieved solutions is done, if needed, to adequately fit the new case. The success of adaptation rely both on the existence of adequately similar cases and sufficiently dense general knowledge.
Output of the Reuse phase is a solved case, to be tested in the next sub- process.

Revise

The solved case is tested either externally; by applying the solution to the real world environment to which it provide support or internally in a simulation. External testing might be interaction with a user or transmitting corrective or control signals to an external control module or physical device
Output of the Revise phase is a justified case, either confirmed or modified through this testing and forwarded to the Retain process.

Retain

This is the learning step in which possible acquired knowledge from the Revise phase is embedded in the case base, either as a new case or as a refinement of the existing case base.

For further elaboration on each of these reasoning steps, the article of [Aamodt, A., Plaza, E. 1994], chapters 3-8 is an excellent reference.

3.2.4 Knowledge- intensiveness in CBR

The degree of knowledge richness in a case varies from different CBR methods, as does the density of the embedded general knowledge. [Aamodt, A. 2004] suggest the following scaling of these dimensions:

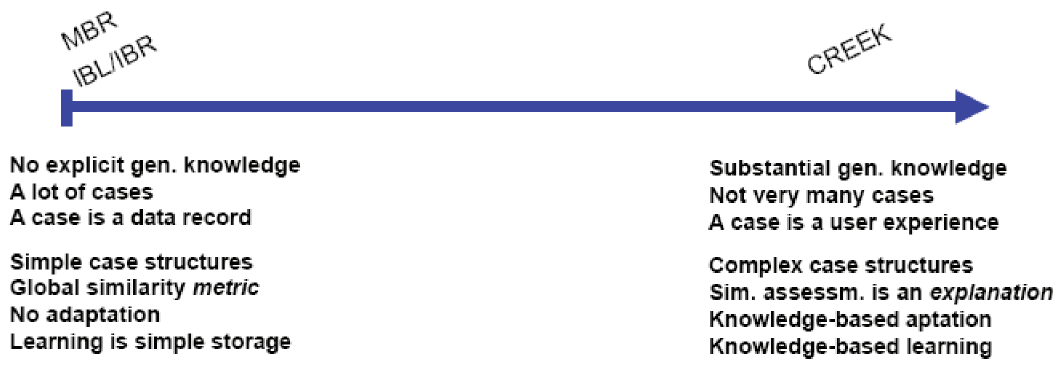


Figure 3.2: The knowledge intensiveness dimensions of CBR methods

From the left extreme, the CBR system resembles an advanced database storing feature records. On the extreme right, an extensive general knowledge model supplemented by few, complexly structured and elaborated cases, comprises the system knowledge. .
The details of ki-CBR (knowledge intensive CBR) is well described in [Aamodt, A. 2004].

An implemented and well tested CBR system, Creek, belongs somewhat to the right on this scale, as illustrated in Figure 3.2 and will be examined in the next sub- chapters.

3.3 TrollCreek –ki-CBR implementation

The ki-CBR was initially formalized and implemented in LISP, giving the Creek system. Background, design, framework and architecture of Creek are fully described in [Aamodt 1991].

A brief summarization: TrollCreek is an implementation of the CBR cycle. The knowledge intensiveness manifest itself as described in the previous sub- chapter: Firstly, it presuppose dense general knowledge, called domain knowledge, which is an interconnected semantic network of frames. Secondly, the cases are represented as knowledge enriched frames themselves. TrollCreek's reasoning abilities depend on the domain knowledge in all phases of the CBR cycle as each similarity assessments is an explanatory process and leave a trail of explanations to the inferred matches.

Java-Creek was the intermediate transfer from the original LISP- version to a more graphic and user friendly Java- implementation, which led to the current TrollCreek- version, but the original methodology remains.

Below is a simple example of how these explanatory trails are utilized when direct matching in the Retrieve- phase leave inadequate or ambiguous results:

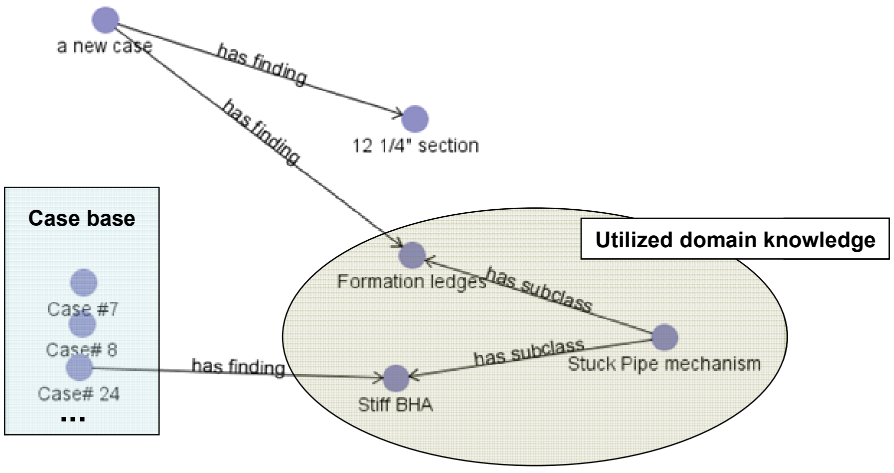


Figure 3.3: Explanations for classifying drilling problems utilizing domain knowledge

The figure illustrates a qualitative Creek mock- up model for the task of classifying drilling problems. Case 24 has a finding-to-finding explanation[7] between the new case and Case 24: Case 24 has a finding of "Stiff BHA" which is a subclass of "Stuck Pipe Mechanism" which has a subclass "Formation Ledges" which is a finding of the new case. Case 24 has a stuck pipe mechanism as does the new case and this enhances the matching strength[8] of Case 24.

3.4 TrollCreek and application in the oil drilling domain

Much work has been put into the current TrollCreek version for prediction and problem – solving in general, and especially within the oil drilling domain. Cross- discipline research at our university between the AI- group and the Department of Petroleum Engineering lead to an extensive domain model implemented by Pål Skalle and assisted by J. Abdullah.

As the general knowledge structure, this model is formalized as a single- tree top- down ontology of concepts and additional relations. This domain knowledge base has been under continuous development and enriched by causal and associational relationships as well as new concepts (approximately 2000 at the time this thesis was written).

A practical description of the oil domain model concepts and relations is described in [Skalle, P., Aamodt, A. Sveen, J. 1998].

Part of the current general, high- level ontology is shown in the figure below:

[7] Creek method to construct explanations for showing similarities between syntactically different values

[8] A combination of activation and matching strength [Aamodt, A., 1991] Chapter 6.2

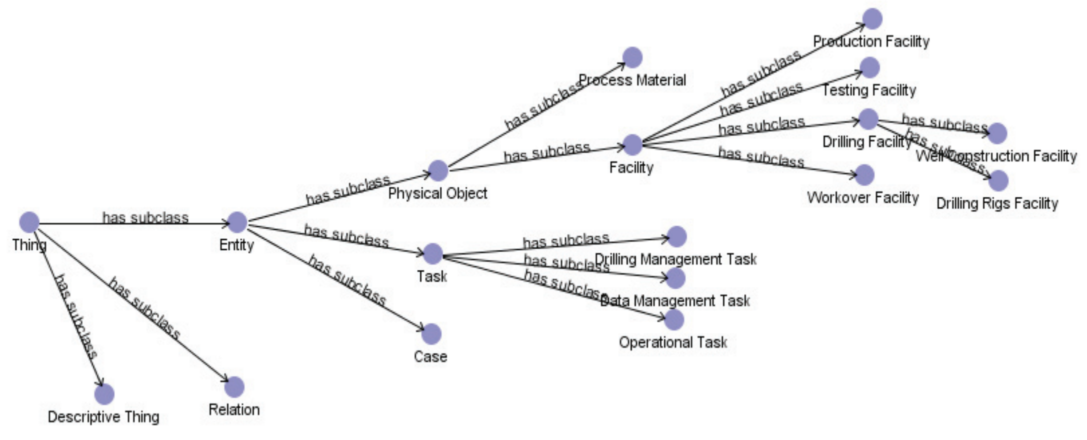


Figure 3.4: Part of the top- level ontology. Each concept is linked by a bi- directional structural relation of the type "has-subclass" to its lower level concept

Part of the ontology specifically designed for the oil drilling domain is shown in the figure below:

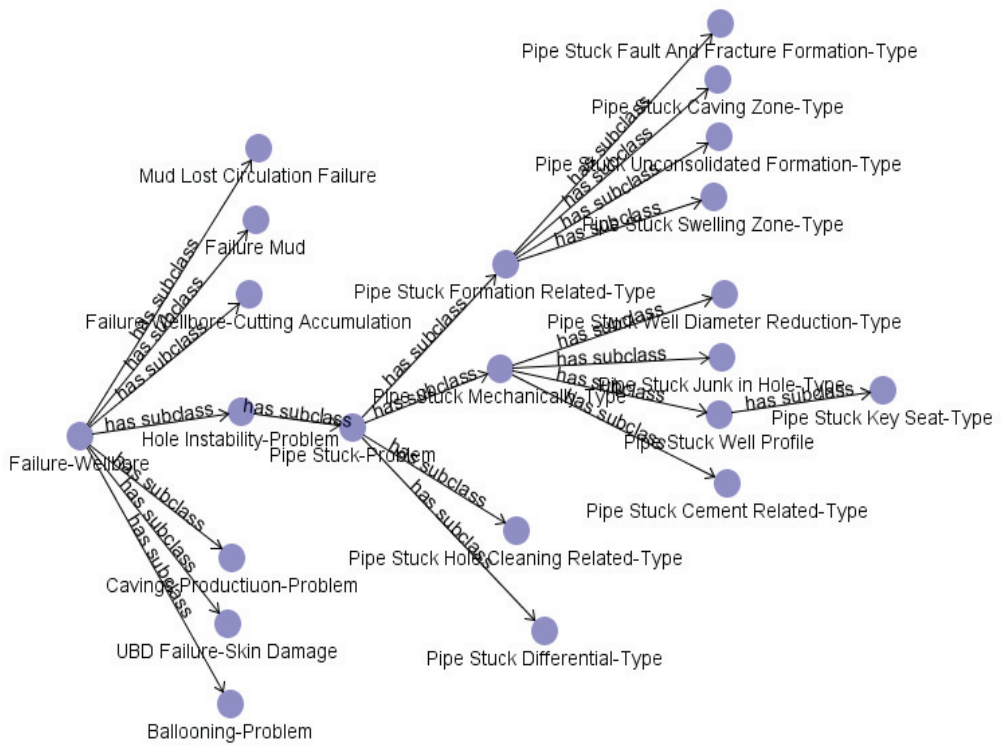

Failure-Wellbore
Mud Lost Circulation Failure
Failure Mud
Failure Wellbore-Cutting Accumulation
Hole Instability-Problem
Pipe Stuck-Problem
Caving-Productiuon-Problem
UBD Failure-Skin Damage
Ballooning-Problem
Pipe Stuck Formation Related-Type
Pipe Stuck Fault And Fracture Formation-Type
Pipe Stuck Caving Zone-Type
Pipe Stuck Unconsolidated Formation-Type
Pipe Stuck Swelling Zone-Type
Pipe Stuck Mechanically-Type
Pipe Stuck Well Diameter Reduction-Type
Pipe Stuck Junk in Hole-Type
Pipe Stuck Well Profile
Pipe Stuck Key Seat-Type
Pipe Stuck Cement Related-Type
Pipe Stuck Hole Cleaning Related-Type
Pipe Stuck Differential-Type
has subclass

Figure 3.5: Stuck pipe ontology. As for all structural relations, each concept is linked by a bi- directional structural relation of the type "has-subclass" to its lower level concept

A causal model of this ontology interlinking concepts via types of "causes"- relations is shown in Figure 3.6. Differentiating the causal strength is achieved by assigning numeric values in the range 0-1 to each relation.

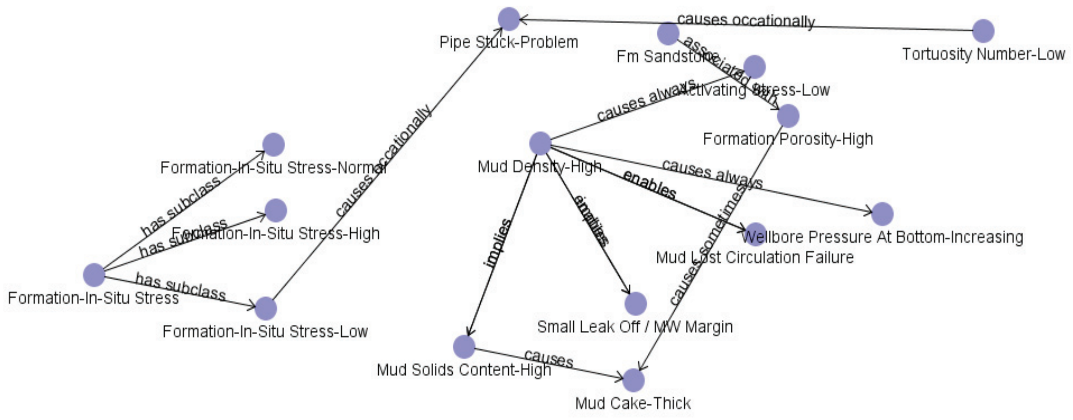


Figure 3.6: Part of the stuck- pipe causal relations, each concept linked by a bi- directional causal relation of differentiated strength to other concepts

3.5 Example of a matching process

The following example is a matching process using the current case base and domain model. Figure 3.7 shows a part of the TrollCreek case base. In this case, the embedded case LC 04, highlighted in a (red) circle, plays the role of the arisen problem already formalized as a case having status of "unsolved case":

:

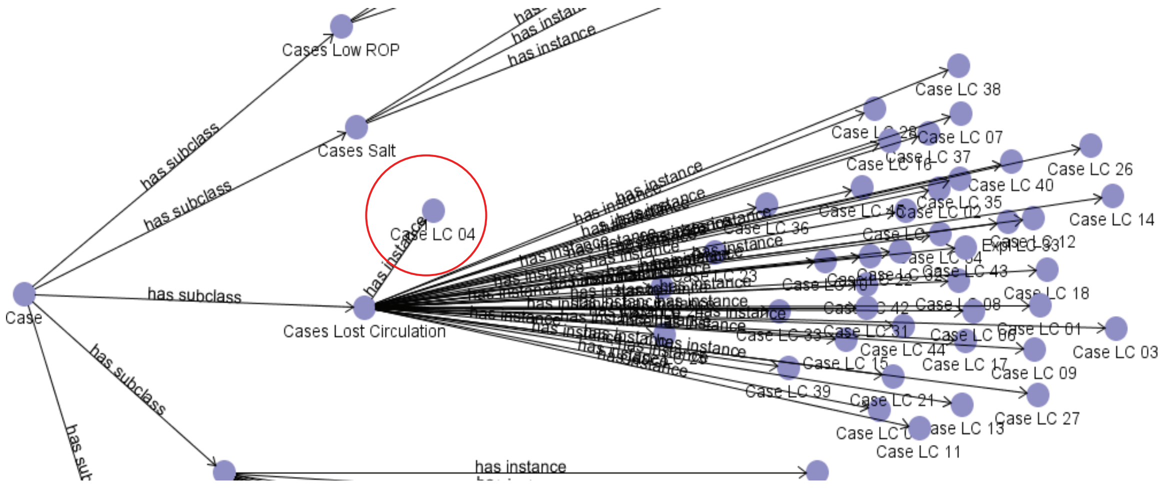
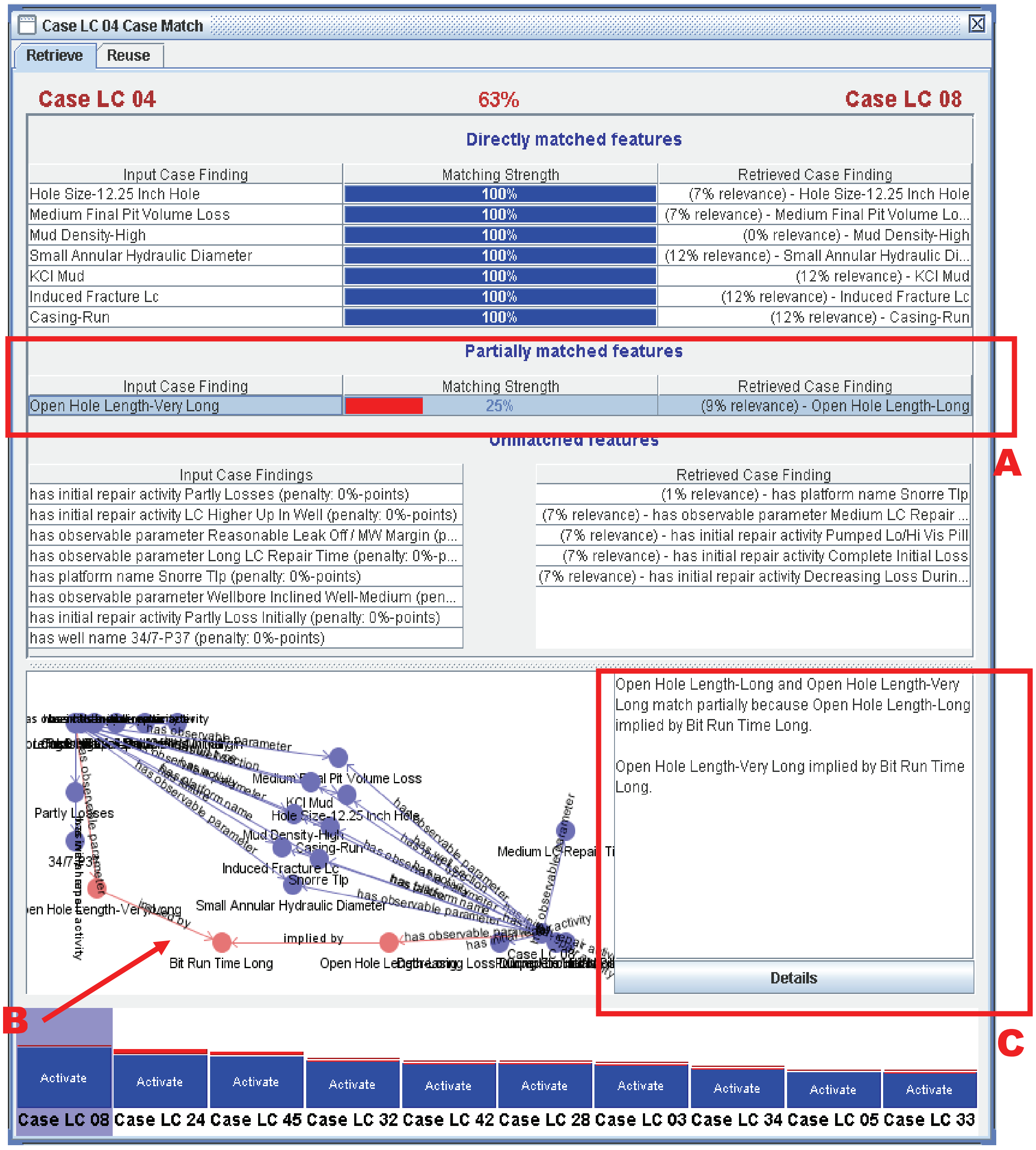


Figure 3.7: Part of the TrollCreek case base

Utilizing the TrollCreek interface, selecting our new case and running "match case" initiate the "Retrieve"- process described in 3.2.3. A ranked set of matched cases are retrieved, and the best matched case is shown in Figure 3.8. Three sections showing reasoning steps utilizing domain knowledge is highlighted in grey (red) rectangles and will be discussed separately:


Case LC 04 Case Match

Retrieve | Reuse

Case LC 04 63% Case LC 08

Directly matched features

Input Case Finding	Matching Strength	Retrieved Case Finding
Hole Size-12.25 Inch Hole	100%	(7% relevance) - Hole Size-12.25 Inch Hole
Medium Final Pit Volume Loss	100%	(7% relevance) - Medium Final Pit Volume Lo...
Mud Density-High	100%	(0% relevance) - Mud Density-High
Small Annular Hydraulic Diameter	100%	(12% relevance) - Small Annular Hydraulic Di...
KCl Mud	100%	(12% relevance) - KCl Mud
Induced Fracture Lc	100%	(12% relevance) - Induced Fracture Lc
Casing-Run	100%	(12% relevance) - Casing-Run

Partially matched features

Input Case Finding	Matching Strength	Retrieved Case Finding
Open Hole Length-Very Long	25%	(9% relevance) - Open Hole Length-Long

Unmatched features

Input Case Findings
has initial repair activity Partly Losses (penalty: 0%-points)
has initial repair activity LC Higher Up In Well (penalty: 0%-points)
has observable parameter Reasonable Leak Off / MW Margin (p...
has observable parameter Long LC Repair Time (penalty: 0%-p...
has platform name Snorre Tlp (penalty: 0%-points)
has observable parameter Wellbore Inclined Well-Medium (pen...
has initial repair activity Partly Loss Initially (penalty: 0%-points)
has well name 34/7-P37 (penalty: 0%-points)

Retrieved Case Finding
(1% relevance) - has platform name Snorre Tlp
(7% relevance) - has observable parameter Medium LC Repair ...
(7% relevance) - has initial repair activity Pumped Lo/Hi Vis Pill
(7% relevance) - has initial repair activity Complete Initial Loss
(7% relevance) - has initial repair activity Decreasing Loss Durin...

Open Hole Length-Long and Open Hole Length-Very Long match partially because Open Hole Length-Long implied by Bit Run Time Long.

Open Hole Length-Very Long implied by Bit Run Time Long.

Details

Figure 3.8: TrollCreek output of matching cases LC 04 to the case base

There are 7 directly and one partially matched features, indicated as rectangle "A", i.e. two syntactically different feature values explained similar through causal and structural paths in the domain model (rectangle "B") occasioning a contextual explanation for the partial match ("C"). An analogous, but more elaborately discussed example can be found in [Skalle, P., Aamodt, A. 2004].

3.6 TrollCreek analysis and discussion

As discussed in subchapters 3.3 through 3.5, the TrollCreek systems collective knowledge is comprised by

1. the ontology from which new and existing entities are structured
2. the interlinking of entities by the structural and causal relations
3. the individual entities
4. the case base

In the following sub- chapter, weaknesses identified in the oil drilling ontology with respect to points 2 and 3, ultimately affects all aspects of the domain model listed. Furthermore, the feasibility of the existing version of TrollCreek implementation with respect to real- time prediction is questioned. These are aspects discussed from both a qualitative and a pragmatic, computational point of view.

3.6.1 Discretization of operational parameters

First, a general definition of discretization is useful:

Definition 1:

"A discretization of a real-valued vector $\boldsymbol{v} = (v_1,\ldots,v_N)$ is an integer-valued vector $\boldsymbol{d} = (d_1,\ldots,d_N)$ with the following properties:

(1) Each element of $\boldsymbol{d}$ is in the set $\{0, 1,\ldots, D - 1\}$ for some (usually small) positive integer D, called the *degree* of the discretization (…)" [Hartemink, A. 2001]

Any discretization of a real- valued number involves loss of information, depending on several factors, as for instance the degree of discretization, distribution and standard deviation.

All feature values with which the TrollCreek system reasons need to be linguistic values. Consequently, discretization of all continuously valued variables is necessary. This affects all utilized "Category 1- data" as described in Chapter 2 or more specifically; all MWD variables.

Table 3.1 is an excerption of a TrollCreek documentation resource [TrollCreek I] Appendix A.3, with only minor cosmetic changes. Entities and entity values having yellow background indicate MWD variables and their discretization:

Table 3.1: Extract from a TrollCreek manual originally titled "Qualitative to quantitative transformation table", which means discretization of MWD parameters by mapping their qualitative linguistic variables (sub- entities) to quantitative values (d_i in the discretization definition).

Entity	qualitative-value	qual. to quant. transformation	comment
		Operational parameters:	
	normal-wob	wob-30	average last 30 m
wob	high-wob	wob-1 / wob-30 > 1.2	wob-1 = average last 1 m
	low-wob	wob-1 / wob-30 < 0.8	
	low-rpm	rpm-1 / rpm-30 <0.8	rpm-1 = average last 1 m
bit-rotating-speed	normal-rpm	rpm-30	average last 30 m
	high-rpm	rpm-1 / rpm-30 > 1.2	
	drilling-break	rop-1 / rop-30 > 1.5	
	low-rop	rop-1 / rop-30 < 0.8	
rop	normal-rop	rop-30	average last 30 m
	high-rop	rop-1 / rop-30 > 1.2	
	Increasing-rop	rop-5 / rop-30 > 1.1	
	decreasing-rop	rop-5 / rop-30 < 0.9	
	high-torque	torque-1 / torque-30 > 1.2	
torque	increasing-torque	torque-5 / torque-30 > 1.1	
	decreasing-torque	torque-5 / torque-30 < 0.9	
	medium-drag	drag-30	average last 30 m
hook-weight	high-drag	drag-1 / drag-30 > 1.2	
	increasing-drag	drag-5 / drag-30 > 1.1	
	decreasing-drag	drag-5 / drag-30 < 0.9	
	erratic-drag	> 5 (drag-max / drag-30 > 1.1)/20 s	
	tight-spot	drag-1 / drag-30 > 1.5	
	took-weight	weight-1 / weight-30 > 0.7	
bit-run-time	high-bit-run-time	t - t-start-bit-run > 100 hours	
	very-long-stands-still-time	> 2 h	
	constant-pump-rate	q-5 / q-30 = (0.9 - 1.1)	
	low-pump-rate	q-1 / q-30 < 0.8	
	increasing-pump-rate	q-5 / q-30 > 1.1	
	decreasing-pump-rate	q-5 / q-30 < 0.9	
pump-parameter	high-pump-rate	q-1 / q-30 > 1.2	
	high-pump-pressure	q-1 / q-30 > 1.2	
	high-hoisting-speed	> 2 m/s	
drill pipe speed	high-running-in-speed	> 2 m/s	
	low-running-in-speed	< 2 m/s	
	noticeable-back-reaming-time	< 1 h	
back-reaming-time*	long-back-reaming-time	> 1 h	
…	…	…	

*Given implicitly by other MWD parameters

TrollCreek's oil drilling domain model also involves about 30 relationships [Skalle, P., Aamodt, A.. 2004] some originate from, or point to, discretized MWD variables. These are relationships listed in [TrollCreek I] Appendix A.4, of which the following relations are examples:

Table 3.2: Non- structural relationships in the TrollCreek oil ontology

Entity 1	position	relation	Entity 2	position	condition (if)
Position indicate where in the upper onthology the two attributes are positioned. Condition means an "if" statement is atatched to the relationship. Comments are necessary if a relationship is not so well known. It may have been learned through a research work or through experience.					
...					
dog-leg	non.obs.p	enabled-by	high-wob	oper.p	pendelum-bha or wedge-bha
drilling-break*	oper.p	occurs in	leaking-fm	geol.fm	
drilling-break*	oper.p	implies	high-pore-pressure	wellb.p	
drilling-break*	oper.p	indicates	sandstone	geol.fm	
drill-pipe-speed	oper.p	causes	pressure-surge	non.obs.p	
...					

* Given implicitly by other MWD data

To summarize: Several TrollCreek oil drilling domain entities and relations are based on discretized real- valued numbers, solely depending on domain expert knowledge.

Each of the discretized MWD parameters produces a number of sub- entities equal to the degree of discretization, of which the top entry in
Table 3.1 is an example: wob is discretized into 3 sub- entities; "normal-wob", "high-wob" and "low-wob".
Keeping the number of operational related entities at a reasonable level necessitate a small D shown in Definition 1 and consequently, rough estimates of the MVD parameter values are produced. Moreover, MWD- parameters interactions are numerous, complex and significant [Valaas, I. 1997]. This means that the qualitative- to- quantitative transformations often rely on other parameters and should not be static ranges.

As discussed introductorily any discretization involves information loss. However, the above show that the discretization of MWD- parameters lead to *significant* loss of predictive information.

3.6.2 Predictive limitations

Two more aspects that are important limit the real- time predictive capabilities of the current implemented version of TrollCreek: The qualitative reasoning capabilities and usability as a real-time Category-1 data processing system.

Variable interactions

The ontology does allow for an expert to model parameter interactions, for instance the top row of Table 3.2: dog-leg (see Appendix 9.3: "Mechanisms causing the unwanted event Stuck Pipe") enabled-by high-wob.

However, MWD- variable interactions are significant. As discussed above, embedding the MWD parameters into the domain model This produces a high number of MWD entities as each MWD- parameter is represented by several syntactic sub- entities. This means modelling adequately based on any ontology using MWD parameters become an extremely complex and time- consuming task, solely relying on the domain experts.

In short: Maintaining real- time variable interactions is presently highly time- consuming and qualitatively questionable.

Real- time prediction

Category-1 data as described in Chapter 2 arrive every 5 seconds, thus "operational snapshots" are produced every 5th second. This means a purely Case- based system as TrollCreek would have to process a new case at this frequency through all stages of the CBR- cycle. TrollCreek is a lazy learner as described introductory, which would lead to a significant computational strain.

3.7 Conclusions

TrollCreek is well suited for capturing and utilizing expert knowledge, and has already an extensive oil drilling domain model readily available. The reasoning process utilizing this knowledge provide sufficient explanations for a fully qualified alarm state, as described in Goal 2.. Furthermore, the system, when fully implemented, provides a framework for sustained learning and continuous improvement.

Representational and practical limitations to the model based part of the system knowledge together with high computational expense make TrollCreek less suitable for real- time surveillance of drilling operations.

As discussed in the theory chapter, this CBR method was designed for knowledge intensive reasoning. Reaching the goals of this work as described in Chapter 1 require both knowledge and data intensive reasoning. The knowledge intensive part of this task is proven achievable using TrollCreek. For the data- intensive, predictive part of the task, however, the system is inadequate.

The next chapter introduces ANN as a natural extension to remedy the shortcoming described in this chapter. Prior to analysing the MWD data, a statistical pre- selection of relevant data is performed in the next chapter.

CHAPTER 4

ATTRIBUTE SELECTION BY STATISTICAL ANALYSIS

The statistical method described in this chapter is one of two possible analytical methods described in the "Analytical Methods" module of Figure 2.3.

This chapter describes a data pre- processing stage of selecting parameters significantly related to the unwanted event, and could have been a subchapter or an extensive addition to the training part of the ANN. However, employment of this method only occurs at the training stage of the prediction module, prior to training the ANN. The latter and the very extent of this method seen from a bookkeeping point of view justify a separate chapter,

Reducing the net complexity proved necessary, as pure black box testing including all drilling parameters gave no meaningful or interpretable results. Another strategy was initially employed: Attribute (MWD parameter) selection based on domain experts. The pre- selected attributes provided by domain experts proved to give poor predictions [Valaas, I, 2004]. An alternative method to identifying relevant predictors was necessary.

MLR (Multivariate Logistic Regression) is a multivariate technique for estimating the probability that an event occurs, for example an unwanted drilling related event.

Considering the functional requirements of 2.2.2, not many statistical methods seem well suited for this analysis.

Regression and multiple regression, for instance, require a continuous dependant variable and a linear mapping between instance and classification, or using statistical terminology: Independent and dependant variables. MLR presuppose neither, qualifying this method as a plausible statistic tool for analysing the dicotome, skewed distributed drilling data.

4.1 MLR theory

[Hosmer & Lemeshow, 1989; Allison, 1999; Kleinbaum,1994; Norusis,1993; Fox, 2000.]

For the case of a single independent variable, the logistic regression model can be written as:

Equation 4.1: Logistic regression model

$$P(\text{event to occur}) = \frac{e^{B_0+B_1x}}{1+e^{B_0+B_1x}} = \frac{1}{1+e^{-(B_0+B_1x)}}$$

x is the independent variable and B_0, B_1 are coefficients estimated from the training instances.

For more than one independent variable, the model rewritten handles multiple independent variables:

Equation 4.2: Logistic regression model, multiple independent variables

$$P(\text{event to occur}) = \frac{e^{Z}}{1+e^{x}} = \frac{1}{1+e^{-Z)}}$$

Here Z is the linear combination of $B_0 + B_1x_1 + B_2x_2 + \ldots + B_nx_n$

Below are listed some important results from the MLR (multiple logistical regression) model:

Equation 4.3: Equal probabilities

$$Z = 0, \quad \frac{1}{1+e^{-Z}} = \frac{1}{1+e^{0}} = \frac{1}{1+1} = 0,5$$

A zero valued Z- value means equal probability for occurrence/no occurrence (same as flipping a coin)

Equation 4.4: increasing event probability

$$Z > 0, \quad \frac{1}{1+e^{-Z}} > 0,5 \ (e^{-Z} < 1)$$

This implies that the probability of an event occurring (e.g. a drilling related unwanted event) increase with an increasing Z- value, and similarly:

Equation 4.5: Decreasing event probability

$$Z < 0, \quad \frac{1}{1+e^{-Z}} < 0,5 \ (e^{-Z} > 1)$$

The probability of an event occurring (e.g. a drilling related unwanted event) decrease with an decreasing Z- value

In linear regression, one method of estimating the parameters of the model is using OLS ("ordinary least squares"), OLS is the selection of regression coefficients that result in the smallest sums of squared distances between the observed and the predicted values of the dependent variable.

In logistic regression, however, the parameters of the model are estimated using the maximum-likelihood method, that is, the coefficients $(B_0 + B_1 + B_2 + \ldots + B_p)$ making the observed results most "likely" are selected. Since the logistic regression model is nonlinear, an iterative algorithm is necessary for parameter estimation.
The outputs from the analyses report the size of the Bs, standard error and significance level for each of the Bs, and the Wald statistic (see section 4.1.2), and these estimates give the bases to assess the relative strength of the different factors or variables.

A summarization could be; *logistic regression is a tool to mapping if and which variables might predict unwanted event and estimating their relative strength.*

4.1.1 Estimation of the coefficients: Maximum likelihood and log likelihood (LL).

[Friel, C.M., 2001]

The maximum likelihood function has been developed for probit and logit regression models. Specifically, the loss function for these models computes as the sum of the natural logs of the logit or probit likelihood L_1 so that:

Equation 4.6: Maximum likelihood function

$$\ln(L_1) = \sum_{i=1}^{n} [y_i * \ln(p_i) + (1-y_i) * \ln(1-p_i)] ,$$

where $\ln(L_1)$ is the natural logarithm of the (logit or probit) likelihood (log-likelihood) for the current model, y_i is the observed value for example *i*, and finally; p_i is the expected (predicted or fitted) probability (between 0 and 1).

The log-likelihood of the null model (L_0), that is, the model containing the intercept only (and no regression coefficients) is computed as:

Equation 4.7: log- likelihood null model

$$\ln(L_0) = n_0 * \ln\frac{n_0}{n} + n_1 * \ln\frac{n_1}{n}$$

where $\ln(L_0)$ is the natural logarithm of the (logit or probit) likelihood of the null model (intercept only), n_0 is the number of observations with a value of 0 (for instance not unwanted event), n_1 is the number of observations with a value of 1 (for example unwanted event) and n is the total number of observations (training examples). The null model is a possible initial performance measure in evaluating the effect of parameters explanation strength to the dependant variable.

Given a set of n training examples with target functions $y_1, y_2, \ldots, y_n$, $y_i \in \{0,1\}$ with success probability P, then the log likelihood can be written :

Equation 4.8: Log likelihood model

$$\ln(L) = y_1 \ln(P) + (1-y_1)\ln(1-P) + y_2 \ln(P) + (1-y_2)\ln(1-P) + \ldots + y_n \ln(P) + (1-y_n)\ln(1-P).$$

The maximum likelihood estimator is the value of P which maximizes this; which can be shown is the sample proportion of 1's, $(y_1 + y_2 + \ldots + y_n)/n$.

In the logistic regression model, $P_{\mathbf{x}} = 1/[1+exp(-\boldsymbol{B_0} - \boldsymbol{B}^T\mathbf{x})]$, where B_0 is the constant and $\boldsymbol{B}$ is the vector of logistic regression parameters to be estimated. This is done by maximizing the likelihood by numerical methods (different iterative methods).

High LL signifies good model fit, i.e. good fit between "predicted" and observed data.
-2LogLikelihood is used to compare the fit between two different models, for example a null model (a model with only the intercept or a model without any parameters). Low values of
-2LL signify good fit.

Testing a reduced model against a full model is based on -2LL, where LL is the natural log of the maximized likelihood. It is based on the fact that$(-2LL)_{\text{full}}$ - $(-2LL)_{\text{reduced}}$ is for large *n* distributed approximately as chi-square with the number of degrees of freedom equal to $k_{\text{full}} - k_{\text{reduced}}$, where these *k*'s are the numbers of explanatory variables in the two models. The chi- square distribution table shows whether the full model represents a significant improvement, compared to the reduced model, or not.

4.1.2 Testing the coefficients of a hypotheses

One of several statistics to test if a coefficient is significantly different from zero (normally when the probability of zero is smaller than 5%) is the Wald statistics. When a variable has degrees of freedom equal to one (i.e. only one variable is tested), the Wald statistics is the squared ratio of the b-coefficient to its standard error; $(B/SE)^2$. However, the Wald statistics overestimate the standard errors when the absolute value of B becomes large. Therefore, a better way is to build models with and without the variable in question and comparing the models using -2LL (see above).

4.1.3 Interpreting the coefficients.

The odds of an event occurring is the ratio of the probability that it will occur to the probability that it will not:

Equation 4.9: Odds of an event

$$\text{Odds} = \frac{\text{P(event)}}{\text{1- P(event)}} = \frac{\dfrac{e^z}{1+e^z}}{1-\dfrac{e^z}{1+e^z}} = e^z,$$

where $Z = b_0 + b_1x_1 + b_2x_2 + \ldots + b_kx_k$ Accordingly:

$$1n(\text{odds}) = \ln e^Z = Z$$

In other words; Ln of the odds transforms the nonlinear function odds to a linear function:

Equation 4.10: Linear transformation of th odds function

$$Z = b_0 + b_1x_1 + b_2x_2 + \ldots + b_kx_k$$

4.1.4 MLR theory summarized

1. Wald statistic is the square of the ratio of the B - coefficient to its standard error.
   P-value is calculated for each Wald statistic. High values of B give too high standard errors; therefore, the results should be interpreted with some caution. If Wald statistic is high it is preferable to use -2 Log Likelihood estimation of variable relevance.

2. -2 Log Likelihood is chi square distributed. The difference between two models in
   -2LL and the increase or decrease in degrees of freedom shows whether one model is significantly better than the other is.

3. The b-coefficient of a variable is the growth in the log odds when that variable increases with one.

4. Exp(B) is a factor of increase in the odds when a variable increases with one (the "new" odds divided by the "old" odds) .

4.1.5 Assumptions

Compared to "classical" multiple regression" logistic regression does neither assume continuous dependent variable nor linear relationship between the dependent variable and the independents. The dependent variable need not be normally distributed and the dependent variable need not to be homoscedastic (same variance for each level of the independents). Furthermore, logistic regression does neither assume normally distributed error terms nor require that the independent variables are at ratio or interval level, i.e. ordinal or categorical variables may be used as independents.

However, logistic regression does assume that the model is correctly specified, i.e.:

1. the true conditional probabilities are a logistic function of the independent variables
2. no important variables are omitted
3. no extraneous variables are included
4. the independent variables are measured without error.

Moreover, logistic regression assumes

1. the instances or points of measurement are independent and
2. the independent variables are not linear combinations of each other.

Perfect multi- collinearity makes estimation impossible, while strong multi- collinearity makes estimates imprecise.

4.2 MLR Analysis of the MWD data

The following analysis is based on Measurement While Drilling variables, hereby called MWD, in an attempt to identify MWD parameters most suited to predict a selected unwanted event. The selected unwanted event is stuck pipe. Variables considered irrelevant from a substantial and theoretical point of view are already removed from the data set. The goal of the following logistic regression analyses is to reduce the data set further to include only the most relevant variables, i.e. variables that both from a theoretical an statistical point of view ought to be included in subsequent ANN analysis.

As a preliminary analysis, a sample was collected during a period of 10 hours before stuck pipe occurred and 10 minutes during the incident. This gave an extremely low "stuck pipe" frequency of the dependant variable (less than 1,5%). MLR does tolerate skewed distributed data, however the results were difficult to interpret and gave little meaning. Relations between the independent and dependant variables seem too weakened to be useful over such a long sample interval.

Meanwhile, the purpose of this analysis was not to predict stuck pipe, merely to identify variables affecting the stuck pipe mechanism, therefore data used in the following analyses were collected 20 minutes before and 10 minutes during stuck pipe. Of the 2086 data points 680 were recorded as stuck pipe, and 1406 as stuck pipe, i.e. still a skewed dependent variable.

4.2.1 Descriptive analysis

All drilling related, significant abbreviations in these analyses have capitalized names and are explained in Appendix 9.1. Other domain specific terms are unimportant in this context.

The tables below, Table 4.1: Means, SD, skewness and kurtosis, show means, standard deviations, skewness, and kurtosis.

Table 4.1: Means, SD, skewness and kurtosis

		stuck	Depth	BitMDepth	blockpos	AvgROP	AvgHookld	calchkl
N	Valid	2086	2086	2086	2086	2086	2086	2086
	Missing	0	0	0	0	0	0	0
Mean		,32598	4321,5748	4075,1286	20,6636	3,2066	1014208,4868	895716,3805
Std. Deviation		,46885	1717,8573	1616,3981	9,1432	9,1083	468200,2180	218178,8753
Skewness		,743	-.054	,134	-.562	2,490	1,620	-1,020
Kurtosis		-1,449	-1,301	-1,295	-1,010	4,206	4,014	,154

Table cont.

		avgWOB	AvgTorq	MaxTorq	AvgRPM	AvgPumpP	avgdrvol	drlvoch
N	Valid	2086	2086	2086	2086	2086	2086	2086
	Missing	0	0	0	0	0	0	0
Mean		2086	8938,5173	16381,2012	1,3966	11321146,5812	59,3070	-5,21747172
Std. Deviation		420940,4452	9359,0853	11223,47201	1,38903	9414438,38470	23,4455	15,5385
Skewness		-1,118	,542	-,311	,499	,124	-,016	-,739
Kurtosis		2,979	-,571	-,994	-1,056	-1,558	-1,451	-,192

Table cont.

	MudDensOut	MudDensIn	ReturDeapth	avgas	DeltaFlow

N	Valid	2086	2086	2086	2086	2086
	Missing	0	0	0	0	0
Mean		1579,0939	1579,0939	4137,9443	,0027	-,0009
Std. Deviation		76,6442	76,6442	1612,0263	,00483	,01371
Skewness		-1,698	-1,698	,006	1,884	-,561
Kurtosis		1,448	1,448	-1,287	2,507	1,058

As discussed in the chapter "Data Description" and as the table emphasises, the variables were measured by very different scales of measurement. While the mean of "avgas" was 0,00483, and "DeltaFlow" 0,0009, the mean of "AvgPumpP" was 11321146,5812.. That will not influence the relative effect of the different independent variables on the dependent ("stuck pipe") provided a sufficient numbers of decimals but the figures regarding regression coefficients or other estimates from the logistic regression analyses will become very small.

The variables "AvgROP", "AvgHookld", and "avgas" are strongly skewed to the right and have high positive kurtosis, while the variables "MudDensOut", "avgwob" are skewed to the left. However, logistic regression analyses are not sensitive to strong deviations from normal distribution.

Two variables "MudDensIn" and "MudDensOut" have identical figures in mean, standard deviation, skewness, and kurtosis, indicating that the two variables are identical. If their zero-order correlation is one, one of them is excluded from the analyses.

The table below (Table 4.2) shows zero-order ("ordinary") correlation among input-variables in the following logistic regression analyses. High correlations among several of the variables may cause problems related to multi- collinearity in the subsequent regression analyses. Multi-collinearity is the inter- correlation of independent variables in the regression model (equation). While simple or zero-order correlations tell something about multi- collinearity, the preferred method in linear multiple regression analysis is to regress each independent variable on all the other independent variables in the equation. However, logistic regression does not assume linear relations among the variables, consequently methods to reveal multi- collinearity used in linear multiple regression are useless. Thus, inspection of the correlation matrix to reveal variables which are linearly or near linearly related to another, is one of few other possibilities to se if two or several independent variables are linearly related (i.e. one variable can be expressed as a linear combination of one or several other variables). The variable "Depth", "BitMDepth", and "ReturDeapth" are obvious at risk with inter- correlations ranging from 0,980 to 0, 997. Likewise, the variables "avgas", "AvgROP", and "calchkl" are highly associated (r > 0,7). The variables

"MudDensOut" and "MudDensIn" are linearly combinations of each other (r=1.00), and only one of them will be included in the model.

Multi- collinearity (complete multi- collinearity exists when two or several independent variables is perfectly inter-correlated) has adverse effect on regression analysis (also logistic regression) and may render the results un- interpretable. In the calculation of the regression coefficients for each variable, the other variables are partialed out, or controlled. Thus, high multi- collinearity leads to a reduction in the magnitudes of regression coefficients. At worst, it may turn out that most or even all of the coefficients are statistically not significant despite the fact that the variables' joint effect is. High multi- collinearity also has an adverse effect on the stability of regression coefficients because of it's increasing effect on standard errors.

1. There are several methods to handle data with high multi- collinearity. The simplest but not the best way is to exclude the variable or variables with highest inter- correlations and then reanalyse the data. However, if the model is correctly specified in advance, exclusion of variables from the model means misspecification.
2. A better method is to group strongly inter correlated variables in blocks, and analyse the block effects rather than the effects of the individual independent variables. In the case of "block analyses", the individual independent variables are seen as indicators to a latent variable or factor, i.e. the focus of the analysis is primarily on the joint contribution of the inter- correlated independent variables in the block with respect to the odds that an event will occur.

Based on the figures in the correlation matrix (Table 4.2below), the strategy of "block" analyses will be followed.   .

Table 4.2: Zero- order correlations

	stuck	Depth	BitMDepth	blockpos	AvgROP	AvgHookld	calchkl	avgWOB	AvgTorq	MaxTorq	AvgRPM	AvgPumpP	avgdrvol	drlvoch	MudDensOut	MudDensIn	ReturDeapth	avgas
Stuck	1	,025	,022	,151**	,150**	,318**	,011	,222**	,106**	,072**	,329**	,114**	,017	,007	,017	,017	,024	,016
Dept	,025	1	,980**	,290**	,078**	,439**	,340**	,345**	,217**	,245**	,059**	,177**	,400**	,111**	,334**	,334**	,991**	,052*
BitMDepth	,022	,980**	1	,291**	,089**	,450**	,506**	,259**	,152**	,194**	,004	,198**	,426**	,002	,300**	,300**	,997**	,094**
blockpos	,151**	,290**	,291**	1	,059**	,007	,157**	,034	,068**	,112**	,003	,108**	,098**	,072**	,118**	,118**	,293**	,021
AvgROP	,150**	,078**	,089**	,059**	1	,145**	,738**	,400**	,234**	,192**	,354**	,136**	,038	,408**	,188**	,188**	,025	,738**
AvgHookld	,318**	,439**	,450**	,007	,145**	1	,324**	,755**	,054*	,188**	,023	,120**	,058**	,049*	,010	,010	,440**	,155**
calchkl	,011	,340**	,506**	,157**	,738**	,324**	1	,196**	,190**	,034	,307**	,105**	,039	,421**	,247**	,247**	,438**	,750**
avgWOB	,222**	,345**	,259**	,034	,400**	,755**	,196**	1	,070**	,231**	,227**	,188**	,078**	,287**	,112**	,112**	,288**	,292**
AvgTorq	,106**	,217**	,152**	,068**	,234**	,054*	,190**	,070**	1	,485**	,382**	,103**	,140**	,039	,012	,012	,179**	,307**
MaxTorq	,072**	,245**	,194**	,112**	,192**	,188**	,034	,231**	,485**	1	,273**	,229**	,114**	,339**	,263**	,263**	,204**	,157**
AvgRPM	,329**	,059**	,004	,003	,354**	,023	,307**	,227**	,382**	,273**	1	,547**	,161**	,455**	,028	,028	,029	,329**
AvgPumpP	,114**	,177**	,198**	,108**	,136**	,120**	,105**	,188**	,103**	,229**	,547**	1	,487**	,151**	,178**	,178**	,207**	,058**
avgdrvol	,017	,400**	,426**	,098**	,038	,058**	,039	,078**	,140**	,114**	,161**	,487**	1	,342**	,495**	,495**	,445**	,192**
drlvoch	,007	,111**	,002	,072**	,408**	,049*	,421**	,287**	,039	,339**	,455**	,151**	,342**	1	,091**	,091**	,037	,335**
MudDensOut	,017	,334**	,300**	,118**	,188**	,010	,247**	,112**	,012	,263**	,028	,178**	,495**	,091**	1	1,000**	,323**	,243**
MudDensIn	,017	,334**	,300**	,118**	,188**	,010	,247**	,112**	,012	,263**	,028	,178**	,495**	,091**	1,000**	1	,323**	,243**
ReturDeapth	,024	,991**	,997**	,293**	,025	,440**	,438**	,288**	,179**	,204**	,029	,207**	,445**	,037	,323**	,323**	1	,028
avgas	,016	,052*	,094**	,021	,738**	,155**	,750**	,292**	,307**	,157**	,329**	,058**	,192**	,335**	,243**	,243**	,028	1
DeltaFlow	,074**	,226**	,198**	,158**	,112**	,093**	,168**	,008	,114**	,032	,283**	,408**	,285**	,311**	,116**	,116**	,224**	,394**

** Correlation is significant at the 0.01 level 2tailed.

* Correlation is significant at the 0.05 level 2tailed.

a Listwise N=2086

4.2.2 MLR analyses

Table 4.3: Initial MLR analysis

Block number	Variables in	Variables out	-2LL	-2LL change	df	Df change	P	P change
0 (L_0)	Constant		2633,77		1			
1 (L_1)	Depth, BitMDeptht, AvgROP		2430,48	-203,29	4	3	.000	<.001
2 (L_2)	AvgHookld, avgWOB		2221,94	-208,54	6	2	.000	<.001
3 (L_3)	calchkl, MaxTorq		2149,94	-9,00	8	2	,000	<,025
4 (L_4)	MudDensOut*		2148,60	-1,34	9	1	.000	NS
5 (L_5)	AvgRPM, AvgPumpP, AvgTorq, avgdrvol, drivoch, ReturDeapth, avgas, DeltaFlow		1014,41	-134,19	16	8	.000	<.001

* "MudDensIn is excluded from the block owing to the fact that this variable is correlated with MudDensOut with a correlation of 1.

Column one includes the block numbers while column two lists the variables in each block. Column three lists variables deleted from the equation while column four and five include -2 Log Likelihood and -2 log Likelihood change respectively. -2LL change shows the reductions in -2LL for each variable or block successively entered in the model. Column six and seven show the degrees of freedom and changes in degrees of freedom, respectively, while column eight and nine show the probability of no effect of the independent variables or blocks of independent variables on the dependent variable (stuck pipe). P > .05 is regarded as insignificant.

The -2 Log Likelihood is chi square distributed, and e.g. a reduction in -2LL greater than 3.84 with a reduction of one degree of freedom (introducing an additional variable) means that the model is significantly improved at the 5% level.

Block number 0 represent the regression equation with only the constant included, introductorily called L_0[9]. This model represent the baseline when deciding if and to which degree model L_1 (Block 1; Depth, BitMDeptht, AvgROP) represents a significant improvement. Inserting block 1 in the model reduced -2LL from 2633,77 to 2430,48, i.e. -2LL was reduced by 203,29 and with a 3 degrees of freedom increase, inserting block 1 the model gave a highly significant improvement of the model.
In model L_2 block two (AvgHookld, avgWOB) was entered. In this case model L_1 served as a baseline to decide if entering block two in the model results in a significant improvement. As shown in the table above, the -2LL was reduced from 2430,48 to 2221,94, a reduction of -208,54. With an increase of two degrees of freedom, the model represented a highly significant improvement compared with model L_1.
Block three gave a small, yet significant contribution to the model fit. Therefore it could be that one of the two variables "calchkl" and " MaxTorq" was insignificant.
Block four consisted of only the variable "MudDensOut" and gave no significant contribution to the model fit. However, multi- collinearity results in complexity in the relation patterns and unsteady estimates. Therefore, it was decided to keep the variable in the model.

Block five was composed by variables weakly to moderately intercorrelated. Decisions regarding "candidate- variables" left out from the model bases on the Wald statistics in Table 4.4 below.

Table 4.4: MLR equation variables, first analysis

	B	S.E.	Wald	df	Sig.	Exp(B)
Depth	,089	,016	30,086	1	,000	1,093
BitMDepth	-,171	,017	100,523	1	,000	,842
AvgROP	-3,960	,489	65,551	1	,000	,019
AvgHookld	,000	,000	30,530	1	,000	1,000
avgWOB	,000	,000	9,996	1	,002	1,000
calchkl	,000	,000	1,752	1	,186	1,000
MaxTorq	,000	,000	19,948	1	,000	1,000

[9] $\ln(L_0) = n_0*(\ln(n_0/n)) + n_1*(\ln(n_1/n)) = 1406\ln(1406/2086)+680\ln(680/2086)=-1316,8858$.
-2LL= -2(-1316,8858) = 2633,77, confer -2LL in block 0 above.

MudDensOut	,085	,015	30,949	1	,000	1,089
avgas	-26,068	55,056	,224	1	,636	,000
AvgRPM	-4,556	,311	214,642	1	,000	,011
AvgPumpP	,000	,000	1,741	1	,187	1,000
AvgTorq	,000	,000	59,532	1	,000	1,000
avgdrvol	,030	,042	,512	1	,474	1,031
drlvoch	,093	,092	1,007	1	,316	1,097
ReturDeapth	,073	,015	24,593	1	,000	1,076
DeltaFlow	-288,832	24,771	135,957	1	,000	,000
Constant	-135,962	35,492	14,675	1	,000	,000

a Variable(s) entered on step 1: Depth, BitMDepth, AvgROP, AvgHookld, avgWOB, calchkl, MaxTorq, MudDensOut, avgas, AvgRPM, AvgPumpP, AvgTorq, avgdrvol, drlvoch, ReturDeapth, DeltaFlow.

The following variables were decided removed; "avgas", "avgdrvol", "drlvoch", and "calchkl" and are found in the third column "Variables out" in Table 4.5. Exclusion of these four parameters resulted in an increase in -2 log likelihood from 1014,41 to 1017,27, an increase in -2LL of 2,86 and a gain of 4 degrees of freedom and gave a clearly better model fit (P≈0).

Table 4.5: Second MLR analysis

Block number	Variables in	Variables out	-2LL	-2LL change	df	Df change	P	P change
0	Constant		2633,77		1			
1	Depth, BitMDeptht, AvgROP		2430,48	-203,29	4	3	.000	<.001
2	AvgHookld, avgWOB		2221,94	-208,54	6	2	.000	<.001
3	calchkl, MaxTorq		2149,94	-9,00	8	2	,000	<,001
4	MudDensOut*		2148,60	-1,34	9	1	.000	NS
5	AvgRPM, AvgPumpP, AvgTorq, avgdrvol, drivoch, ReturDeapth, avgas, Delta Flow		1014,41	-134,19	16	8	.000	<.001
6		avgas drlvoch avgdrvol calchkl	1017,27	+2,86	12	-4	.000	NS

* "MudDensIn is excluded from the block owing to the fact that this variable is correlated with MudDensOut with a correlation of 1.

An overview of the variables remaining in the model is given in Table 4.6. All included variables are significant at the 5% level. On the other hand, many of the variables are still highly inter- correlated, and it is therefore difficult or not to say impossible to rank these parameters by relevance or importance. However, the purpose of this MLR analysis was primarily to identify variables significantly related to stuck pipe.

Table 4.6: MLR equation variables, second analyses

	B	S.E	Wald	Df	Sg.	Exp(B)
Depth	,091	,007	174,943	1	,000	1,096
BitMDepth	-,160	,011	198,172	1	,000	,852
AvgROP	-4,123	,292	199,301	1	,000	,016
AvgHookld	,000	,000	32,989	1	,000	1,000
avgWOB	,000	,000	12,626	1	,000	1,000
MaxTorq	,000	,000	210,128	1	,000	1,000
MudDensOut	,068	,005	222,922	1	,000	1,070
AvgRPM	-4,396	,273	259,963	1	,000	,012
AvgPumpP	,000	,000	13,464	1	,000	1,000
AvgTorq	,000	,000	57,954	1	,000	1,000
ReturDeapth	,061	,005	146,415	1	,000	1,063
DeltaFlow	-296,277	21,653	187,220	1	,000	,000
Constant	-92,518	6,030	235,390	1	,000	,000

a Variable(s) entered on step 1: Depth, BitMDepth, AvgRPM, AvgHookld, avgWOB, MaxTorq, MudDensOut, AvgROP, AvgPumpP, AvgTorq, ReturDeapth, DeltaFlow

Table 4.7 below is a summary of the final analysis. The table shows approximately how many percent of the variance in stuck pipe is explained by the independent variables in the model. However, these figures are not analogous to R squared in multiple regression (see Note below the table). The figures most comparable with the "traditional" R squared values are Nagelkerke R Square. By this estimate approximately 75% of the variance in stuck pipe is explained by the remaining independent variables.

Table 4.7: Model summary, second analysis

Step	-2 Log likelihood	Cox & Snell R Square	Nagelkerke R Square
1	1017,266(a)	,539	,752

Note: **The Cox & Snell R-square** is a generalized coefficient of determination, used to estimate the proportion of variance in the dependent variable which is explained by the predictor (independent) variables. The Cox & Snell R-square is based on the log likelihood for the model compared to the log likelihood for a baseline model.

The **Nagelkerke R-square** is an adjusted version of the Cox & Snell R-square. The Cox & Snell R-square has a maximum value of less than 1, even for a "perfect" model. The Nagelkerke R-square adjusts the scale of the statistic to cover the full range from 0 to 1. (SPSS, version 13)

Table 4.8 below shows how well the model is fitted to the data. The null hypothesis says that it is no significant difference between the data and the model. If "Sig" > .05 this hypothesis is not rejected. Unfortunately, the "sig" is < .001, meaning there is a significant difference between model and data. The chi square is, however, highly dependent on the sample size. With 2086 measuring points, the model may have a god fit despite high chi square.

Table 4.8: Hosmer and Lemeshow test, second analysis

Step	Chi-square	df	Sig.
1	172,246	8	,000

How well do the model distinguish between stuck and not stuck pipe? Table 4.9 shows the number of observed stuck pipe instances related to the numbers of "predicted" stuck pipe instances, hence demonstrating how well the model distinguishes between the two categories. 547 of all 680 stuck pipe examples or 80,4 % are correctly classified, while 1341 of all 1406 not stuck pipe examples are correctly classified. Thus, the model seems to be a good stuck pipe classification tool, i.e. the selected variables seem to be strongly related to this unwanted event.

Table 4.9: Classification table(a), second analysis

Observed		Predicted		
		Stuck		Percentage Correct
		,0000	1,0000	
Stuck	,0000	1341	65	95,4
	1,0000	133	547	80,4
Overall Percentage				90,5

a The cut value is ,500

4.2.3 Variables selected for further analysis

This MLR analysis has lead to conclude that the variables in Table 4.9 are all significant at the 5% level and are selected for further analysis: "Depth", "BitMDepth", "AvgRPM", "AvgHookld", "avgWOB", "MaxTorq", "MudDensOut(removed)", "AvgROP", "AvgPumpP", "AvgTorq", "ReturDeapth", "DeltaFlow".

As Table 4.5 show, MudDensOut (mud density out) correlates perfectly with MudDensIn (mud density in) which means they are similar at all recordings. This means the gauge recording this value does not discriminate between these values. In addition, domain experts[10] marked this parameter as non- reliable at the time of recording most of these wells. This variable survive the MLR analysis and was decided dropped from the selection.

4.3 Discussion and conclusions

Multiple logistic regression is well suited to handle data with dicotome (binary) dependent variables, and is therefore also a useful tool to reveal significant relations between relevant independent variables and unwanted events. As the present analyses showed, using a broadly accepted statistical performance measure, 29 variables taper off to 13 significantly related to the selected unwanted event. However, the assumption that the training examples or points of measurement are independent is not fulfilled. Moreover, autocorrelation, i.e. correlation between the residuals, is a problem (as always in longitudinally data).

Nevertheless, the results from the logistic regression analyses serve as an adequate guidance for selecting variables in an alternative approach to predict stuck pipe. which was the ultimate objective of the presented analyses.

The next chapter make use of these results as developing and testing the extended Backpropagation Neural Net heavily depends on reducing the input vector without significant information loss.

[10] Domain experts during data acquisition were a team consisting of a drilling supervisor and five drilling engineers.

CHAPTER 5

ARTIFICIAL NEURAL NETWORKS

In this chapter, two ANN (Artificial Neural Networks) methods are investigated to find an optimal method, or set of methods, to process data arriving at the "Prediction module" of Figure 2.3.
At this stage, design criterion, i.e. the possibilities, requirements and constraints of the TrollCreek system were set as concluded in Chapter 3: The design criteria given in Chapter 2, second iteration (Section 2.3.2) for the "ANN"- sub- module remain.

Based on previous work, one method has proven potential, the Standard Backpropagation Neural Net, and one method showing sufficient tendency to explore [Valaas, I, 2004]

ANNs are a collection of methods for classification, pattern recognition, prediction and function approximation (in addition to several other utilizations and specializations within these areas). The purpose of this chapter is justifying why this AI- tool was selected as an alarm signal processing method by showing how ANN's excel over other signal processing methods, given the requirements presented in Chapter 2.3.. and showing the basics of an ANN.

5.1 ANN Applications and benefits compared to conventional methods

Besides having the advantage of being an eager learner: once trained the computationally expensive part is done, several other advantages are listed:
Partridge, D., Abidi, S. S. R., and Goh, A. (1996) listed several benefits of ANN's over conventional computation and manual analysis:

1. Implementation using data instead of possibly ill defined rules.
2. Noise and novel situations are handled automatically via data generalization.
3. Predictability of future indicator values based on past data and trend recognition.
4. Automated real-time analysis and diagnosis.
5. Enables rapid identification and classification of input data.
6. Eliminates error associated with human fatigue and habituation.

Passold, F., Ojeda, R. G., and Mur, J. (1996) summarized the benefits of neural networks as follows:

1. Ability to process a massive of input data
2. Simulation of diffuse domain reasoning
3. Higher performances when compared to statistical approaches
4. Self-organizing ability-learning capability
5. Easy knowledge base updating

Using Partridge et. al's list of benefits both 1 & 2 applies to the MWD data; no set of rules or transparent methods might cover all recorded variables and their possible interactions. Furthermore, this tool need to handle error prone data and a poorly classified training database as described in the functional requirements, sub- chapter 2.3.1. pt 2 & 6. Point 3, 4 and 5 could be functional requirements for the very goal for this part of the analysis namely Goal 1: "Real- time, computationally efficient prediction" and meet the functional requirement from the data description 2.3.2 pt 1. Complementing Pertrige et. al.'s summarization with Passold et al.'s summarization, ANN's are capable of processing massive datasets, which accommodate the requirement, chapter 2.3.2 pt. 1 and partly 4. Pure statistical approaches have been tested and failed as a predictive means on these data (I. Valås, 1998) but point 3 together with the above mentioned merits might indicate ANN's surpassing classical methods in a predictive perspective

5.2 ANN fundamentals.

This is a brief introduction to the functionality of a MLP (Multilayer Perceptron) ANN having a single hidden layer and serves as an example readily generalized to more complex net topologies and as an overview rather than a complete theoretical basis. The detailed foundations and basic learning algorithms of an ANN are thoroughly covered in e.g. [Mitchell 1997 pp.81-111], [Luger 2002 pp. 419-434], [Rumelhart, Hinton, Williams 1986. pp. 533-536].
Furthermore, for the purpose of this thesis, only the supervised methods SBP (Standard Back- propagation) and RBF (Radial Basis Function) are discussed.

Figure 6-1 shows a typical single hidden layer Feed – Forward net in a training phase having n input nodes, m nodes in the hidden layer and k output nodes. Furthermore, this net is fed with an input vector **X** (or a "training instance") of n features and attribute values [X_1, X_i, …, X_n] at the input layer producing an output vector **Z** of k attributes, [Z_1, ..,Z_k]. Finally **Z** is compared to the desired output, the target vector **t =** [t_1,.., t_k].

Weight connections between input layer unit i and hidden layer unit j are denoted by v_{ji}, $i=1, 2, ..., n, j=1, 2 ..., h,$ while weight connections between hidden layer unit j and output k are designed as w_{kj}, $k=1, 2, ..., m$

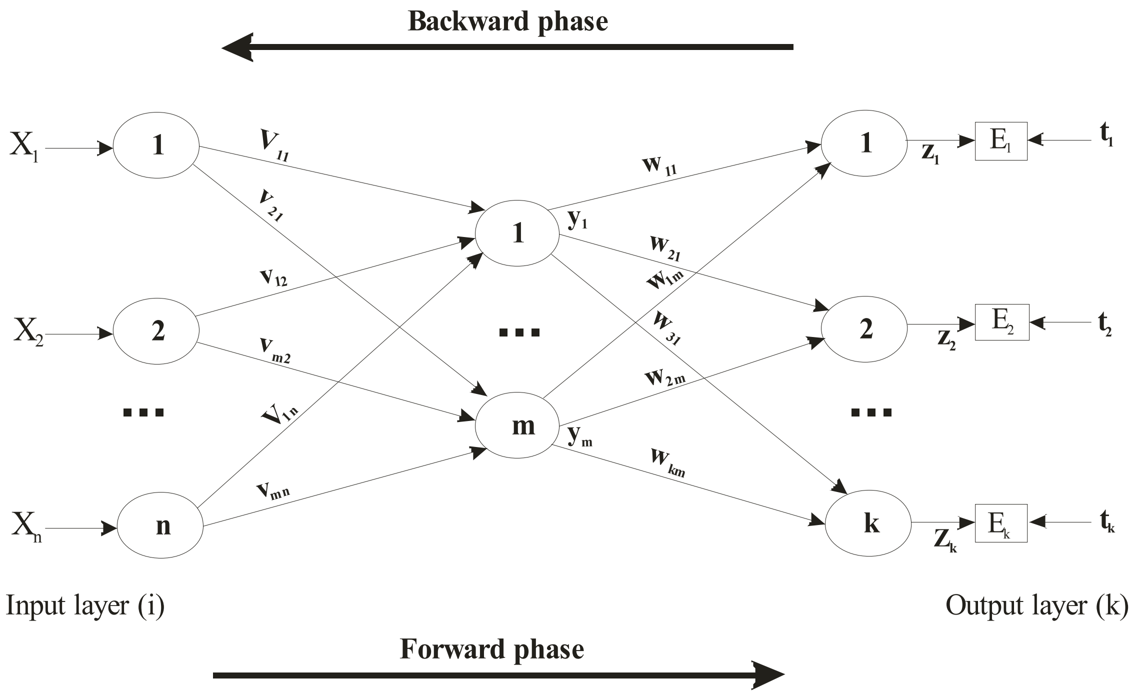


Figure 5.1: Feed Forward ANN, single Hidden Layer

t_i denotes the true target values, z_i the calculated output and E_i represents the error term $E_k = t_k - z_k$

Depending on the performance criterion; a means of quantifying deviations from the target vector **t**, the produced output vector **Z** and their deviations **E**, the output produced is either acceptable or a correction strategy updating the weights is employed.

5.2.1 Measuring performance

The performance of the net is evaluated using its precision in target estimation; a performance measure.

Root Mean Square Error

One of the most basic performance measures is the Root Mean Square Error (RMSE).

Equation 5.1: Root Mean Square Error over a sample of size n

$$RMSE = \frac{\sqrt{\sum_k {E_k}^2}}{n}$$

and a performance criterion c; RMSE $\leq$ c.

Separate validation set

The best way of measuring performance is testing on unseen data using a separate test set:

Holdout: Split the training data in two, train on one half and test on the other.

If data are scarce however, there are possible training schemes measuring performance using the same data on which the ANN is trained.

Cross validation

A portion of the training data is reserved for testing on which the trained model can be tested. There are essentially three cross validation variations:

1. N-fold cross validation: Split the training data in n folds. Of these n folds, n-1 are used for training and the remaining fold for testing. This is repeated until all n sets are tested upon and the results are averaged.
2. Leave-one-out is N-fold cross validation using n=N-1.
3. Stratified n- fold cross validation: Stratifying the data means assuring an equal portion of each classification is distributed on each fold in the cross validation. Performance estimate variance is particularly sensitive in random fluctuations of instance classifications. This method forces a uniform classification distribution on training and validation data.

Regardless of data and network configuration, the training method reusing training data of the above discussed methods yielding best results appears to be stratified 10- fold cross validation [Kohavi, R. 1995].

5.3 Backpropagation ANN

The most popular ANN method for pattern recognition and classification is the BP (Backpropagation) or SBP (Standard Backpropagation) algorithms as introduced by [Werbos, P, 1982].

5.3.1 Forward phase

In the forward phase of a SBP ANN the input of any PE (Processing Elements) is the cumulated weighted input from all PE's in the previous layer, e.g. PE k of the output layer in fig. 6-1 receives input I where

Equation 5.2: Cumulated weighted input to process element k

$$I_k = \sum w_{kj} y_j,\ k=1, 2, ...,m$$

Outputs computed by unit k of the output layer is given by

Equation 5.3: Output of any PE not belonging to the input layer

$$z_k = f(I_k),\ k = 1, 2, ..., m$$

f in Equation 12 is the selected layer specific activation function.

Referring to Figure 6-1, the output unit k the following response to an input pattern x is

Equation 5.4: Output of a PE not belonging to the input or first hidden layer

$$z_k = f(I_k) = f\left(\sum_j w_{kj} y_j\right) = f\left(\sum_j w_{kj} f(H_j)\right) = f\left(\sum_j w_{kj} f(\sum_i v_{ji} x_i)\right)$$

5.3.2 Backward phase

Depending on the performance of this net as discussed in the general theory, error propagation backwards is initiated.

Weight update is performed according to gradient descent learning:

Equation 5.5: Gradient descent weight update for the output layer

$$\text{i)}\quad \Delta w_{kj} = \eta \frac{\partial E}{\partial w_{kj}} = \eta \delta_k y_j$$

$$\text{ii)}\quad \delta_k = (t_k - z_k)\ f'(I_k)$$

η in the equation 15i denotes the learning rate, or the preferably infinitesimally small increment by which the weights change per backpropagation iteration.

Weight update for the output vector **k** resulting from a performance measure below a chosen criterion:

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k)\, f'(I_k) y_j$$

Consequently, weight update for the last hidden layer:

$$\Delta v_{ji} = \eta \delta_j x_i = \eta x_i \; f'(H_j) \sum_k \delta_k w_{kj}$$

Figure 6 shows how the backpropagation algorithm successively updates weights backwards according to the error term E_k at output node k:

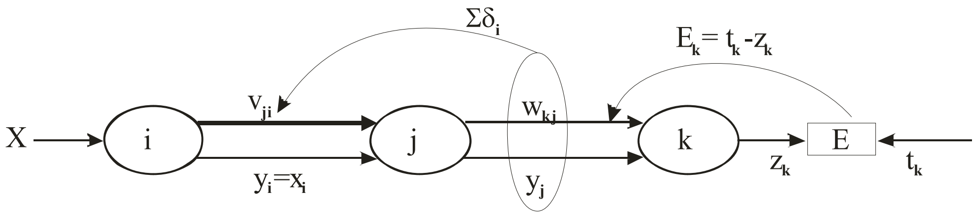


Figure 5.2: Error propagating backwards

5.3.3 Tentative experiments

As the explorative analysis produced promising results, all experiments, results and discussion is presented in detail in chapter 4.5.

5.4 Radial Basis functions ANN

Radial basis function (RBF) networks (hereby referred to as RBFN), first utilized by Broomhead and Lowe [Broomhead, D.S., Lowe, D. 1988] constitutes a special class of ANNs suited for function interpolation and capable of multidimential space interpolation. RBFNs is the core of other machine learning techniques such as Support Vector Machines [Cortes, C., Vapnik, V, 1995] and in specialized regularization networks [Poggio, T., Girosi, F 1990].

As for any ANN algorithms, they are iterative and irreproducible[11], but RBFNs has two advantages compared to the SBP networks with respect to:

1. Modeling and empirical effort: Network topology is significantly simpler by being (nearly) exclusively represented by a single hidden layer but more importantly by the locality property of these nets allowing the hidden layer to be incrementally

[11] There are examples of knowledge extraction from RBFNs, see e.g. [McGarry, K., J., Wermter, S., MacIntyre, J, 1999]

expanded or pruned. As each PE represent a local region of the dataset by it's centroid, RBFNs allows the network layout to be incrementally constructed [Millian, J.R., 1994]

2. Computational efficiency: two separable components which can be treated individually and occasion training optimization.

Furthermore, the RBFs have a significant mathematical foundation imbedded, which was believed to be advantageous in analyzing an analytically pre- processed dataset originally intended as input to a transparent, predictive stuck pipe function

.

The following subchapter gives a high level description of the functionality and basic training algorithms of RBFNs, sufficiently elaborated to understand the underlying functionality of the commercial software used in the explorative RBNF experiments:[NeuFrame].

5.4.1 RBF fundamentals

In the next section the principles behind function interpolation is described, omitting the mathematical foundation in mapping the interpolation matrix to an appurtenant weight matrix producing a possible exact interpolation

Interpolation of functions

A set of suitable radially symmetric function, e.g. the Gaussian radial function, shown in Equation dummy and figure dummy

:

Equation 5.6: Gaussian RBF

$$G_\sigma(x;\mu) = \exp(-\frac{1}{2\sigma^2} \| x - \mu \|^2$$

are superposed

Equation 5.7: Gaussian function generation

$$F(X) = \sum_{i=1}^{3} w_i \cdot e^{\frac{1}{2\sigma^2}(\chi - \mu_i)^2}$$

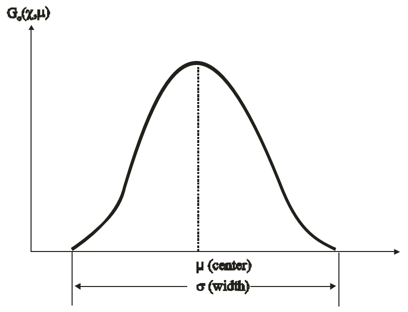


Figure 5.3: 2- dimensional Gaussian RBF

to generate the interpolated function F(x) shown below :

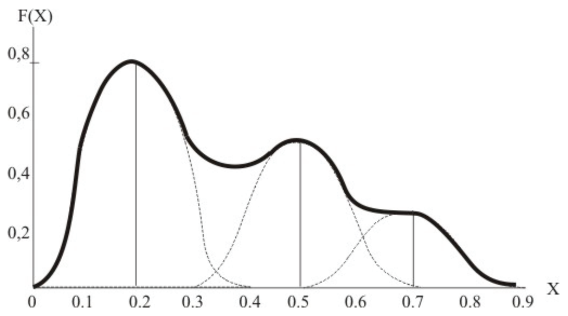


Figure 5.4: Function generation by radial basis function superposition

The 3 RBFs representing F(x) in Figure 4 6 have a uniform σ-value (width) distribution and centres are , and (corresponding to the x- values). Once having found correct RBFs, established centres and widths, the remaining unknown in the function modelled is the set of weights corresponding to wi in Equation 5.8

Terminology

In this chapter the RBF parameter names width (σ) and centre (μ) are used instead of the frequently used terms mean and variance. The former version was chosen since these terms are more descriptive and separates them from their statistical interpretations. Furthermore, using the term variance might indicate a RBF width uniform distribution, which often is the case, but not always.

RBFN - approximation by minimizing error

The task of the RBFN is modelling a physical process using a limited set of N samples represented by M RBFs in a network structure. The first simplification is explained in the general ANN theory. The second simplification is equally necessary when modelling physical processes, which necessitates an M << N and contributes to the overall true error.

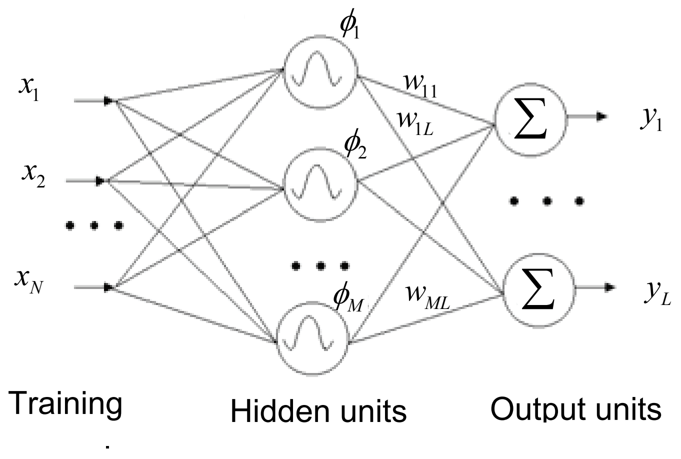


Figure 5.5: A typical RBF network structure

The figure above describes a RBFN structure by which nearly all RBFNs is described and is superficially described in the general ANN theory, however with some important differences:

The linear component to which the output layer belongs can be trained separately searching for optimal weights using for instance the SBP algorithm giving the weight matrix {w}. The hidden layer consist of M RBFs, θ_i with their radially symmetrical functions ϕ_i, their centres, μ_i and their widths, σ_i.

Hence a single layer RBFN is completely described by the set $\omega = [\theta_1, \theta_2, .., \theta_M]$ and the weight matrix {w}.

5.4.2 Training Algorithms

In the example of deriving Figure 4-5 all instances of the training data were used as centers of the RBFs and all widths of the basis function were equal This technique only serve as a conceptual foundation considering the excessively large networks being produced and high susceptibility to overfitting. There is a need for strategies in optimizing the hidden layer topology and searching for optimal radial basis function centers. Below are some of the possible training and optimization strategies discussed:

Finding the Number of Hidden Units

There are mainly 3 strategies in modelling the hidden layer:

1. Constructive – starting with an arbitrary small number of PEs, new units are added if the selected deviation measure does not decrease.

2. Pruning – starting with a large set of PEs, hidden units which do not contribute significantly to the measured performance are disregarded
3. Mixed increasing-decreasing topology

There is always a danger of overfitting, i.e. having added too many PEs in the hidden layer thus reducing ability to generalize, and underfitting, either by stopping the constructive training too soon or having pruned too many PEs. Several algorithms have been developed to optimizing the trade- off between over- and underfitting and will be elaborated if justified by the explorative analysis.

Selecting centers randomly from data

Presupposing there is little clustering tendency in the input data, a straightforward technique is to chooses an arbitrary number M of PEs in the hidden layer and randomly choose M instances to serve as centres for the RBFs.

A possible training algorithm is to use Increasing Topology until acceptable performance is reached:

```
M = initial_noof_PEs;
While (performance < criterion){
        centers = randomSelectCenters(M);
        net = constructRBFM(centers);
        performance = net.measurePerformance();
        M = M + 1;
}
```

Centre clustering algorithms

If the data exhibit clustering tendencies, one solution is grouping those instances and assigning ether real or constructed prototypical cluster representatives. Identifying these centroids is done unsupervised using a clustering algorithm of which K-Means is an example.

A usual training algorithm in this scheme is to use Increasing Topology until acceptable performance is reached:

```
M = 2;
While (performance < criterion){
        centers = ClusteringAlgorithmFindCenters(M);
        net = constructRBFM(centers);
        performance = net.measurePerformance();
        M = M + 1;
}
```

Gradient descent centre search

Any training by gradient descent can be expressed by the learning rule

Equation 5.9: Gradient descent RBF learning rule

$$\Delta\kappa_i = \eta \frac{\partial E}{\partial \kappa_i}$$

where κ_i is any component of ω or $\{w\}$, η is the learning rate and E is the error estimate, for instance RMSE. A possible algorithm for κ_i - optimization is SBP.

5.4.3 Experiments

In all experiments, widths are estimated from data, i.e. no global width, and cross validation is used as stop criterion. Furthermore, the training data is comprised by wells 1-4 while wells 5-7 are reserved for validation.

Computational limitations and consequences

As expected, using 10- hour data were not computationally feasible unless settling for a fixed, extremely slim hidden layer topology. The number of instances per well were reduced from 7000 to 2500 meaning a significantly less representative selection than the original.

Both the quantitative and qualitative level of empirical testing suffered from the poor processing performance; quantitatively as each training session consumed a disproportional amount of time or lead to an unstable net as topology grew. This meant severely reducing the scope of possible trials. Qualitatively the testing suffered primarily as a result of the needed data reduction and secondarily as a result of the limited topology possibly explored.

K-means clustering algorithm

It was decided to run a training algorithm having embedded self- constructing topology initially to find how far the RBFN would develop until cross- validated performance decayed. Starting with 2 clusters, the k-means algorithm stopped after 3 iterations having increased the hidden layer to 5 PEs until CV (cross validation) performance decayed.

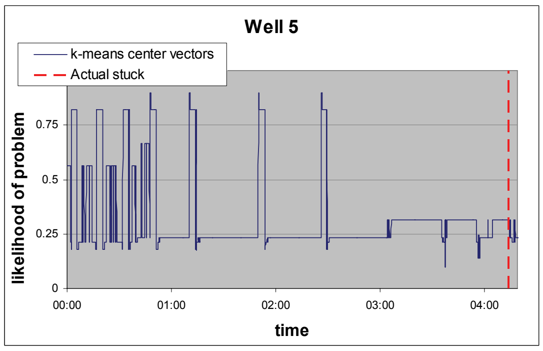


Figure 5.6: K-means centre selection - Well 5

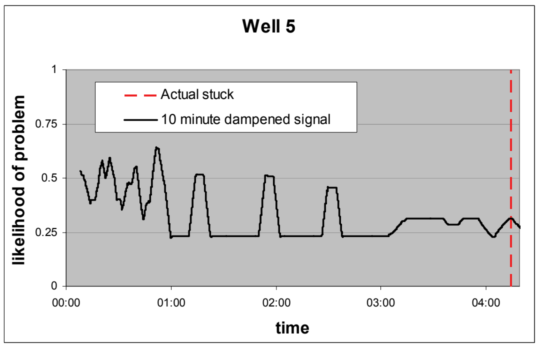


Figure 5.7: moving average - 10 minutes delay

Figures 4-6 and 4-7 show the result of training using the K-means algorithm, instant readings and moving average respectively. The instant reading curve has a characteristic fluctuating tendency, resulting from a near binary signal. This make the signal difficult to interpret as there is no possibility for e.g. an incrementally moving threshold responding to erroneous alarms. Moreover, the overall trend point toward an increasingly safer situation, which clearly is erroneous.

The moving average curve has a noise dampening effect as it averages the 120 last readings thus smoothen the amplitudes of the curve. The cost however, is that the 10 minute averaging period also becomes a 10 minute signal delay and furthermore might suppress genuine warnings. Unfortunately in this case, the trend curve only served as a clear illustration of how severely the model failed in modelling the physical process for validation well 5.

Figure 4-8 show an even more severe fluctuating tendency than the previous validation well, both with respect to frequency and amplitudes. There are, however, a more consistent trend towards stuck pipe 1 hour prior to the event occurs meaning the interpretation of the drilling situation is an improvement compared to the former experiment. Nevertheless, a signal interpretation from real- time surveillance of this well would trigger too many erroneous alarms to be useable.

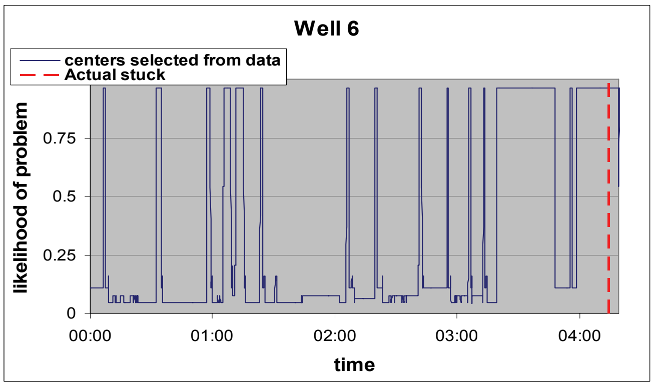


Figure 5.8: RBFN K-means Well 6

The last validation well shown in Figure 4-9 show improvements over the two former validation wells in two important aspects:

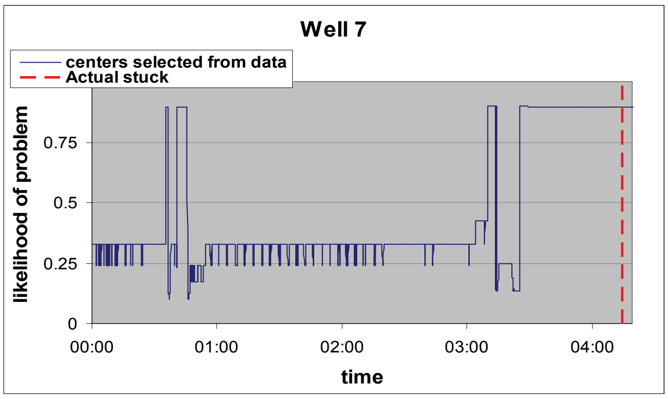


Figure 5.9: RBFN K-means Well 7

1. Less fluctuation tendencies, both with respect to frequency and to amplitudes.
2. Fewer erroneous alarms

Although there are promising facets to this test, there are several uncertainties to the model:

- The predicted unwanted event likelihood still has a binary characteristic, i.e. there still is no trend- line or a build- up curve rendering a possible adjustable alarm threshold.
- The alarm state is constant during approximately one hour prior to and including the unwanted event
- There are still 4 equally severe peaks during this short interval

Gradient descent search

Forward search by gradient descent resulted in a non- responsive net and could not be tested using this large training dataset.

Random selection

The k- means algorithm's CV performance measure decayed when introducing the sixth PE. This serves as an indication from where to start exploring possible topologies by random selection.

Figure 4-10 show the most promising result during the RBFN experiments. It is an extraordinarily clean signal and no anomaly exceed the first maximum amplitude 1:04 (hh:mm) prior to stuck pipe.

This was achieved choosing 40 PEs, which represented the computational borderline despite using the computationally least expensive random selection algorithm.

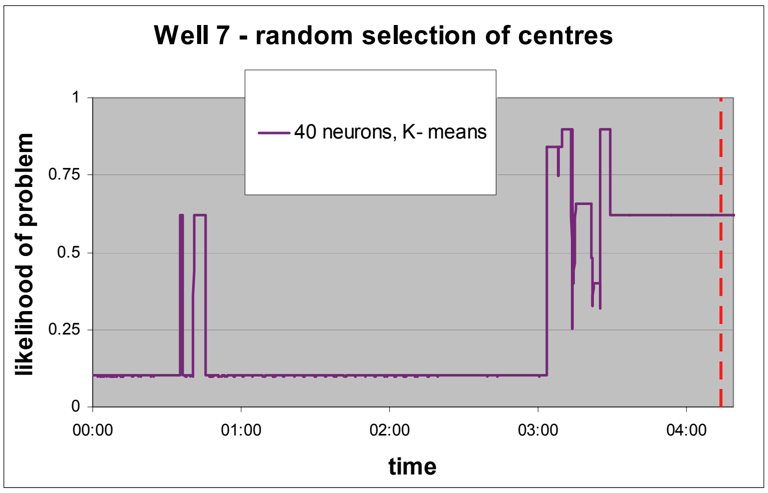


Figure 5.10: Well 7 - random centre selection, 40 PEs

The effects of incrementally expanding the hidden layer are shown in Figure 4-11 and further illustrate how performance for validation well 7 improves by increasing the hidden layer topology:

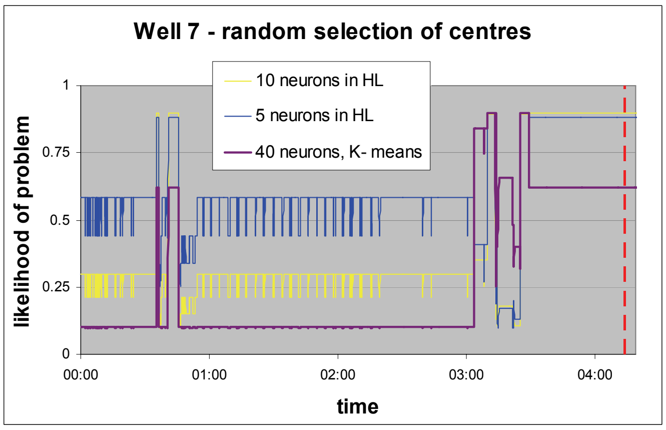


Figure 5.11: Well 7 - effects of varying topology

This might indicate that the performance plateau of expanding the hidden layer is not reached and further expansion would lead to a better, overall performance. However, Figure 4-12 shows how the main trends remain regardless of topology. Well 5 gave poor predictions using the k-means 5 cluster centres as it does using what appeared to be a fine tuned RBFN.

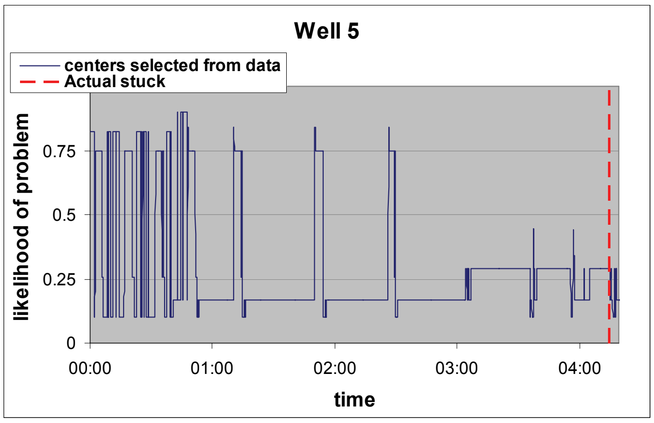


Figure 5.12: Well 5 - random centre selection, 40 PEs

This was further confirmed by validation Well 6 shown in Figure 4-13:

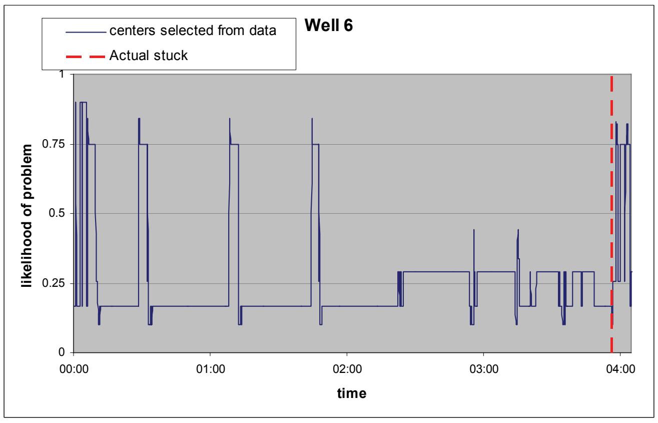


Figure 5.13: Well 6 - random centre selection, 40 PEs

The RBFN modeling the underlying function resulted in an interpretation of the drilling situation close to perfectly fitted to the training data and could be interpreted as proof of severe overfitting, if interpreted alone.

5.4.4 Discussion and conclusions

Of the 3 validation wells tested, only validation well 7 gives a somewhat adequate signal, thus leaving the feasibility of RBFN applied to analytically derived data as not recommendable. The reasons seem to be both within the RBFN learner exposed to these data and the data itself:

Learner problems caused by the data

Although the RBFN possesses several computational advantages over SBP ANNs, it is sensitive to the amount of training examples, especially if their characteristics need a large set of representatives (centers) to be adequately modeled. The advantage of being near insensitive to the amount of input variables was poorly utilized as the prior analytical treatment of the MWD data already severely reduced the instance features. The dependant variable or the instance classification is categorical or binary which is unfortunate in using RBFN. Moreover, the nature of unwanted events while drilling, their respective frequencies and consequently; the large set of skewed distributed training instances needed, seem to be an insurmountable obstacle. Massive training data exhibiting scattered characteristics seem difficult to model using centre representative quantities within a reasonable computational expense and without overfitting the training data.

Indications of poor data

Algorithms auto- generating the hidden layer topology ceases at an early stage due to an increasing CV error, although tests with increasing hidden layer topology improved performance. This might indicate infeasibility of modeling unwanted events based on these data, regardless of learner type.
As a consequence of the above points, both RBFNs and the analytical derived dataset were left at the explorative stage.

To summarize:
One result of the analysis performed in this chapter is that the only remaining analytical method processing the "Analytical Method" sub- module is the statistical approach presented in Chapter 4. The second important result is that the remaining method to test of the outlined AI- methods in the Prediction- module of Figure 2.3 is s Standard Backpropagation Neural Network., which also is the method of proven potential. The development will be presented and developed in the next chapter.

CHAPTER 6

DEVELOPMENT OF THE PREDICTION MODULE

This chapter is a continuation of the previous chapter, focusing exclusively on the Standard Backpropagation Neural Networks and using the statistical results from Chapter 4 to pre- select MWD- attributes for analysis.

6.1 Related research

There is a remarkable amount of published academic and commercial work on general SBP-ANN (Standard Backpropagation ANN). Initially the need was a simple SBP algorithm, of which there are numerous, undocumented downloadable examples at the Internet. An attempt was made at reusing code earlier developed for TrollCreek, meanwhile, this code could not be found. Thus, the simple SBP algorithm was implemented, inspired by numerous, undocumented implementations, supported by the theory referenced in Chapter 5.

The iterative nature of this development occasioned numerous addendums, documented successively.

6.2 General implemental remarks

Signal processing was implemented using a basic SBP ANN as a starting point, adding several extensions and data tool support classes as necessary to meet the functional requirements.

All classes have been implemented using Java SDK (Standard Development Kit) 1.4 prioritizing fast code above correct object oriented programming. Classes having low cohesion and high coupling suffer from a trade- off between execution efficiency in keeping message exchange at a low level and slim method signatures on one hand, and correct implementation within the object oriented paradigm on the other. Furthermore, object instantiating is kept to a minimum, and primitive data types are used almost exclusively to further enhance execution efficiency. The last implemental issconcern has been keeping a flat class- hierarchy, to ease code reuse and possible TrollCreek integration.

The code is available for download and review at:

http://www.idi.ntnu.no/~ingeasmu/thesis2005_valaas/index.html
Appendix 9.4 shows the prototype GUI and gives a graphic overview over the functionality within the implemented ANN.
Much work has been put into making easily readable code and thorough code documentation for possible reuse. Detailed class and method documentation has been implemented according to the JavaDoc html- standard and is found in Appendix 9.5.

The next section gives a brief overview of the main and selected supporting classes. Unless stated explicitly, all presented techniques are implemented and integrated into the code. Fully implemented including all additions the developed net is referred to as "extended SBP- ANN" (e-SBP ANN).

6.3 SBP ANN framework

The classes ArtificialNeuralNet.java, Node.java, Link.java, Layer.java, Instance.java, and InstanceSet.java constitute the framework for which SBP ANN algorithms is needed. Any standard functionality is embedded, such as ability to represent an arbitrary net topology, both with respect to the number of features and classifications, number of layers and their PE's (processing elements). In addition, the application offers a variety of I/O operations, such as loading instances, net topologies and weights from file, the two latter also with a save- to- file option. These are included in the methods maintained by the class FileOps.java.

A consequence of the peculiarities of the MWD data, reflected by the functional requirements of section 2.3.2, several further additions and extensions were necessary.

6.4 Additions to the SBP- algorithm

The following describes extensions to the *internal* data processing of the SBP ANN:

6.4.1 Randomized instance selection

Considering the net continuously being trained on negative[12] examples prior to presenting a small fraction of continuously positively classified examples and furthermore; considering the functional requirements' of Section 2.2, point 4, an option of randomizing the order by which training instances enter the input layer was

[12] Negative in this context means stuck pipe = false

implemented. Various methods for randomizing lists without replacement are implemented in the class MathLib.java.

6.4.2 Weight update schemes

There are two distinct training schemes by which the weights in SBP training are updated; batch training (in which the weights are updated after processing the entire training set), and incremental training (in which the weights are updated after processing each training example) [Bertsekas, D. P. and Tsitsiklis, J. N. 1996]. Batch training is usually considered faster as the number of weight updates is independent of the number of training examples. However, there are recent studies showing contradictory results in gradient descent training, especially on large datasets [Randall, D., Martinez, T., R. 2003]. The increased number of iterations needed using batch training does not necessarily balance the efficiency gain of less weight updates.

Considering the functional requirements of Chapter 2.2, both strategies were implemented, the latter having an important extension:

1. Sequential training – weights are updated after each training instance
2. Selective batch training – a selected fraction m of the n training examples is presented to the net prior to a weight update giving $\left(\frac{m}{n}\right)$ updates per iteration.

Strategy 2 becomes a compromise being a "$\left(\frac{m}{n}\right)$'th batch training technique". (well documented in ArtificialNeuralNet.java or it's pertinent JavaDoc)

6.4.3 Layer specific activation functions, momenta and learning rates

Besides the choice of net architecture, there are two static parameters and a polynomial function by which training is directly influenced: Momentum, learning rate and choice of activation function. The Layer class of the ANN maintains their pertinent PE's momentum and learning rate values hence allowing fine tuning of these training parameters, if necessary and valuable. In addition, each layer's PE activation functions can be of type linear, sigmoid or hyperbolic tangent and are among the methods implemented in the class MathLib.java.

Figure 7 shows possible tuning of each layer representing their pertaining PE's.

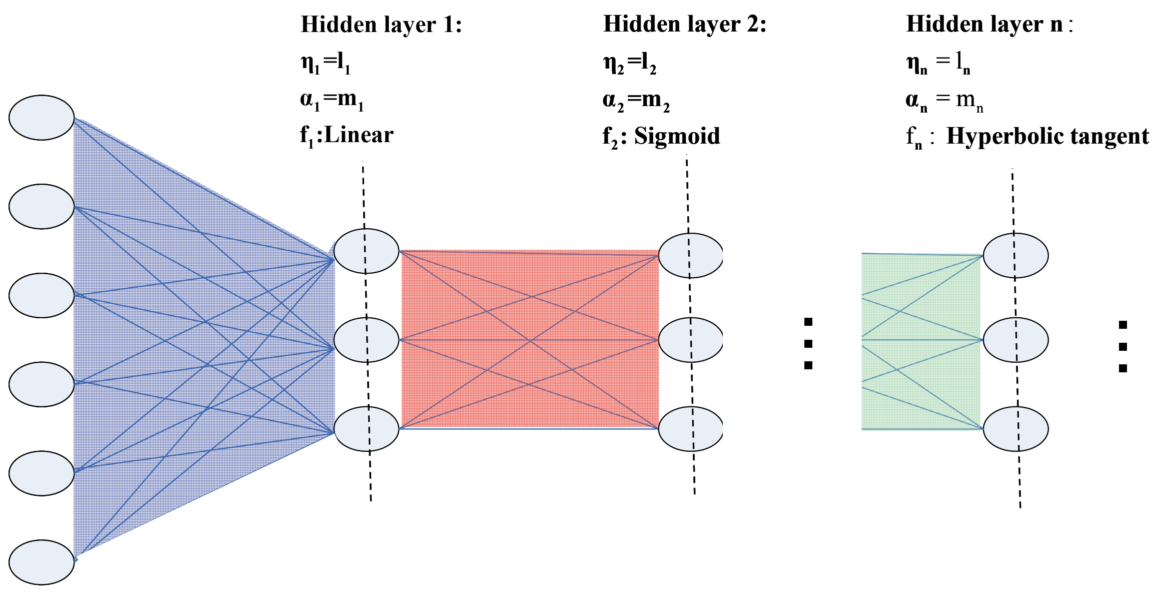


Figure 6.1: Tuning options of the enhanced SBP ANN

Signals following inks on blue background are processed using a linear activation function, a learning rate $\eta = l_1$ and $\alpha = m_1$, different from layers 2 and n having unique η and α values and sigmoidal and hyperbolic tangential activation function respectively.

6.5 Data pre and post- processing

The following sub- section discusses the *external* data processing.

The implemented ANN is capable of receiving any input data regardless of data range and has no formal requirement for normalized data or any functional limitation with respect to data range. However, referring to the functional requirements of section 2.2.2, particularly point 6 an option of normalizing the training instances' features (here; the MWD variable values) seem necessary. Methods for pre- and post- processing of features are implemented in the class DataTools.java.

6.5.1 Normalization

Normalization of data in this context means giving all feature values the same relative value, and must not be confused with database normalization. The purpose is assigning all features values I within the range $I \in [I_{min}, I_{max}]$.

There are 3 common types of normalization:

1. Linear transformation: An input range $D \in [D_{min}, D_{max}]$ results in the simple mapping between each feature value D to a normalized value I

Equation 6.1: Linear normalization

$$I = \frac{I_{min} + (I_{max} - I_{min}) * (D - D_{min})}{D_{max} - D_{min}}$$

2. Utilizing a statistical measure of central tendency and variance to help remove extreme, non representative values and spread out the distribution of the data.. This is a relatively simple method of normalization, in which the mean and standard deviation for the input data associated with each input are determined. D_{min} is then set to the mean μ minus a selected number of standard deviations c $D_{min} = \mu - c$ and conversely , $D_{max} = \mu + c$
3. Minimizing standard deviation of the heights of the columns in the initial frequency distribution histogram.

Method 1 results in a new data range solely depending on the dataset's minima and maxima, which presuppose errorless data. Method 2 and 3 smoothen the effect of erroneous extremal values, however considering the multitude by which errors occur in the MWD data (Chapter 2.2.1, sub- section "noisy data"), mean values and standard deviations are still susceptible to extreme noise. Another approach is to identify and replace erroneous data by acceptable extremal values, introduced in the next section:

Outliers – a technique for noise identification and removal

An option is to identify a the most deviant feature values for a chosen feature and change these values so that they are deviant, but not as deviant as they were [Tabachnick & Fidell, 1989]. One simple technique for identifying the outlying values is removing the n% most extreme min/max values, using the remaining extremal values as V_{min}, V_{max} and replacing the removed n% with the new extremal values [Howell, D. C., 1995]:

Equation 6.2: Outliers replacements

i) $\frac{1}{2}(n\%)_{high} = V_{max}$

ii) $\frac{1}{2}(n\%)_{low} = V_{min}$

Data normalization, if chosen, then becomes first removing outliers, then using the linear transformation as described above (6.5.1, point 1).

6.5.2 Output signal smoothening

A prototype of these technique was tested using Microsoft Excel, and pending further evaluation of the smoothening necessity and contribution, is left at the prototype stage. (See for instance [Moving Averages 1], [Moving Averages 2])

Smoothening of the output value in terms of SMA, *simple moving average* means buffering up and averaging a portion of the outputs prior to presenting the signal. This means a lag (here; slow response to changes) of the specified average period and consequently a signal delay correspondingly. The lag might be reduced using the EMA, *exponential moving average* weighting instances inverse exponentially to its distance in the buffer. A simple spreadsheet prototype was tested showing the effect of both in the figure below:

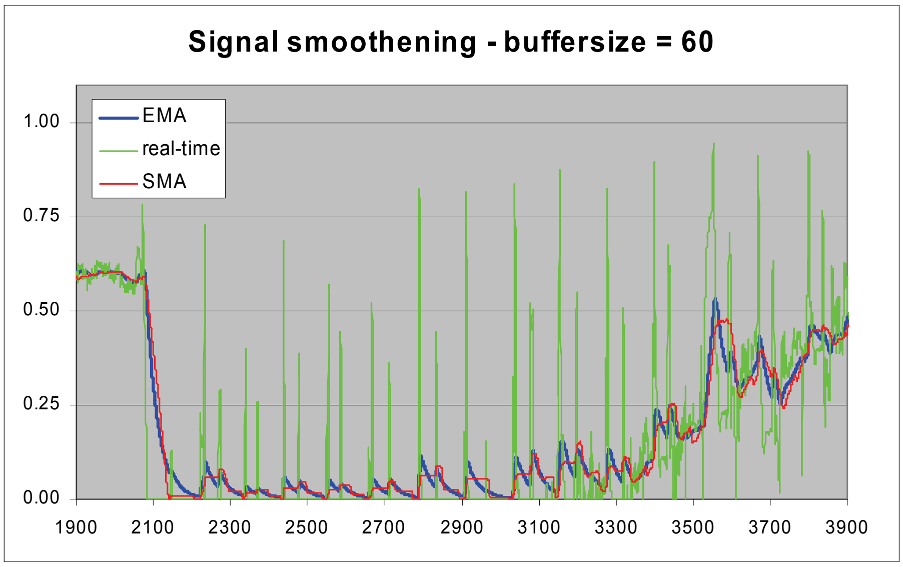


Figure 6.2: Signal smoothening by moving average

The red curve shows the exponential moving average, reacting more rapidly to recent changes than the simple moving average in blue.

Random peaks representing noise rather than real anomalies are averaged away by his method and might reduce noisy fluctuations, but delay the signal proportionally to the smoothening factor and might hide real and valuable amplitudes giving actual warnings.

6.6 Modeling and training the ANN

There are several non- trivial issues to be addressed in determining a suitable trained SBP ANN for classification of new instances. Two such major issues are avoiding overfitting and the topology design of the network itself.

6.6.1 Measuring performance

The optimal performance measurement discussed in 5.2.1 is an attractive idea, but 10-fold stratified cross-validation imply an a- priori distribution of 700 positive samples per 49000 negative samples in 10 folds and seem unfeasible. Unwanted events in this domain are not normally distributed and are results of several inter-correlated events. Consequently, variables should not be subject to any prior, even distribution.
A "randomized cross validation" training data set was prepared in the InstanceSet class, dividing the data set in n folds and picking members to these folds randomly without replacement using the MathLib class.

6.6.2 Ensuring ability to generalize by avoiding overfitting

Typically, an ANN begins training with a poor fit to the data due to its random weight initialization. As training progresses, the neural networks fit as the data improves. At some point, however, the neural network begins to overfit the data, meaning that its performance on the training data continues to improve, but only because it is beginning to memorize the peculiarities of the training examples, not because it is learning more about the underlying process.

Early stopping involves training with some fraction of the data, and testing periodically with a separate set of data. Performance on the training data called the "apparent error" will usually only trend downward. Performance as tested on the test data (an estimate of the "true error") will typically go down like the apparent error, but eventually diverge and, at some point, starts to become worse. The best place to stop training is indicated by the minimum estimated true error, that is, the best performance on the test data. [Freeman, J. A., Skapura, D. M., 1991].
There are several strategies in avoiding overfitting- one such strategy is using cross validation used to determine when the network has been trained as well as possible without overtraining thus achieving maximum generalization (Mitchell, T., 1997 pp. 108 - 112).

6.6.3 Deriving a suitable net topology and tuning parameters

While the choice of number of input and output nodes is defined by the problem, the number of hidden layers and of how many nodes each hidden layer should consist remains an empirical task. In general there is no way to determine a priori a good network topology. It depends critically on the number of training examples and the data complexity [Castiglione, F., 2001]. There are, however, a large number of methods being developed, most of which follow the evolution's paradigm (Genetic Algorithms or Evolutionary Strategies) which is beyond the scope of this work.

One simple strategy suggested by Castiglione is a possible "brute force" approach where all possible nets are explored between a manageable fixed set of combinations of n_{hl} (number of hidden layers), n_i (number of nodes in hidden layer i), e.g. $0< n_{hl} \leq 2$, $2<_{ni}\leq 15$. The number of possible topologies grows exponentially with these n_x boundaries, therefore only a limited set of topologies within each number of hidden layers has been explored and measured for performance. More precisely; promising nets are selected on the basis of the RMSE (root mean square error, implemented in the MathLib class), and these candidate nets are further explored on validation data comparing calculated vs. true values.

A positive bias was given to quickly learned topologies (i.e. with a quickly descending RMSE gradient) over nets reaching an adequately low RMSE from extremely large number of iterations.

To summarize; the final set of nodes of which the ANN is composed is a result of a *guided brute force empirical* approach.

Empirical testing in network topology optimization

It was not within the time frame of this thesis to thoroughly explore the many possible ANN configurations. Despite the computational limitations to possible topology configurations, the "topology search space" is considerable. The following search strategy was employed:

1. Start using a simple initial topology
2. Guide the topology search towards promising topology arrangements for instance evenly distributed PE's (Figure 4-15), most processing elements in the first layer(s) (Figure 4-16) or a preponderance of PE's in the hidden layers close to the output layer (Figure 4-17)

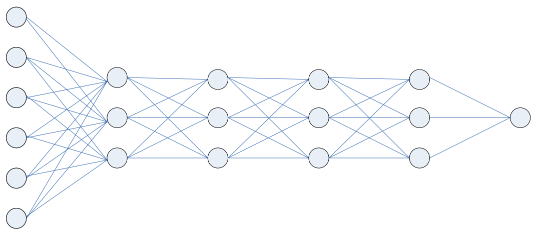


Figure 6.3: Symmetric topological arrangement

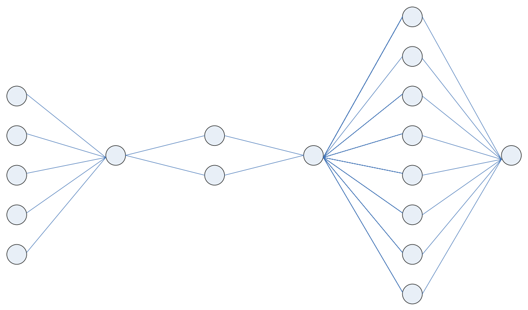


Figure 6.5: Right asymmetrical topology

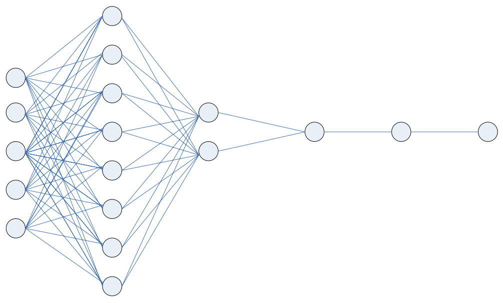


Figure 6.4: Left asymmetrical topology

3. Successively increase topological complexity primarily by adding PE's to existing layers until performance reaches a plateau or decays, secondarily by adding hidden layers.

Initial AN Net development strategy

Based on the referred theory it is persumed that topology is the predominant limitation both with respect to computational expense and performance. The training strategy then becomes:

1. Initialize the tuning parameters α and η uniformly over all layers making η (learning rate) small (0.01). This because the extra computing cost is relatively small compared to the reduced danger of missing the true global minimum error [Luger, G., F., 2002 pp. 429]. α (momentum) was chosen to be in the "sound" range $0{,}5 \leq \eta \leq 0{,}9$.
2. Design a net topology resulting in interpretable results
3. Fine tune the net varying the α and η parameters, uniformly over the ANN or stratified, giving layer specific learning rates and momenta

6.7 Testing the ANN

6.7.1 Test setup.

In all tests, 7 wells with stuck pipe incidents were utilized, each well having 10 hours of data, 9 hours 35 minutes prior to the recorded stuck pipe event and 25 minutes during/after. Translated into number of instances this becomes: 6900 instances prior to, and 100 instances during/after the recorded event.

The test setup had a fixed composition of training wells/test wells. with no a- priori knowledge of the qualitative contents of data, or bias towards any well being more suitable for training or validation, the composition was Well 1-4 as training pool and Well 5-7 as validation pool. As such, selection of training and validation wells were a result of "randoml selecteion". Lacking further, external validation wells, the fixed validation pool of 3 validation wells becomes the external test wells.

Experiments follow the training strategy of 6.6.3. Some trends became clear only having a few test-runs using the e-SBP ANN, leading to a revision of the initial training strategy:.

6.7.2 Revised training strategy

The following sections describe findings during initial testing and have impact on the rest of the e-SBP ANN analysis.

Selecting training algorithms

Regardless of topology or composition of training examples, only the randomized instance selection (described in section 6.4.1) and the modified batch- processing (as described in section 6.4.2) produced any meaningful, interpretable results. This training algorithm is hereby referred to as randomized N-fold batch-training. Standard methods lead to a net consistently predicting near or equal to zero. As well as being computationally favorable, N-fold batch- processing proved qualitatively necessary.

Estimating performance

In this context, performance measurements and validation did not fit in the usual concepts of errors and overfitting. The targeted feed forward ANN is one that actually deviates from training data, these being roughly classified (section 2.2.1) and should predicts trends towards an event rather than predicting the event exactly. Furthermore, estimating performance was expected to be difficult given the severely skewed distribution, but regardless of validation scheme, any net predicting all output near or equal to zero commits a small cumulative error. This seems sensible considering the classification distribution: Considering the test setup of 6.7.1: Each training example consists of 7000 instances of which 100 are positive, negative otherwise. If a net is trained to predict all classifications to be zero, the error frequency become f_{error}= 100/7000 = 0,015 , giving a flattering RMSE:

$$\text{RMSE} \leq \frac{\sqrt{100*(1-0)^2 + 6900*(0-0)^2}}{7000} = 0{,}0014\,.$$

Figure 4-19 shows an example based on [Valaas, I., 2004]. The appearant errors, RMSE decay as the number of training cycles increases.

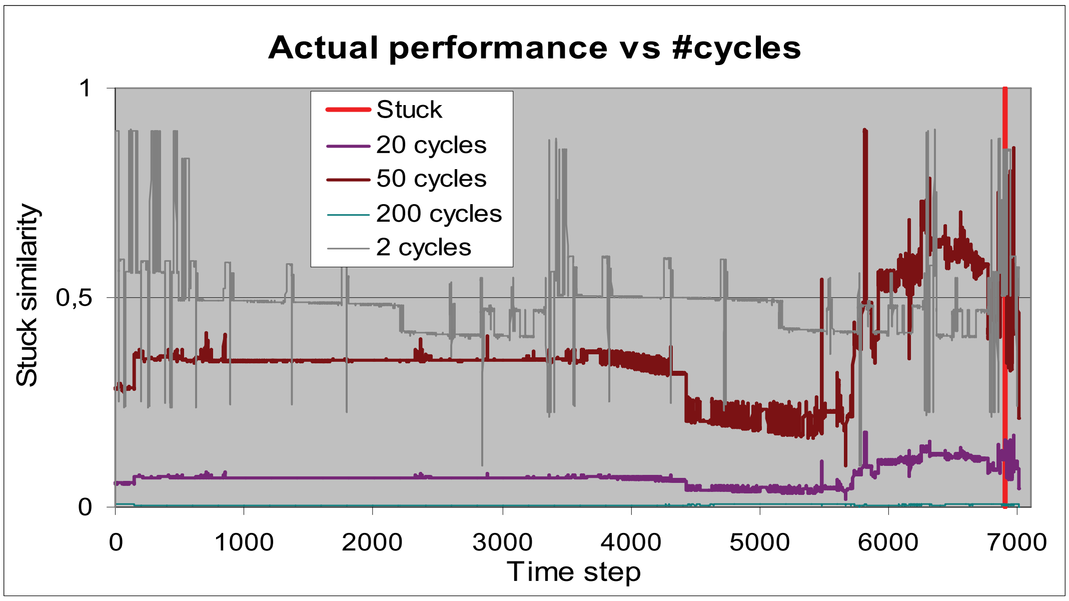


Figure 6.6: Actual performance vs. measured performance

This measured performance enhances as the cumulative sample distances deviates less from zero. while the actual performance, i.e. the prediction capabilities decay. The initial 2 cycles of backpropagating errors, shown as the most fluctuating (grey) curve, coincide with the poor performance measure. As the training process proceeds through the next 18 cycles, performance measure and actual predictive power enhances as shown in the second upper (brown) curve. As measured performance improves, actual performance decays as shown in the curve second closest to the x-axis (purple, 50 cycles) and the curve closest to the x-axis (green curve, 200 cycles).

Performance in this context seems to be particularly difficult to measure automatically as it is not merely a quantifiable quality. Consequently, relying on any quantitative "deviation from target" performance measure as a stop criterion for training is particularly dangerous on these data, as continuing training often lead to enhanced performance measures as the ANN is guided towards always predicting negatively or "not stuck pipe".

The remaining reliable iteration count strategy seems to be early stopping, merely guided by a simple performance measure.

Consequently, in the remaining of the analysis, RMSE is used as a guideline for early stopping and validation by manual inspection of test wells.

To summarize: Performance is guided by RMSE measures and validated on a separate data set by manual inspection.

In the next section, strategies for empiric and feasible optimization of the net is discussed,:

6.7.3 Empirical search for optimal ANN configuration

The following, initial tests were performed testing the net on the Test setup described in 6.7.1 and following the net development strategy and performance measurement derived in chapters 6.6 and 6.7,,

Topology development

introduction of the final hidden layer might serve as a typical example of a structural development increment: At this stage the ANN had the general arrangement as suggested in Figure 6.5: Right asymmetrical topology, more specifically 11 x 2 x 16 x 1

. This was attempted enhanced by adding PE's to the existing layers up to the point of achieving marginal enhancements, if not counterproductive behavior. Introduction of a third hidden layer, however, gave noticeable performance enhancements although comprised by merely two PEs.

The topology producing the cumulatively best results is a 3 hidden layer, right asymmetric topology shown in Figure 6.7.

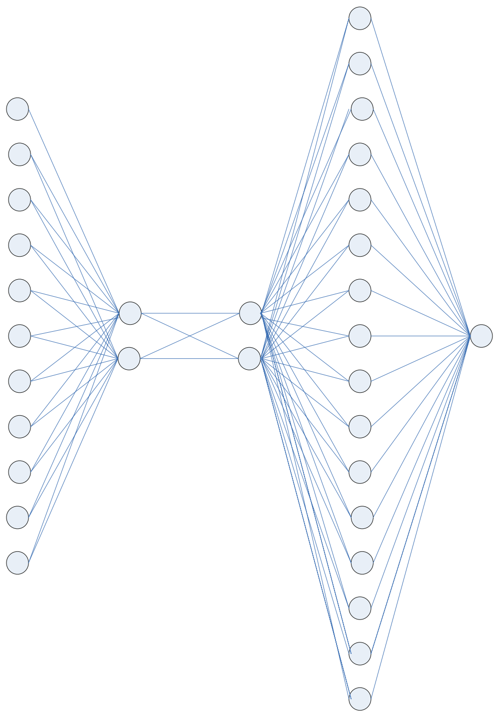


Figure 6.7: Topology for the pre- processed MWD data.

Consequently: *All training referenced in these results is based on the 11 x 2 x 2 x 16 x 1 topology.*

Stop criterion for training

There were ultimately 4 training methods tested based on RMSE:

1. A common and simple strategy is using a fixed stop value, i.e. continue iterating through the training data until a preset RMSE value has been reached:

   Algorithm 6-1: Stop- at- reached- RMSE

   $$\text{while}\left(\frac{d(\text{RMSE}_i)}{d(\text{Cycle}_i)} \prec c\right)\{\text{train}\}$$, where c is a real valued, positive stop criterion.

2. Early stopping criterion based on the RMSE gradient that simply stops training as soon as the RMSE gradient ceases descending:

   Algorithm 6-2: Descending RMSE gradient stop criterion

   $$\text{while}\left(\frac{d(\text{RMSE}_i)}{d(\text{Cycle}_i)} \prec 0\right)\{\text{train}\}$$

3. A primitive “n times non- descending gradient” stop criterion was tested maintaining a stack of the n last RMSE readings in an attempt to avoid stopping at early local minima:

Algorithm 6-3: Continually descending gradient stop criterion

$$\text{while}\left(\left(!\frac{d(\text{RMSE}_i)}{d(\text{Cycle}_i)} \succ 0\right) \&\& \left(\sum_{i-n}^{i}\frac{d(\text{RMSE}_i)}{d(\text{Cycle}_i)} \prec 0\right)\right)\{\text{train}\}$$

4. Using no error stop criterion, setting a fixed number of iterations based on manual validation and empirical testing if such a favorable number of iterations exist.

Besides the simple criterion discussed above, only the non- performance based fixed number of iterations remains, frequently used for testing purposes.

A typical training session results in a continuously improving apparent error trend:

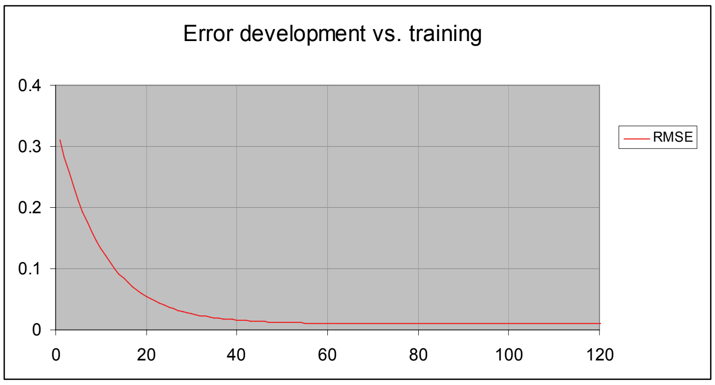


Figure 6.8: Measured error vs. training iterations

Although the error slope show a decreasing improvement tendency, training sessions follow variations of Figure 6.8 pattern almost exclusively. At reading 120, the last iteration shown in the figure, there still is an infinitesimally small improvement in RMSE compared to the last iteration. Consequently neither of the algorithms 2 or 3 gave functional stopping criterions as both presuppose reaching non- descending gradients.

Numerous variations of training sessions were run based on both strategies 1 and 4, fixed RMSE and number of training cycles respectively.
The best performing ANNs seemed to be those batch- trained through approximately 15 - 30 cycles, regardless of resulting RMSE.

Data pre- processing

As Chapter 2.2"Data description" and the preliminary analyses indicated, only training and validation data having removed the outlying feature values prior to normalization gave interpretable results. The degree of noise both at training example (well) level and feature level vary, but within the scope of this work it was unfeasible to develop an automated method customizing the degree of noise reduction. Lacking such an algorithm it was decided to use a global empirically derived value.

All training data and validation data had 10% of the outlying feature values replaced by the remaining extreme value as described in Chapter 6.5 and implemented in MathLib.java prior to normalization. Unless allowing fine-tuning of the outlying thresholds across wells, this was the optimal acceptable global noise removal prior to normalization.

ANN tuning – activation functions, α and η values

Experiments were conducted following the training scheme of section 6.6.3 in an attempt to establish the impact of ANN tuning parameters, narrowing down the set of advantageous α and η- values and assuming the predominant performance factor to be topology.

Sensitivity to over/under- training was evident at an early stage of testing which effectively limited the number of iterations feasible. Allowing few iterations and initial randomization of weights made it difficult producing comparable graphs, thus studying the effects of α- and η – variations isolatedly. Learning rates below 0.1, however, lead to poor performance and often an increasing RMSE gradient. Lacking empiric foundation to guide the choice of α and η values, and to keep the empiric search space at an acceptable level, *the initial momenta and learning rates were kept at 0.1 and 0.8 respectively, uniformly across layers.*

Choice of activation function, however, has a large effect on the net predictions.

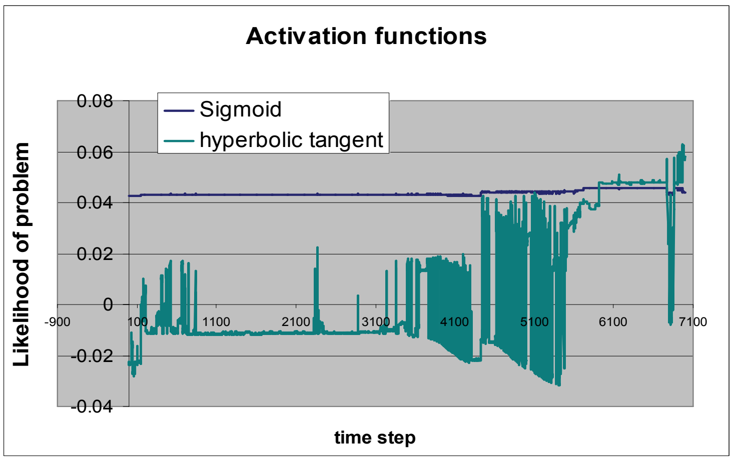


Figure 6.9: Impact of activation functions

As shown in Figure 6.9: Impact of activation functions, the “widespread” (green) hyperbolic tangential activation function maximize node activation thus significantly amplifies the signal amplitudes as opposed to the more commonly used sigmoidal function, shown as the “flat” (blue) curve. As node activation significantly influences weight tuning during training (Equation 5.5), as well as the calculated output (Equation 5.4), this is not merely amplification of substantially the same signal and is of principal importance.

As the hyperbolic tangent produced the overall best interpretable results, this was used uniformly across all layers.

ANN training summarized

Experiments resulted in the following ANN setup and training scheme:
The best performing topology found in the topology search space were a three hidden layer ANN having a 11 x 2 x 2 x 16 x 1 configuration This was trained by 15-25 iterations through the training set using batch- training of sizes s = n/15 giving 15 folds and consequently 15 cumulative weight updates per iteration (see for instance ArtificialNeuralNet.nfoldBatchTraining, MathLib.getBatchSet). Net tuning utilized were hyperbolic tangent activation function, a learning rate and momentum of 0.1 and 0.8 respectively, uniformly across all layers.
This training scheme gives a feasible, adequately functional stop criterion, gave the best overall performance and produced the most consistent, stable predictions.

6.8 Results

Below are the results from training an ANN implementing the algorithms and tuning characteristics described in the subchapters above, shown successively from validation well 5 through 7. Outputs from validation runs have been normalized using the methods described in section 6.5 for more convenient interpretation, especially as all local minima and maxima have values of 0 and 1 respectively. Furthermore, all references to hours in the text are the time span from start of measurements until the referenced time.
Finally, unless explicitly stated the validation session presented, each comprised by the 3 validation wells, are outputs using the same set of weights and thresholds.

6.8.1 Some independant development results

Below are some independent results obtained during the search for a possible optimal number of iterations. Independent in this context means performance for an isolated well was evaluated disregarding outputs from the other validation wells. In these cases less fortunate performing outputs were achieved on the other validation wells and consequently these sets of weights and thresholds were dismissed.

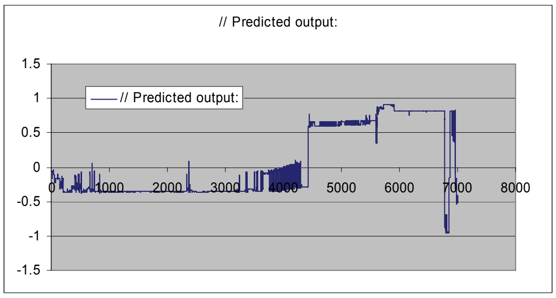


Figure 6.10: Early testing (raw data): Under-trained net: Early validation session using well 6, 10 iterations

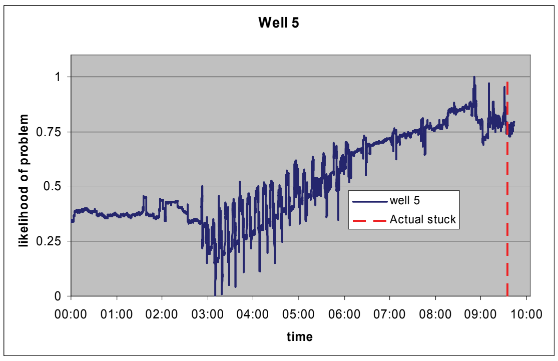


Figure 6.12 Validation using well 5, 30 iterations. Excellent performance on well but poor validation set performance

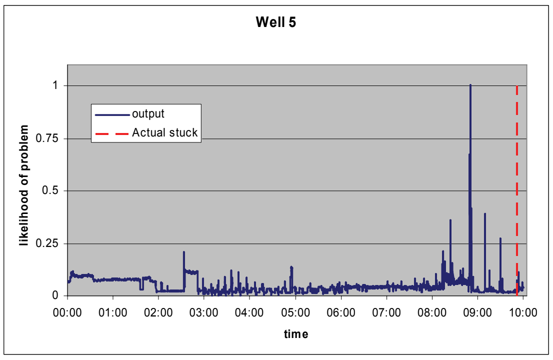


Figure 6.11: Possibly over- trained net: Validation using well 5, 40 iterations

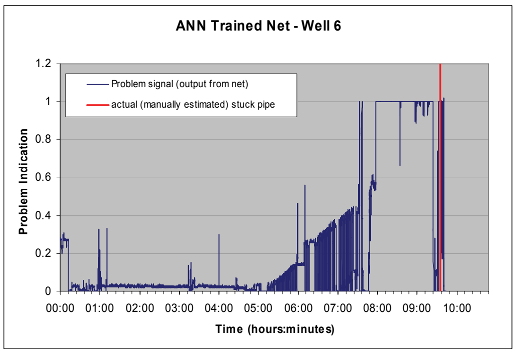


Figure 6.13: Validation using well 6, 25 iterations. Excellent performance on well but poor validation set performance

6.8.2 Experiments using 15 iterations

The following results show a validation session using a possibly undertrained ANN and are included to illustrate 2 points:

1. Demonstrating the impact of continuing training the net beyond the point of adequate performance, of which one example is included in the next subchapter
2. Showing all 3 validation wells using raw data in the same plot demonstrating how the signals are global, independent of training example (wells) and might serve as a un- interpreted standalone problem signal.

After tuning the weights of the net as illustrated in Figure 4-19 the ANN interpreted likelihood of the unwanted event "stuck pipe" for validation well 5 is shown below:

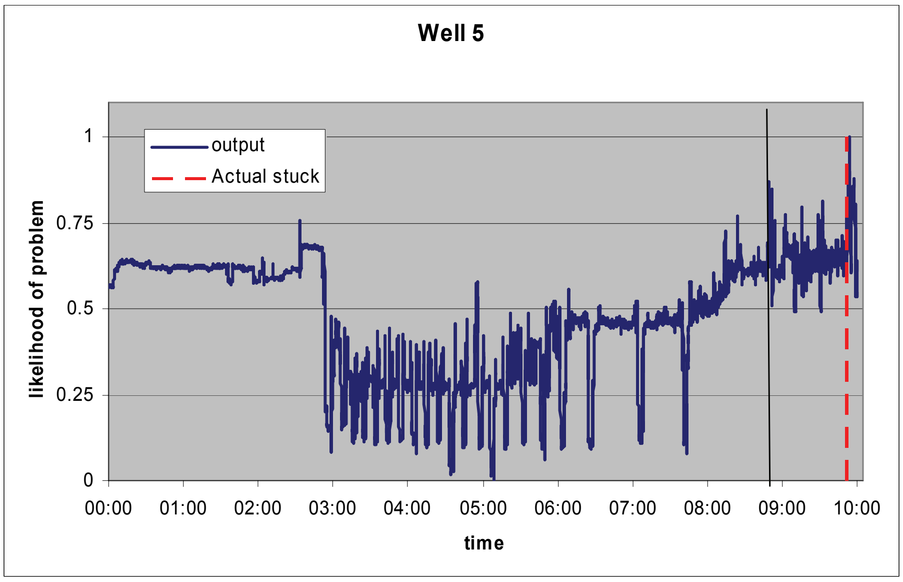


Figure 6.14: predicted likelihood of event – Well 5

The high entrance value and general fluctuating characteristics of the curve, especially from 02:50 to 06:10 hours, lead to a curve somewhat difficult to interpret. Local anomalies that might trigger warnings might appear as early as 02:40 hours. After a pronounced decrease in oscillation at approximately 07:40, there is an increasing trend towards the first of four maximum amplitudes occurring at 08:25. The second is the extremal value of this well prior to stuck pipe, reached at 08:51 (indicated as a grey line), 44 minutes prior to the recorded stuck pipe event. The trend of the curve then significantly drops until the third and second largest amplitude occurs 25 minutes prior to the event

.

The next validation well shown in Figure 6.15: predicted likelihood of event – Well 6 produces a "cleaner" output curve having little fluctuation tendencies as did the previous validation run.

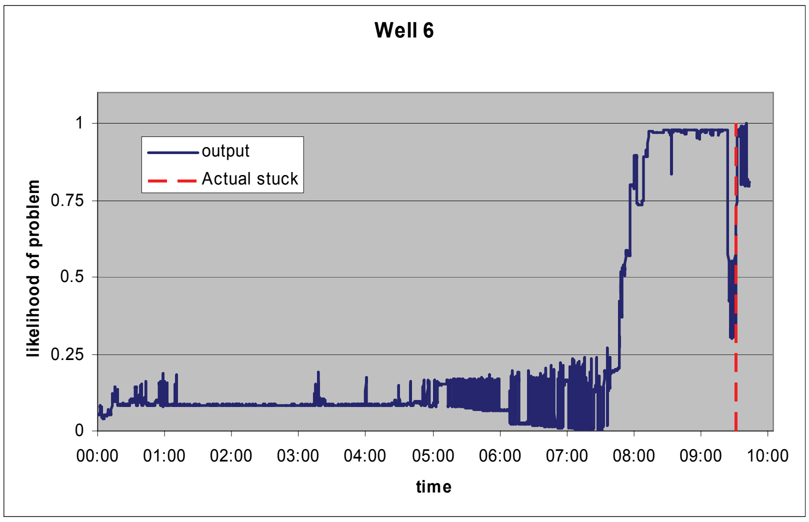


Figure 6.15: predicted likelihood of event – Well 6

Stabilizing at approximately 10%, the first prominent peak giving a signal at 100% occurs as early as 08:00, 1:35 hours prior to the registered unwanted event. The curve then briefly drops to 0.75, 50% above the stabilized "normal" then peaks and stabilizes at 08:18, 1:17 prior to the unwanted event. 3 minutes prior to the stuck pipe, the likelihood curve drops to 0%.

Output of the final validation well number 7 shown in Figure 6.16 shows the same severe fluctuating characteristics of validation well 5, but having intermittent zero- or close to zero- valued anomalies.

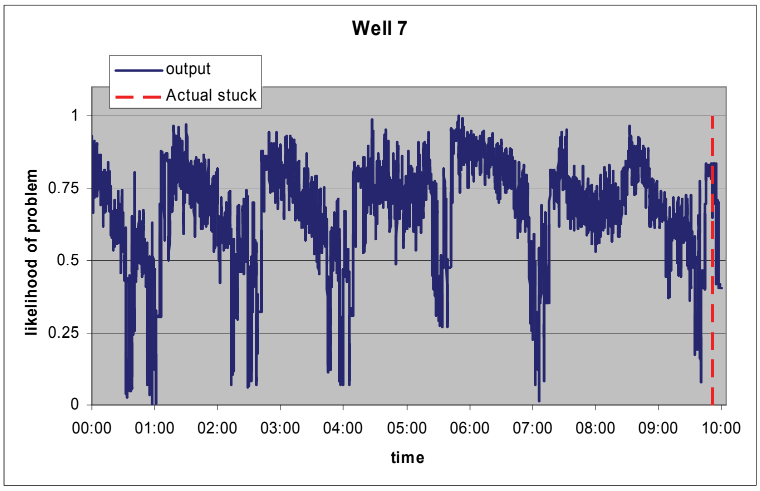


Figure 6.16: predicted likelihood of event – Well 7

Between each of these extremal anomalies the signal oscillates around an increasing "intermittent average". The maximum value occurs at approximately 06:00, nearly 4 hours prior to the registered unwanted event, then the signal steadily descends close to zero 10 prior to the recorded event.

As described introductorily, an important aspect of this chapter was demonstrating example independence of the ANN output. This is done showing the output signals from the validation run discussed above in the same graph:

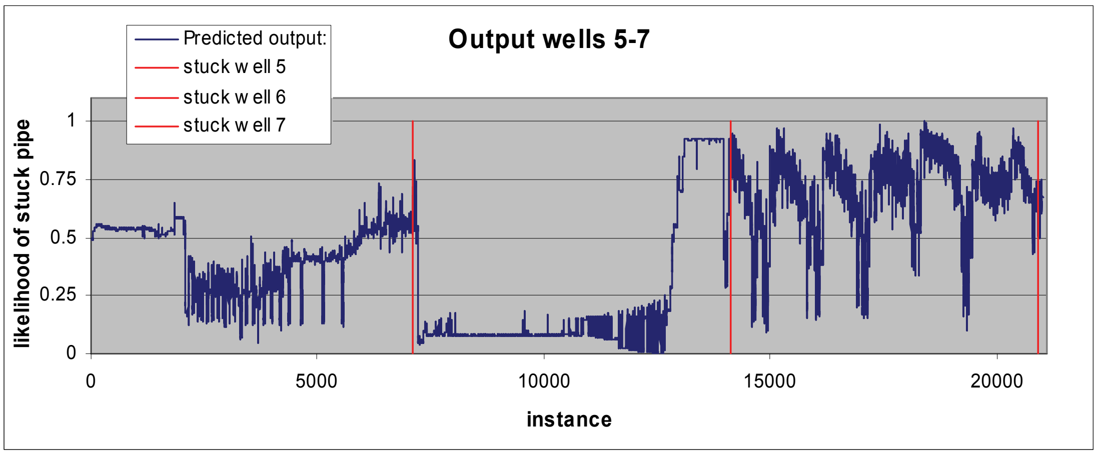


Figure 6.17: Unprocessed output from validation wells 5-7

All validation wells share an approximately common minimum value. Both validation wells 5 and 6 have value spans within comparable range and somewhat similar behaviour prior to stuck pipe. Validation well 7, however, differ both with respect to maximum amplitudes and behaviour prior to stuck pipe: Of the 6 intermittent periods of validation well 7 described above, each period is represented by a local maxima continually stretching the global value span.

6.8.3 Experiments using 25 iterations

The results presented in this section serve as a representative example of outputs produced by training the ANN using 20 iterations and the effect of altered performance resulting from increased training beyond the "adequate".

Predictions made on the first validation well, shown in Figure 6.18: ANN output - validation well 5, have a clearly increasing trend from 3 hrs peaking and stabilizing at 08:35, 1:18 hours prior to the unwanted event. The problems of interpreting the output of validation well 5 using the less trained net has decreased significantly, however, some of the problems remain:

The initial early warning described, remain, as does the fluctuating period. Meanwhile, the fluctuating period has an evident increasing build-up towards the recorded unwanted event.

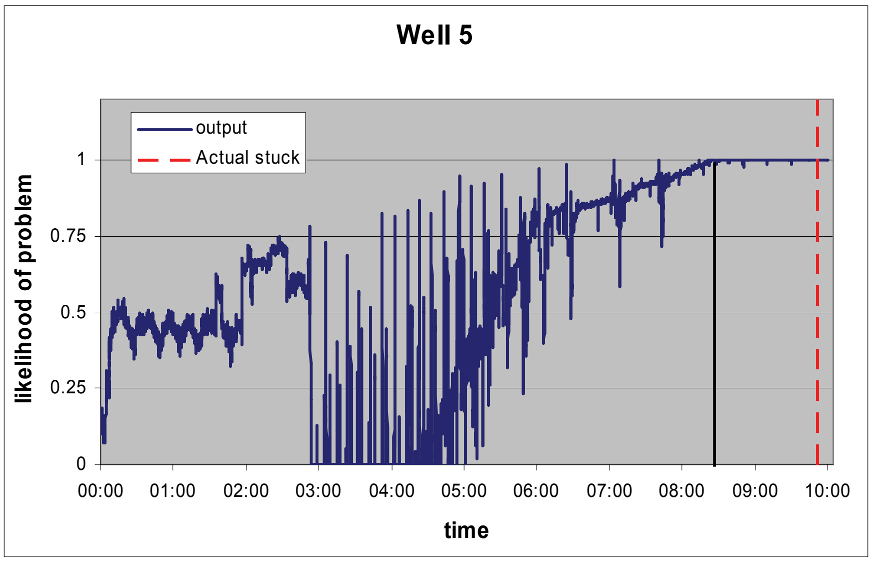


Figure 6.18: ANN output - validation well 5

A dampened signal using EMA (exponentially moving average), is shown in Figure 6.19: ANN output well 5 – signal smoothened by moving average:

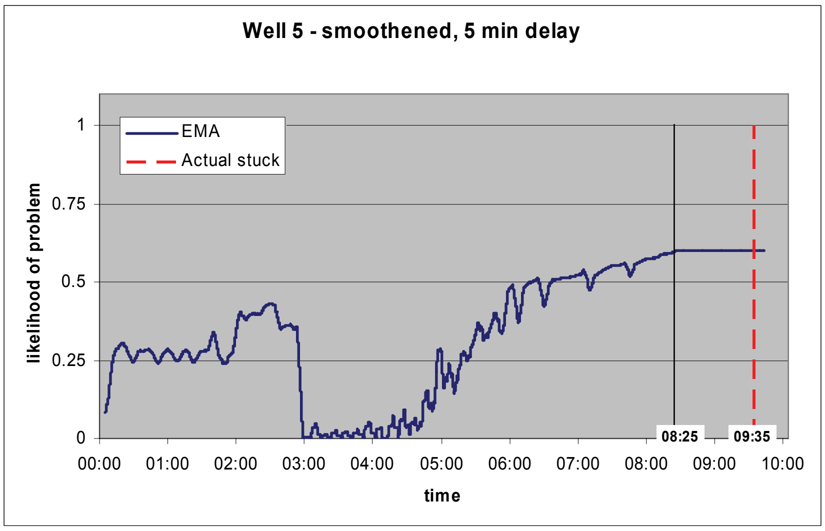


Figure 6.19: ANN output well 5 – signal smoothened by moving average

In this example, 60 readings are averaged prior to transmitting giving a 5 minute delay. As shown in the figure this significantly reduces the non- contributing fluctuations in the real- time example above.

Output of validation well 6 is shown below and start almost at output maximum, then rapidly descends close to zero.

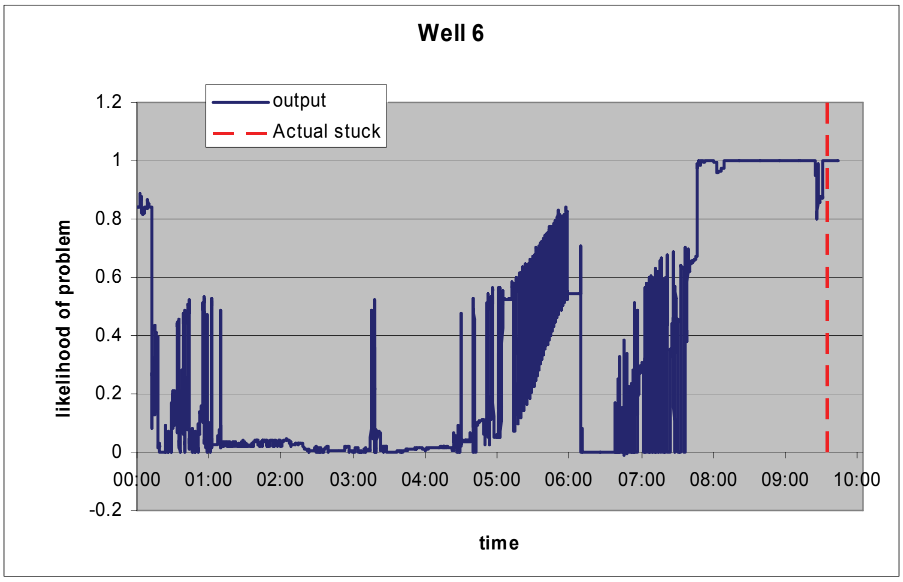


Figure 6.20: ANN output - validation well 6

After a brief period of noisy fluctuation, the predicted likelihood oscillates close to zero between 01:10 to 04:25 interrupted by an anomaly at 03:20. Until 05:15 the intermittent local maxima were limited to 0.6 of the maximum output value for this validation well, but past 05:15 this "temporary maximum" is surpassed and there is a consistently increasing trend to a new maxima of 0.82 of maximum output at 05:55. After a rapid drop to zero a similar period occur starting at 06:35 reaching and stabilizing at its output maxima at approximately 08:00 hours.

Output from the final validation well is shown in Figure 6.21. Compared to the previous output from this validation well, there are oscillations around trends rather than severe fluctuations.

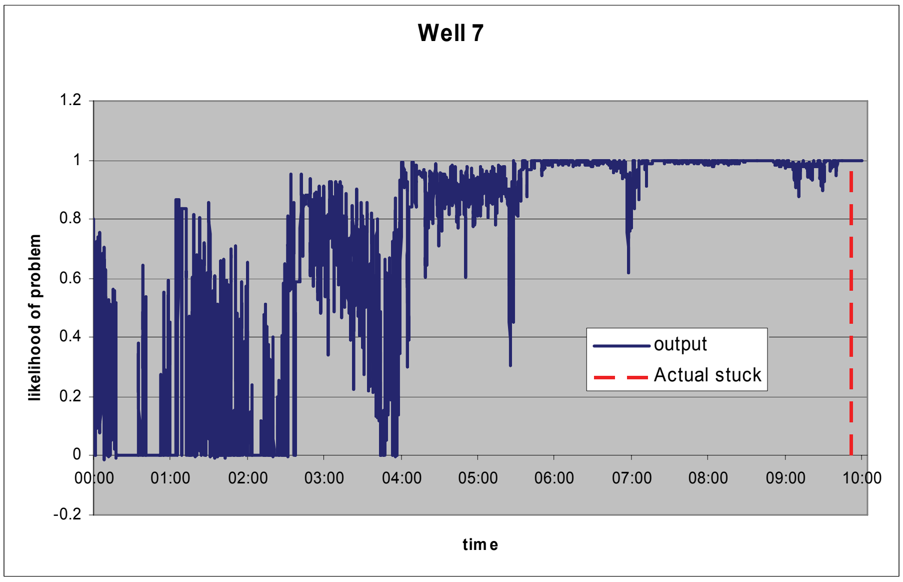


Figure 6.21: ANN output Well7

Nevertheless, as for the previous validation session of this well, there are distinct intermittent peaks, each having localized maxima. There is an increasingly consistent trend towards this validation sessions maximum, reached already at 05:35. From that point the signal essentially oscillate around this session maximum.

6.9 Discussion of results

In this subchapter, experiences from the testing are evaluated, first with respect to the ANN training algorithms, configuration and tuning, then with respect to the produced results.

6.9.1 Training and validation discussion

The testing includes training and evaluation. Training involves adjusting the tuning parameters available, while evaluation of results, as demonstrated in the previous chapter, implicitly depend on stopping criterion of the training.

These are the toplics discussed in the next sub- chapter:

Training algorithm

Only randomized N-folded batch- training produced acceptable outputs. Failure of the common sequential instance processing was expected due to the uniform and severely skewed instance classification distribution. The qualitative necessity of batch training was unexpected, however. The initial, main rationale for implementing the modified batch-processing extension was reducing the number of weight updates, thus reducing processing time. However, considering the massive amount of training data and the categorical, skewed distributed instance classification, the "rougher" cumulative weight updates offered by batch training might be favorable also from a qualitative point of view.

Performance measurement and stopping criterion for training

As a consequence of not being able to utilize traditional performance measures as discussed in section 6.7.2, and due to the slowly descending RMSE gradient and the need for early stopping, no functioning stopping algorithm could be implemented. Consequently, the only remaining of the possible stopping criterion discussed in section 6.7.3 was the fixed RMSE limit and fixed number of iterations.

A RMSE criterion might be reached after unrepresentatively few or many iterations due to the random initialization of weights and and the narrow set of productive iteration. This might lead to an over –or undertrained net of which the flat (purple) over- fitted curve and the fluctuating (grey) under- trained curve of Figure 6.6 are example. Actual performance as described in Sub- section 6.7.2 seems to be particularly sensitive to over-training. As described in Chapter 2 the training data consist of relatively few true *training examples*, although cumulatively many *training instances*, which might explain the ANNs susceptibility for early adaptation of the training data peculiarities rather than true generalization.

Controversially but as a consequence of the above, a fixed number of iterations was the best performing stopping criterion.

ANN parameters and tuning

The sovereign most influential ANN parameter on performance proved to be net topology. Less sensitivity to the tuning parameters was expected, however there were unexpected peculiarities that need elaboration:

Using momenta within range $\langle 0.5, 0.8 \rangle$ made this parameter seem uninfluential, but outside this range performance dropped, and increasing RMSE gradients were typically observed. The range of acceptable learning rates was narrow; $\langle 0.1, 0.2 \rangle$, in addition to having the potential counterproductive behavior outside this range.

As discussed in Section 2.2 "Data Description" there are several plausible explanations for these findings:

1. Few independent training examples (wells) allow only a narrow range of constructive training iterations and each number of training iterations is small.
2. The dicotome (binary) classified dependant variable result in a set of training data without a potential for fine tuning

The first point need some further elaboration and implies training has to be performed rapidly, which might explain the lower bound of constructive learning rates. The upper bound might be explained equivalently; too high learning rate leads to too large learning steps which is unfortunate given the few iterations constituting the training. The few allowable iterations might also explain the seemingly uninfluential momentum as there, for instance, is little room or danger for oscillations.

Data pre- and post processing

Although a global noise reduction value reduces complexity, allows for effortless automation and might provide a significantly improved dataset, as in all noise removal there is always a risk of loosing valuable anomalies. Actual, noiseless extremal- values might be the most valuable in predicting deviations from a normal situation, therefore setting the outlier threshold becomes a trade- off between the risk of loosing valuable anomalies and properly normalizing the dataset.
As there was no objective, quantitative or qualitative performance measure of noise reduction utilized in this work, valuable anomalies might have been lost and/or noise might have disturbed the normalization of the MWD data

One validation example was susceptible to periods of severe fluctuations and output could benefit from signal smoothening. The need for such signal dampening might be more of an indication of modeling problems or scarce data rather than a proper alternative for improving signal interpretability and further treatment.

6.9.2 Evaluation of results

Below is a validation of the descriptive results analysis above.

Validation well 5

The predicted likelihood of event of validation well 5 shown in Figure 6.16 had 4 distinct maximum amplitudes, all of which were within 1:30 hours, and the second largest 25 minutes prior, to the recorded unwanted event. This would make it a good problem predictor. However, within the same timeframe there are periods of a scattered characteristic within which there are amplitudes that might cause problems establishing an adequate threshold and transmitting consistent and reliable warning signals. Moreover, the trend towards the stuck pipe event is somewhat weakened by the initial 3 hours of relatively high output, thus the signal is insufficient to cumulatively give a clean output signal thought a possible moving average. This is feasible using the more trained net on this validation well as shown in Figure 6.19, with the delay and lag cost involved as discussed in the "output signal smoothening" section of subchapter 6.5. Moreover, the relatively long warning period prior to stuck pipe might cause predictive problems from a practical perspective.

Validation well 6:

The output of validation well 6 produced by the less trained net shown in Figure 6.15 has few evident weaknesses but possibly one; the maxima occurring at 1:30 hours followed by a prominent drop. Although the drop in system alert state after a warning of this magnitude might be erroneously interpreted as normal, the likelihood of an unwanted event remains significantly high until the recorded event. The brief anomaly giving this maximum should not lead to for instance a threshold adjustment leaving the unwanted event uncaught by the system.

The output produced by the more trained net (Figure 6.20) are surprisingly more scattered and produces a curve showing two periods both apparently building up to an unwanted event. The latter period however, build up to a consistent plateau representing this validation sessions maximum. Independently of the two ANNs used this validation session would be of near benchmarking predictive quality overseeing a potential problem: As for validation well 5, the possibly long period of warning prior to the actual event might indicate a weakness if a post analysis prove these early warnings wrong.

Validation well 7:

The less trained net produced outputs having an overall noisy characteristic and intermittently severe fluctuations. Moreover, there is a consistently decreasing trend towards the unwanted event having its minima 1:06 hours prior to the recorded event. This output signal is feasibly useable as a warning signal, as a possible threshold would have to be elevated at an early stage. Referring to Figure 6.16, the threshold elevation would occur already at the start of data readings. Nevertheless, having 6 intermittent periods, each having extreme local maxima is difficult to interpret for the signal receiver.

Furthermore, relative to the other validation wells, outputs produced instantly and consistently surpass the global maxima for this validation pool.
The more trained net produced an interpretably enhanced curve by reducing the severe fluctuations to oscillations around a more pronounced inclination towards the unwanted event. Although improved, the same intermittent periods of local minima and maxima as from the less trained net remain. Furthermore, this additional training leads to an output producing a constant warning signal approximately 4 hours prior to the recorded event.
Independently of training extent, this well either represents an outlying example or could be a symptom of a model weakness. Only a DDR clearly indicating stuck pipe related problems during the entire recorded interval could justify the output curve of validation well 7.

6.9.3 Common evaluation

It is difficult to discriminate qualitatively between the 15- and 25- iteration test runs. The general trend is as expected a smoother output for the more trained net and strengthening of trends less expressed in the less trained net. However, there are exceptions: Validation well 6 shows a cleaner, and benchmarking trend in the 15-iterative run. This might be caused by a trend not fully developed, indicated by the increasingly scattered period starting at 05:00 hours but still underdeveloped and fully expressed in the 25- iteration run.
The general trend remain: As the data tolerate 25 iterations, cleaner and more representative patterns emerge when training upon the iterative tolerance of 25.

All useable predictions occur at least 1:10 hours prior to the recorded unwanted event. This might be a data issue caused by the rough estimate of the stuck pipe occurrence, or a causal, drilling related issue; the fact that there were stuck pipe problem indications reflected in the MWD data as early as indicated or a combination of both. On the other hand, this might be an indication of a modeling problem and consequently a model failure indication..

6.9.4 Prerequisites of a functional warning signal

Regardless of the success in training the ANN, both successfully tested wells would benefit from having a moving, not a static, global warning threshold. Figure 6.17 could suggest a global warning signal of 0.75 would set off appropriate alarms for validation wells 5 and 6. Nonetheless, if such a global threshold is set too high, the warning signal amplitude might never reach this threshold, thus never catching an unvanted event. On the other hand, if this threshold is set too low, repeatedly erroneous signals might overload the Deep reasoning- module or continuously set off invalid alarms. Moreover, variations in validation results strengthens the need for a moving threshold, which presuppose a negative response on faulty warnings.

Following this sub- chapter are some general concerns regarding the reliability of the results presented,:

6.10 Validity and reliability of the results

The strict separation of a training data set (Well 1-4) and a test data set (Wells 5-7) assure some reliability of the validation process. However, although indirectly, the test data set is used in performance measurements in the development of the presented e-SBP ANN. Together with the few true training examples, this might somewhat weaken the reliability of the results presented. The supereminent goal of all AI systems is producing a generalizable system, which depend on valid test results. Correspondingly, validity depend on reliable "instruments of measurement". The only fully reliable test is gathering fresh data and test on this system.

6.11 Conclusions

Training

Training and validation require much "needlework" as automated performance evaluation is unfeasible. The high sensitivity to overtraining leaves little opportunity for, or gain by, using standard tuning parameters.

General results

3 wells were randomly selected and placed in a separate validation pool. Of these 3 wells , 2 were promisingly predicted while the third was less than successfully predicted.

Despite the challenges of Chapter 2 ("Methods and Design"), Chapter 4 ("Attribute Selection By Statistical Analysis") combined with this chapter have shown feasibility of processing the MWD through a trained ANN in real- time warning signal generation

Furthermore, once trained, outputs of the net are globally comparable and might serve as one of possibly several independent parameters in a warning signal to the next stage. As such, the warning signal generated by this ANN is a fully qualified and independent, real-time, problem specific operational warning signal..

However, to achieve a reliable and appropriate warning signal, the "Prediction Module" might depend on feedback from the warning signal receiver to enable the necessary moving threshold.

To summarize: Given the reliability of the validation scheme used and pending further validation, the presented enhanced ANN seem suitable as a technique for the task of achieving Goal 1.

CHAPTER 7

DISCUSSION AND FURTHER WORK

In this chapter, the cumulative results of Chapters 3 - 6 are collected, and measured upon the goals set for this work. The outcome of this analysis decides the extent of work possibly remaining. However, the goal of this thesis was never to implement the system described in the Thesis goals. This means that some work remains and will be outlined in the last sub- chapter.

7.1 Results

The results are presented as a practical training, validation and real- time prediction scenario. Unless explicitly stated, all functionality has either been analyzed and documented (Chapters 3-4) or implemented and demonstrated (Chapter 6). References to the relevant chapters or sections are given sequentially.

One of the major advantages of the ANN as a prediction unit is that once trained, the majority of the work is done (Section 5.1) which will be emphasized in Figure 7.2.
Figure 7.1 show the resulting system during training:

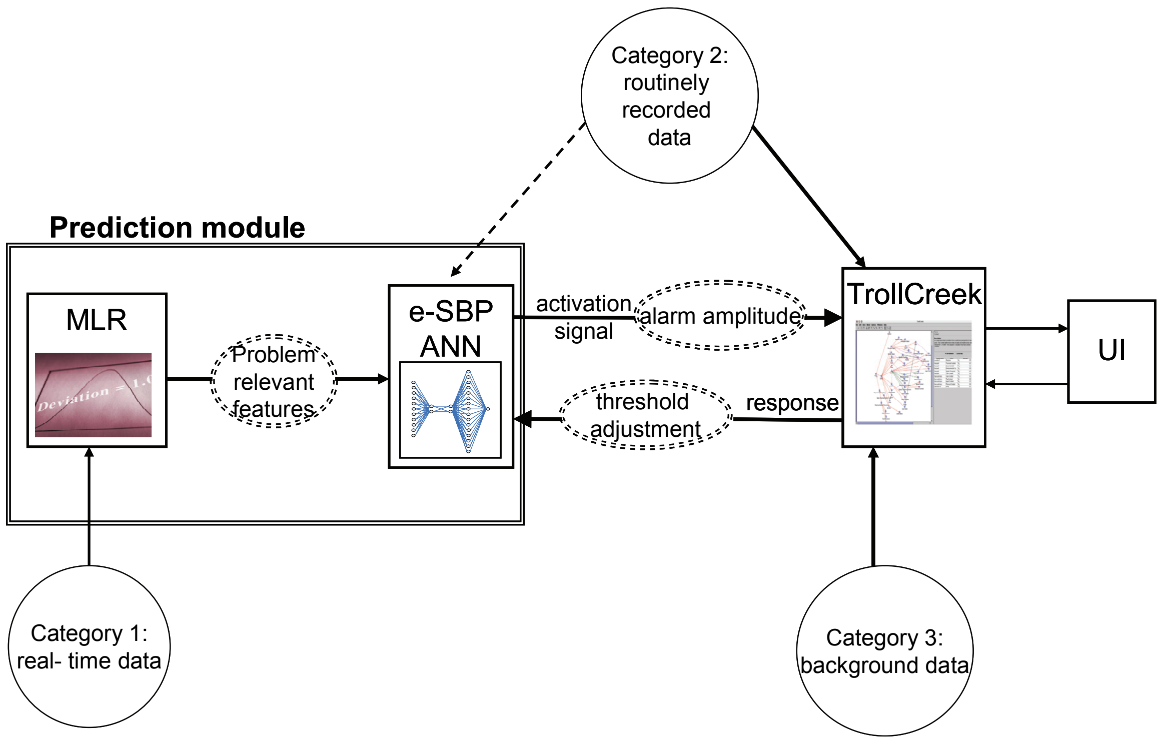


Figure 7.1: Complete system architecture, training phase - (final iteration)

At this stage, the system architecture is finalized, but the following testing and prediction scenarios presupposes three non- addressed issues in this work; Implemental Prerequisites.

1. The presence of a somewhat complete Case Base as described in Chapter 3
2. An interface between the Prediction module and TrollCreek
3. The presence of a threshold value, either dynamic or global and static, as described in Section 2.3.3
4. The response signal resulting in either an incremental threshold increase or decrease, as described in (dummy)

7.1.1 System training – fully implemented

This is a presentation of a possible training session, prior to deployment. The "Category 1 data" input to the MLR sub- module in Figure 7.1 is the training data pool, as demonstrated in Chapter 6:

Data enters the MLR (Multiple Logistical Regression) module where 33 parameters are reduced to 11 problem relevant features, which was completed in Chapter 4. These

features constitute the input layer of the e-SBP ANN. The training session result in a trained net as described in Chapter 6.
A possible testing scenario would be replacing the training data pool by an external validation data pool, and then initialise a simulation:

Prediction is followed by an alarm, the alarm signal then invokes the CBR- cycle which either dismisses or recognises the alarm. The retrieval and implicitly matching of possibly similar cases and learning step following an alarm, confirmed or rejected, in TrollCreek is described in Chapter 3. Finally, given a dynamic threshold, a consequence of a dismissed alarm in the Prediction module would be an incremental increase of the threshold.

7.1.2 Real- time prediction

Figure 7.2 shows the remaining, active parts of the system once trained indicated by a simple summarization symbol in the ANN sub- module.. The computational expensive ANN training steps are performed and simple feed- forward summarizations remain (Section Dummy) Furthermore, the "Category 1 data" input is presently the actual MWD data recorded real- time.

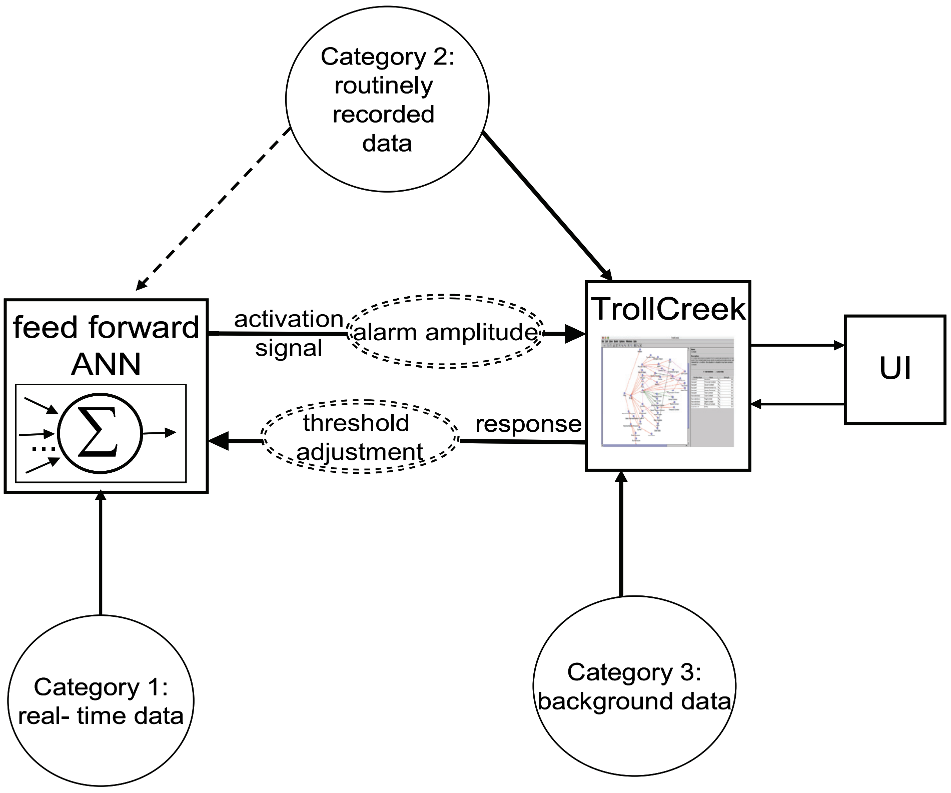

Category 2: routinely recorded data
feed forward ANN
activation signal
alarm amplitude
TrollCreek
UI
threshold adjustment
response
Category 1: real- time data
Category 3: background data

Figure 7.2: System architecture, prediction phase - final iteration

Process flow and a possible threshold adjustment in the Prediction module is performed as described in the testing step of Section 7.1.1

7.2 Discussion of results

In this section the presented results are measured against the goals of this thesis.

Goal 1: Real- time, computational efficient prediction

The implemented e-SBP ANN does perform computational efficient predictions under "Goal 1 conditions":

1. A global threshold for warning is sufficient
2. The trained net produced in Chapter 6 is reliable using the performance measurement (Section 6.11) and still holds when trained and validated using other, external data

Goal 1 condition 1: The trained net producing the outputs of Figure 6.17 supports this hypothesis. However, although *trends* remain from one training session to another, there are significant qualitative variations in the ANNs produced, thus a global threshold for warning is recommendable.

Goal 1 condition 2: If valid, so is Sub- hypothesis 1.1 of Section 1.1.1, and given the simple feed- forward net in the prediction module, Sub- hypothesis 1.2 of Section 1.1.1 has been proven valid.

Goal 2: Provide justified explanations, recommendations and learning capabilities
Interpreted alone, Chapter 3 concludes that Goal 2 is reached. However, considering the Implemental Prerequisites of Section 2.1, it is still necessary to *initiate* the CBR cycle facilitating explanations, recommendations and learning. The initiation above is a reliable warning signal depending on the validity of the Goal 1 conditions above.

7.3 Conclusions

The discussion above shows that both goals of this project is reached provided the following extensions of the presented work:

1. An interface between, or integration of, the Prediction module and TrollCreek allowing message exchange
2. Implementation of a threshold value, possibly global and static but preferable dynamic

Furthermore, the ability to generalise beyond the presented dataset depends on reliable trained Artificial Neural Nets using the current performance measurement.

7.4 Further work

The primary target of this thesis was reaching the goals. Although mainly achieved, some implemental issues pointed out in the conclusions part, remain unsolved:

7.4.1 Implemental issues

The following points are needed to fully achievethe goals of this work from an implementation perspective:

- As a minimum; add a simple static threshold value structure to the existing for automatic unwanted event detection and reporting, but preferably a dynamic threshold value.
- Facilitate message exchange between the Prediction module and TrollCreek or integrate the two.

Further issues remain as there still are minor bugs in the code made available:

- completely debug the e-ANN to allow multiple, separate training sessions without re-loading the classes
- complete the User interface to make the code more user- friendly and intuitive

7.4.2 Validation issues

As pointed out in the conclusions, the results of this work rely on the reliability of the results presented in Chapter 6:

Qualitative validation of the results in this thesis

Qualitative remarks regarding reliability in this thesis could be clarified having and a domain expert analysing the DDR (Daily Drilling Report) supporting the validation analysis, e.g. possibly explaining why a well (7) is particularly difficult to analyze and predict, and clarify whether the consistently early warnings from ANN outputs are model inaccuracies or valuable, genuine diagnosis of an unwanted event in progress.

Quantitative validation expanding the data- pool

The definitive validation of this work is realized when expanding the existing data- pool of 7 wells significantly by adding more true examples, that is, more wells. More extensive datapool would possibly allow:

- opportunity for more fine tuning of net tuning parameters (differentiating learning rates, momenta, and activation funfunctions flatness)
- Asa consequence of the above: more fine tuning of net parameters across layers
- might allow for automatic performance measure as n-fold cross validation

To summarize: Expandingt the data- pool would possibly qualitative improve the results, in addition to the important validation and facilitate adequate generalization accomplished by automation and less work intensive empiric “needlework”.

LIST OF REFERENCES

[Allison, Paul D. 1999] “Comparing logit and probit coefficients across groups”, Sociological Methods and Research, 28(2): 186-208

[Arango, G., Colley, N., Connely, C., Durbec C. 1997] “What’s in IT for us”, Schlumberger Oilfield Review:
http://www.oilfield.slb.com/media/services/
resources/oilfieldreview/ors97/aut97/init.pdf

[Bailey L, Jones T, Belaskie J, Ortban J, Sheppard M, Houwen O, Jardine S and McCann D 1991] “Stuck Pipe: Cases, Detection any Prevention" Oilfield Review 3, no. 2 (October) pp. 13-26.

[Bertsekas, D. P. and Tsitsiklis, J. N. 1996] “Neuro-Dynamic Programming”, Belmont, Athena Scientific, ch. 3.2

[Broomhead, D.S., Lowe, D. 1988] “Multivariable function interpolation and adaptive networks” Complex Systems 2, 321-335.

[Castiglione, F. 2001] “Forecasting price increments using an artificial Neural Network, in Complex Dynamics in Economics, a Special Issue of Advances in Complex Systems”, 3(1), 45-56, Hermes-Oxford

[Canterbury District Health Board, 2003]
http://www.cdhb.govt.nz/glossary.htm

[Doraisamy, H., Ertekin, T., Grader, A. 1998] "Key Parameters Controlling the Performance of Neuro Simulation Applications in Field Development," SPE 51079, Proceedings, SPE Eastern Regional Conference Nov. 9-11, Pittsburgh, PA.

[Fox, John 2000] “Multiple and generalized nonparametric regression”. Thousand Oaks, CA: Sage Publications. Quantitative Applications in the Social Sciences Series No. 13 1.

[Freeman, J. A and Skapura, D. M 1991] "Neural Networks – Algorithms, Applications and Programming Techniques", Addison-Wesley Publishing Co Reading, MA(1992)

[Friel,C. M. 2001] "Estimating Nonlinear functions".
http://www.shsu.edu/~icc_cmf/cj_789

[Gruber, T.R 1996] "The Foundations of Knowledge Acquisition" Vol. 4 pp. 115-131, London Academic Press

[Hartemink, A. 2001] "Principled computational methods for the validation and discovery of genetic regulatory networks" Massachusetts Institute of Technology, Ph. D. dissertation.

[Hosmer, D.W., & Lemeshow, S. 1989] "Applied Logistic Regression" New York: John Wiley and Sons.

[Howell, D. C., 1995] "Statistical Methods for Psychology, Third edition" .
Belmont, Carlifonia: Duxbury Press

[Intelligent Solutions, 2005]:
http://www.intelligentsolutionsinc.com

[Kleinbaum, D. G. 1994)] "Logistic regression: A self-learning text" New York: Springer Verlag

[Kohavi, R. 1995] "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection", International Joint. Conference on Artificial Intelligence (I.JCAI)

[Lehman, L., V. 2004] "Mitigating Drilling and Completion Risks Through Expert Planning" Scandinavian Oil/Gas Magazine 11/12 2004

[**Luger, G. F. 2002]** "Artificial Intelligence, Structures and Strategies for Complex Problem Solving" Addison Wesley

[McGarry, K., J., Wermter, S., MacIntyre, J, 1999] "Knowledge Extraction from Radial Basis Function Networks and Multi-layer Perceptrons", International Joint Conference on Neural Networks, Washington D.C, July 10th-16th, 1999

[Millian, J.R., 1994] "Learning efficient reactive behavioral sequences from basic reflexes in a goal-directed autonomous robot" In Proceedings of the third International Conference on Simulation of Adaptive Behavior

[**Mitchell, T. M. 1997]** "Machine Learning" McGraw-Hill International Editions

[Mohaghegh, S., Arefi, R., Bilgesu, I., Ameri, S. 1995] "Design and Development of an Artificial Neural Network for Estimation of Formation Permeability," SPE Computer Applications, December, pp. 151-154.

[Moving Averages 1] ChartSchool - StockCharts.com
http://www.stockcharts.com/education/IndicatorAnalysis/indic_movingAvg.html

[Moving Averages 2] ForexTrainning
http://www.forex-training.com/moving_average.htm

[NeuFrame] "Neusciences NeuFrame 4.0.0.0", Neusciences 2000 Neusciences:
http://www.neusciences.com/index.htm

Norusis, M. 1993] "SPSS for Windows: Advanced Statistics", Release 6.0. SPSS Inc. Chicago

[Office of Statistical and Economic Analysis U.S. Maritime Administration 2005] Offshore Drill Rig Market Indicators July 2005":
http://www.marad.dot.gov/marad_statistics/

[Osprey Risk 2004] Schlumberger Drilling Software: http://www.oilfield.slb.com/content/services/software/drilling/osprey_risk.asp

[Partridge, D., Abidi, S. S. R., and Goh, A. 1996] "Neural Network Applications in Medicine" Proceedings of National Conference on Research and Development in Computer Science and Its Applications REDECS'96 pp. 20 - 23.

[Passold, F., Ojeda, R. G., and Mur, J. 1996] "Hybrid Expert System in Anesthesiology for Critical Patients" In Proceedings of the 8th IEEE Mediterranean Electrotecnical Conference - MELECON'96 (ITALIA), Vol. III, pp. 1486-1489.

[Petrel Classification 2005] Schlumberger Geology and Reservoir Software: http://www.oilfield.slb.com/content/services/software/geo/petrel/classification_estimation.asp

[Randall, D., Tony R. Martinez, T., R. 2003] "The general inefficiency of batch training for gradient descent learning" Neural Networks archive Volume 16, Issue 10 (December 2003) pp. 1429 - 1451

[Rumelhart, D.E, Hinton, G. E. and Williams, R. J. 1986] "Learning representation by back- propagating errors" Nature 323

[Salehi, I 2004] "PUMP Project: Quantifying Best Practice Analysis to Cut Costs and Boost Output "GasTips, Vol 10 Number 2, Gas Technology Institute, the U.S. Department of Energy and Hart Energy Publishing, LP: pp. 25-31

[Shelley, R., Stephenson, S., Haley, W., Craig, E. 1998] "Red Fork Completion Analysis with the Aid of Artificial Neural Networks," SPE 39963, Proceedings, Rocky Mountain Regional Meeting / Low Permeability Reservoir Symposium, April 5-8, Denver, CO

[Skalle, P., Aamodt, A. Sveen, J. 1998] "Case-Based Reasoning, a method for gaining experience and giving advise on how to avoid and how to free stuck drill strings" Proceedings of IADC Middle East Drilling Conference, Dubai, November 1998.

[Skalle, P., Aamodt, A. 2004]: Knowledge-based decision support in oil well drilling. Zhongzhi Shi (ed.), Proceedings of the ICIIP 2004, International Conference on Intelligent Information Systems. Beijing, China, October 21 – 23

[Tabachnick, B. G. & Fidell, L. S, 1989] "Using Multivariate Statistic, Second Editions," New York: Harper Collins Publishers: pp. 79

[TrollCreek I] Skalle, P., Sørmo, F., Aamodt, A., "How to model in TrollCreek in the Petroleum Engineering domain", TrollCreek documentation

[Valaas, I, 2004] "Using NeuFrame to Predict stuck pipe using CI techniques", Project work submitted at Institute of Production Quality and Development NTNU 2004

[Valaas, I., 1997] "Analysis of Digital Drilling Data in Order to Predict Stuck Pipe", ,MSc Thesis submitted to the Faculty of Petroleum Technology and Applied Geophysics.

[Watson, I., 1997] "Applying Case-Based Reasoning: Techniques for Enterprise Systems", The Morgan Kaufmann Series in Artificial Intelligence, Elsevier

[Werbos, P., 1982] "Applications of advances in nonlinear sensitivity analysis in system modeling and optimization" Proc. of the int. federation for information processes, Springer Verlag, pp. 762-770

[Aamodt, A. 1991] "A Knowledge-Intensive, Integrated Approach to Problem Solving and Sustained Learning" A doctoral dissertation submitted to the University of Trondheim, Norwegian Institute of Technology (NTH), Division of Computer Science and Telematics

[Aamodt, A., 1994] "Explanation-Driven Case-Based Reasoning", Topics in case-based reasoning. Springer Verlag, 1994, pp 274-288.

[Aamodt, A 1995] "Knowledge Acquisition and Learning by Experience – The Role of Case – Specific Knowledge, Machine Learning and Knowledge Acquisition: Integrated Approaches", Academic Press Ch. 8, 99 pp. 197-245

[Aamodt, A., 2004] "Knowledge-intensive case-based reasoning in Creek", Advances in case-based reasoning, 7th European Conference, ECCBR 2004, Proceedings. Madrid, Spain, August/September 2004. Lecture Notes in Artificial Intelligence, LNAI 3155, Spinger, 2004. pgs. 1-15

[A. Aamodt, E. Plaza, 1994] "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches" AI Communications. IOS Press, Vol. 7: 1, pp. 39-59.

[Aamodt, A, Nygård 1995] "M. Different roles and mutual dependencies of data, information, and knowledge - an AI perspective on their integration, Data and Knowledge Engineering" 16 191-222, Elsevier

[Aamodt, A., Skalle, P., 2005] Volve - Knowledge Intensive CBR presentation for Norske Hydro

APPENDICES

9.1 Statistically significant MWD variables

In this chapter the variables selected from the available MWD data for the purpose of ANN signal processing is presented. These variables constitute the final set of variables proven to be significant at the 5% level from a Multiple Logistic Regression analysis.
Each parameter is presented with a brief explanation, their perceived impact on the actual problem (here; stuck pipe), measurement techniques and reliability and their measurement unit (exclusively SI units).

9.1.1 Depth [m]

This parameter express the actual length of the wellbore, or the total length of the drill string including the bottom hole assembly from the sea bed and down. This parameter is regarded as correctly measured and have several functions. It is known that a well is more prone to problems as the length of the well increases; it is more exposed to cavings congestion, the time span from when a situation arise downhole until the response reaches the surface increases, the horizontal section of the well increases and so on.

9.1.2 BitMDepth: Bit measured depth [m]

The description for this parameter is equal to the one given for depth, except that this parameter express the total length of the drill string from the sea bed and down independent of the situation. In combination with the depth-parameter one can see if the drill string is "off bottom" or not. In addition this parameter must be considered if one wishes to use "returns depth" in an assessment of e.g. hole cleaning situation.

9.1.3 AvgROP: Average rate of penetration [m/s]

This parameter is included because it expresses direct information about drilling progression and large deviances from predicted progress can reveal problems. Very high rate of penetration over time in combination with low annular speed can cause an unbalance in cuttings generation and removal. This situation referred to as poor hole cleaning is a common reason for stuck pipe.
The parameter is readily measured: Movement in the block pr time unit gives rate of penetration, and should be considered as a correctly registered value. However, noise can

occur which can affect this parameter. E.g. in well 2 at sample 8755 the ROP is 22500 m/s which clearly is erroneous.

9.1.4 AvgHookld: Average hookload [N]

Hookload is the measured block load from the drill string and downhole assembly in combination with possible overload caused by forces from the formation onto the drillstring. At the time even lift speed is obtained and excess force due to acceleration drop to zero, the hookload should not deviate substantially from the relative weight of the drill string in a vertical hole, or calculated values in non vertical sections. In situations where this is not the case, it can be an indication of an abnormal load on the drillstring or downhole assembly. This can be interpreted as a warning about a starting or already occurred problem, for instance stuck pipe.

9.1.5 AvgWOB: Average weight on bit [g]

This value is averaged over the 5 second sample time. In a normal situation, weight on bit increases reversely proportionally to hookload, that is; the less tension in the crown block, the more of the drillstring weight is added to the drillbit. Deviations from the hookload – WOB reversed proportionality might be a problem indicator. WOB is considered an important parameter and is measured reliably.

9.1.6 AvgTorq: Average torque [Nm]

Few of the other parameters, if any, are considered more important when interpreting the drilling situation with a single variable. It is a highly prioritized measuring method, and consequently measured with reliable values.

9.1.7 MaxTorq: Maximum Torque [Nm]

This parameter differs only from average torque in being the maximum reading during the 5 second sample period.

9.1.8 AvgRPM: Average rounds pr. minute [2π /s]

This value simply reflects the speed at which the drillstring revolves and represents an easily measurable, hence a reliable value.

9.1.9 AvgPumpP: Average pump pressure [Pa]

Like most of the pressure measurements this parameter is also measured with an expected margin of error. This value might give a picture of the circulation resistance and therefore act as a problem indicator.

9.1.10 DeltaFlow: Drill volume change [no unit]

In contrast to the other volumetric parameters, this parameter is accepted as reliable because it is measured with a float ball in the mud container. The only source of error for this parameter is the measured depth because the area of the container is final and known.

9.1.11 ReturDepth: Returns depth [m]

This parameter expresses at which depth cavings arriving at the shaker was picked up. This value is derived from the number of pump strokes necessary in order to bring cavings up to surface from a static mud situation and until the first particle reaches the surface. With a given well volume, length of stroke, number of pump cylinders and a given cylinder diameter one can estimate the distance cavings are transported. Obviously the calculated value is vulnerable to a number of possible errors; however the value is regarded as useful in a comparative analysis.

As the deviance between the pickup depth and the depth of the drillbit increases, the hazard for accumulation of debris and consequently hole cleaning problem increases.

9.2 Dataset derived from applicable analytical models applied to relevant, available MWD data

This dataset is a result of expert knowledge about fluid mechanics, fluids and solids characteristics, drilling mechanics, rigsite conditions and established mathematical models combining these. This does not contradict the system prerequisite: Only real-time data (MWD) are used. Other parameters involved are established and acknowledged average values or constants that might be embedded in such an expert system.

9.2.1 Equations involved and variables produced:

Use Stoke to find the particle's vertical speed (the speed at which a particle descends in a given fluid):

Equation 9.1: Stoke's equation

$$Vs \frac{d^2{}_{particle} x(\rho_{partikkel} - \rho_{mud})xg}{18x\mu_{eff}}$$

Where recommended average for $d_{particle}$; 4 mm, and $\rho_{particle}$; 2600 kg/m^3

The ρ_{mud} can be found in the drilling data.

Effective viscosity used in Stoke's: $\mu_{effective} = \dfrac{1488 \times k \times 12^n \times \dfrac{(ID_{csg} - OD_{PIPE})^{(1-n)}}{12}}{\dfrac{12}{32,2} \times V_{annulus}^{(1-n)}}$

where

ID_{csg} is the inner diameter of the casing, and

OD_{pipe} is the diameter of the drill pipe.

Furthermore:

$$n = 3,32 \times \log\left(\frac{600rpm_{measureed}}{300rpm_{measured}}\right) \text{ and } k = \frac{300rpm_{measured}}{(511)^n}$$

The viscosimeter readings are fetched from the daily drilling report.

Equation 9.2: Annular speed

$$V_{ann} = \frac{q_{mud,out}}{\frac{\pi}{4} \times \left(ID_{csg}^2 - OD_{pipe}^2 \right)}$$

Equation 9.3: Transport ratio

$$R_t = -\frac{V_s}{V_{ann}}$$

Horizontality

The larger part of the well that is over 30^o the more exposed the well is to sagging, or in other words depositing particles on the wellbore's lower side.

Suggest the following simple equation for degree of horizontality:

Equation 9.4: Horisontality

$$H_s = \frac{MD - MD_{a > 30^o}}{MD}$$

where MD is Measured Depth. MD over 30^o can be found in drilling data.

Active mud volume and cuttings production

In the following an attempt is made to find a function which yields information about how good the circulation rate is related to cuttings production.
As an indicator of the quality of the circulation the volume of the returned mud is used, because this volume yields information about how much of the active mud volume supports hole cleaning.

Equation 9.5: Cuttings generation

$$KG = ROP \times \frac{\pi}{4} \times \left(D_{BIT} \right)^2$$

In order to obtain values between 0 and 1, the following is done to get the interaction effect with the variable above:
Cuttings generation and active mud ratio,

Equation 9.6: Cuttings - active mud ratio

$$K_s = \frac{1 - KG}{100 \times `MudIn}$$

where ROP and MudIn can be found in the drilling data.

String Rotation

A vital reason for cuttings accumulation and consequently unfavourable hole cleaning is situations where the string is static or when just the drill crown is moving (as is the situation while directional drilling). The hole cleaning is presumed close to zero in situations where there is a long horizontal –or high deviation section and the rotation of the drill string seizes. In order to be able to describe this fact the experience of drilling personnel have supplied the following heuristic rules, summarized as pseudo- code:

```
if (rpm>=120)
   rotf = 1;
   if (rpm < 120 && rpm >=90)
      rotf = 0.8;
      if (rpm < 90 && rpm >=60)
         rotf 0.5;
         if (rpm < 60 && rpm >=20)
            rotf = 0.25;
            if (rpm < 20)
               rotf = 0.1;
```

Return depth for drill cuttings

If the cleaning situation is good it will be less deviation between the depth from where the cuttings are picked up and the measured depth. A formula which gives a rough indication of the hole cleaning is:

Equation 9.7: Hole cleaning

$$HR = \frac{RETURdepth}{BitMDepth}$$

9.2.2 Resulting dataset

The analytical pre- processing of the MWD- variables produced a data file as shown in Table 7 1, "Rotf" denotes the evaluated heuristic rule- set, "HS" Horizontality from Equation 7 8, "Vann" annular speed from Equation 7 9, "Vs" vertical speed from Equation 7 10, "Rt" transport ratio from Equation 7 11, "KG" cuttings generation from Equation 7 12 and finally "HR" hole cleaning from Equation 7 13.

Table 9.1: Analytically pre- processed MWD data

WellID	Rotf	HS	Vann[ft/s]	Vs	Rt	KG	HR	Stuck
1	1	0.747569	0.766728	0.064652	0.747034	0	1	0
1	1	0.747569	0.766728	0.064652	0.747034	0	1	0
1	1	0.747569	0.766728	0.064652	0.747034	0	1	0
1	1	0.747569	0.766728	0.064652	0.747034	0	1	0
1	1	0.747569	0.766728	0.064652	0.747034	0	1	0
1	1	0.747569	0.770701	0.064706	0.748128	0	1	0
...	...	...	...	...	...	...	...	...

9.3 Mechanisms causing the unwanted event Stuck Pipe

Drilling an offshore well is a highly complex operation having multiple, often inter-correlated sources of problems. Below is a summarization of the most common mechanisms of which stuck pipe is the result.

9.3.1 Geometrically dependant stuck mechanisms

One category of these problem sources has their root cause in a single or a combination of geometrically challenging lithology and/or disadvantageously shaped wellbores

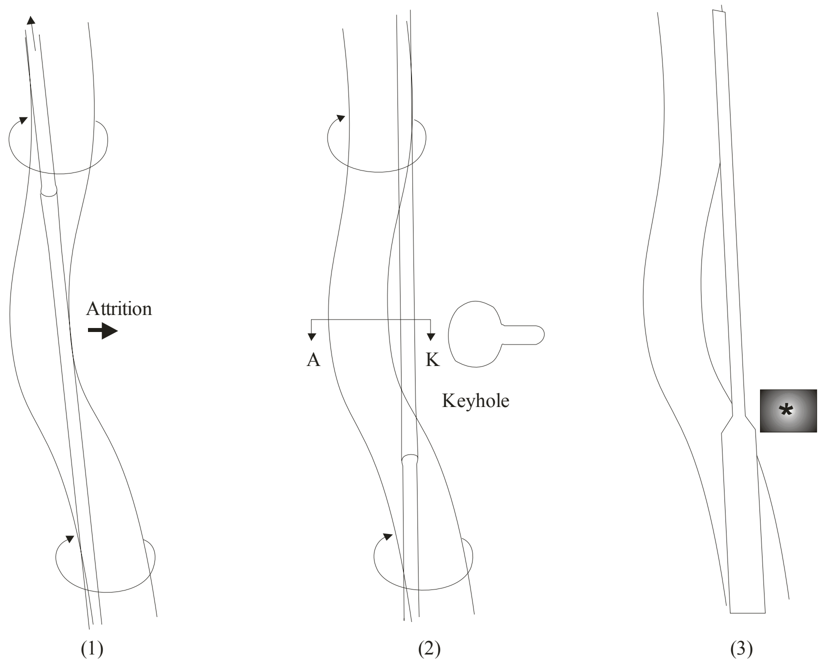


Figure 9.1: Key seating stuck pipe mechanism

Key Seating might occur after having passed a major dogleg[13] (1) where drillpipe and coupling have worn into the formation over some period of time (2). There are no single parameter that might give an indication of this problem prior to pulling out of hole and the BHA (bottom hole assembly) reaches the key seat (3)

[13] dogleg meaning a curvature or deviation from a vertical line built up over a short vertical distance. Informally, the degree of dogleg severity is proportional to the deviation build up and reversed proportional to the build up vertical distance

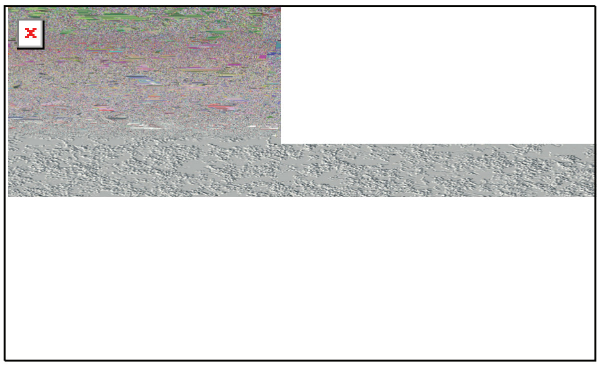
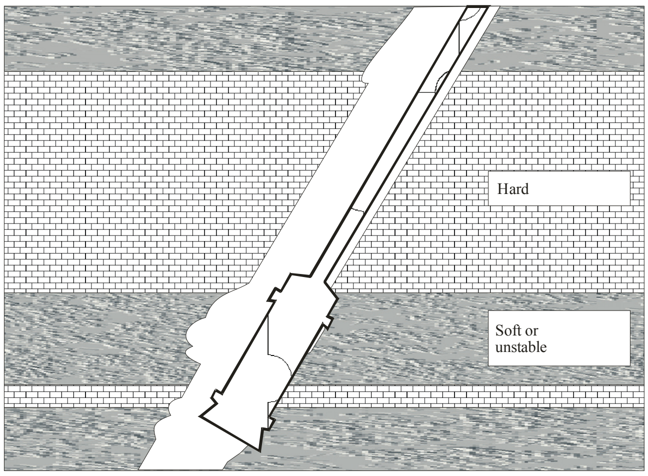


Figure 9.3: **Undergauged hole when changing worn drillbit**

Undergauged Hole occurs when the drillbit outer part wear down and consequently there becomes a deviation from actual and nominal hole diameter. The problem only occurs if the entire section cannot be drilled using one drillbit. When running in hole with a drillbit gauged to nominal diameter, this will be larger than the replaced drillbit.

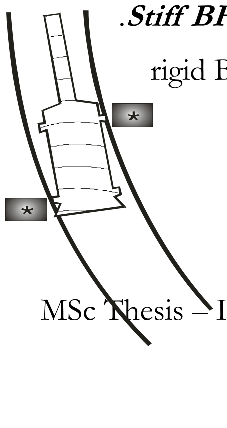
No single or comprehensible set of parameters give any indication to a potential problem until it occurs.

Ledges occur at alternating hard and soft layers in the formation Drilling in soft formation less resistant to mechanical wear might lead to traps when leaving hard formations less susceptible to mechanical strain. This stuck pipe mechanism might strike bi- directional; both when running in and pulling out of hole.



9-3: Ledges caused by alternating formation hardness

Of the available MWD data there are possibilities for experienced rig personnel to monitor torque curves to reveal varying resistance, thus finding varying hardness in the formation implicitly giving a warning of a possible unwanted event.



.***Stiff BHA combined with severe doglegs*** might lead to stuck pipe due to a rigid BHA not being able to follow the trajectory lead by such a dogleg.

Figure 9.5: **Stiff BHA trapped in severe doglegs**

Deforming salts and formation influx

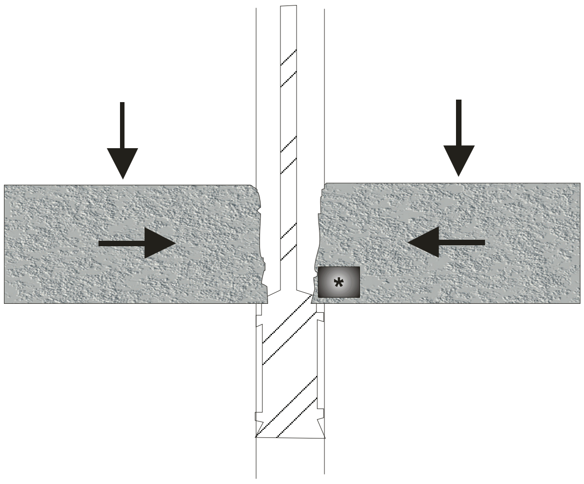


Figure 9.6: Salts and formation influx

9.3.2 Solids and hole cleaning problems

Thick mudcake

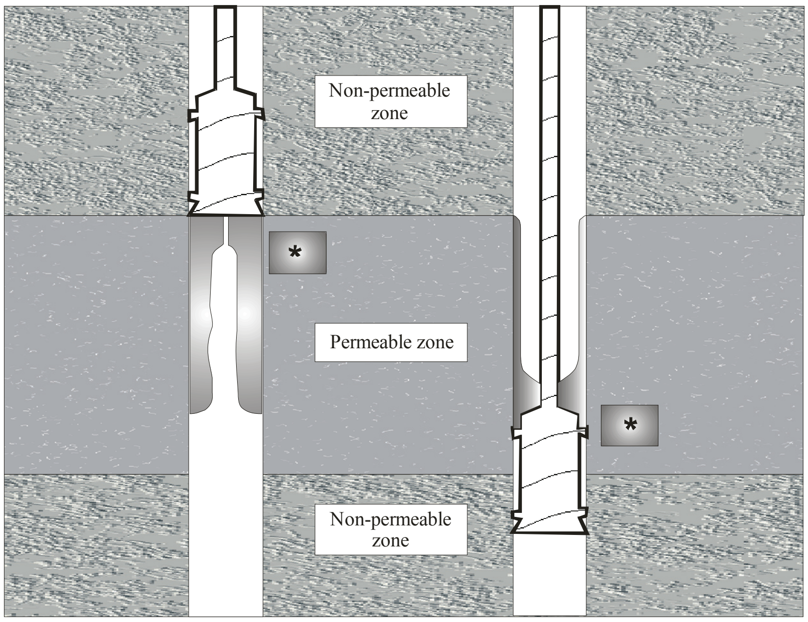


Figure 9.7: Permeable formation and thick mudcake

Fractured formations test

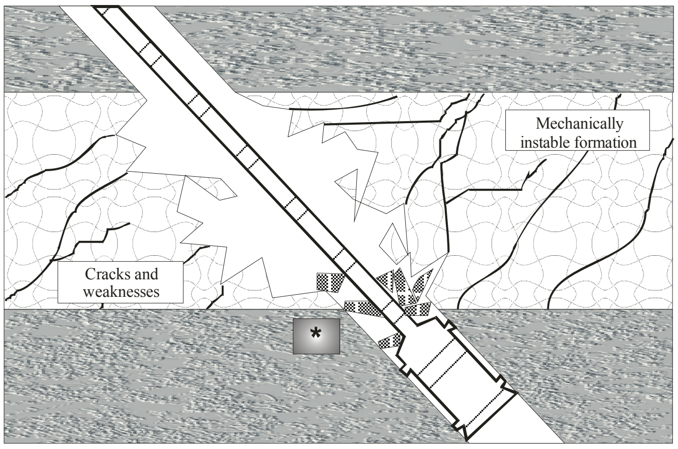


Figure 9.8: Fractured formations and debris accumulation

Junk test

Differential sticking test

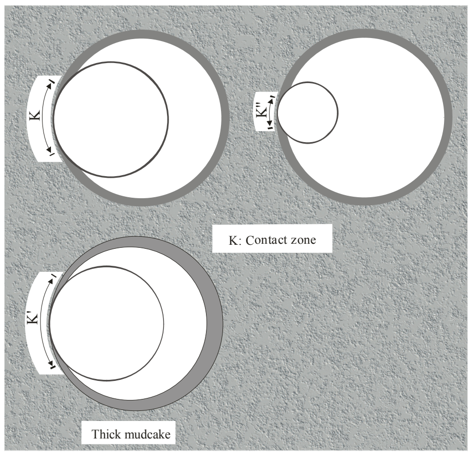


Figure 9.9: Differential sticking and immobile drillstring

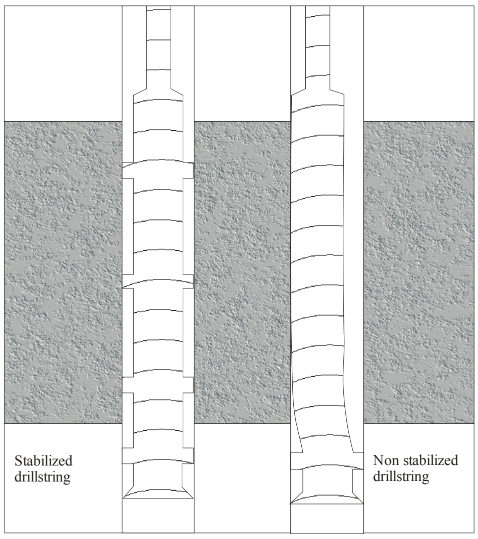


Figure 9.10: Consequences of non- stabilized drillstrings

9.4 GUI

As it still is at the prototype level, there is no functional link between this GUI and the e-ANN package, but only small implementation- steps remain. The main purpose of attaching this GUI is providing a graphic illustration over the main functionality of the e-ANN.

The prototype has been prepared for training only, and the main frame, in which all needed information is entered, is shown below:

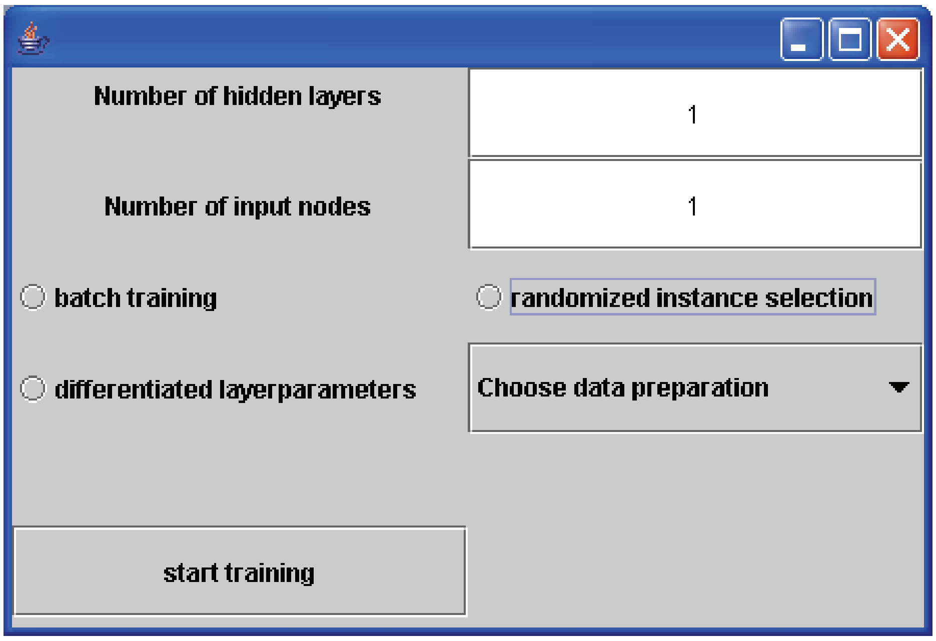
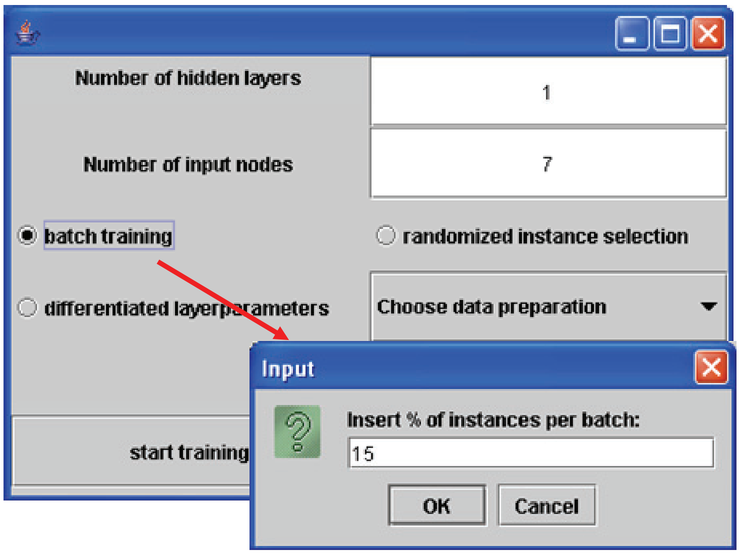


Figure 9.11: Main menu of the e-ANN GUI.

First the number of hidden layers is entered as this sets the dimension of the input matrix and instantiate the number of layers in the ANN BatchTraining class. As the input nodes have no processing purposes, this number is entered in the main frame.

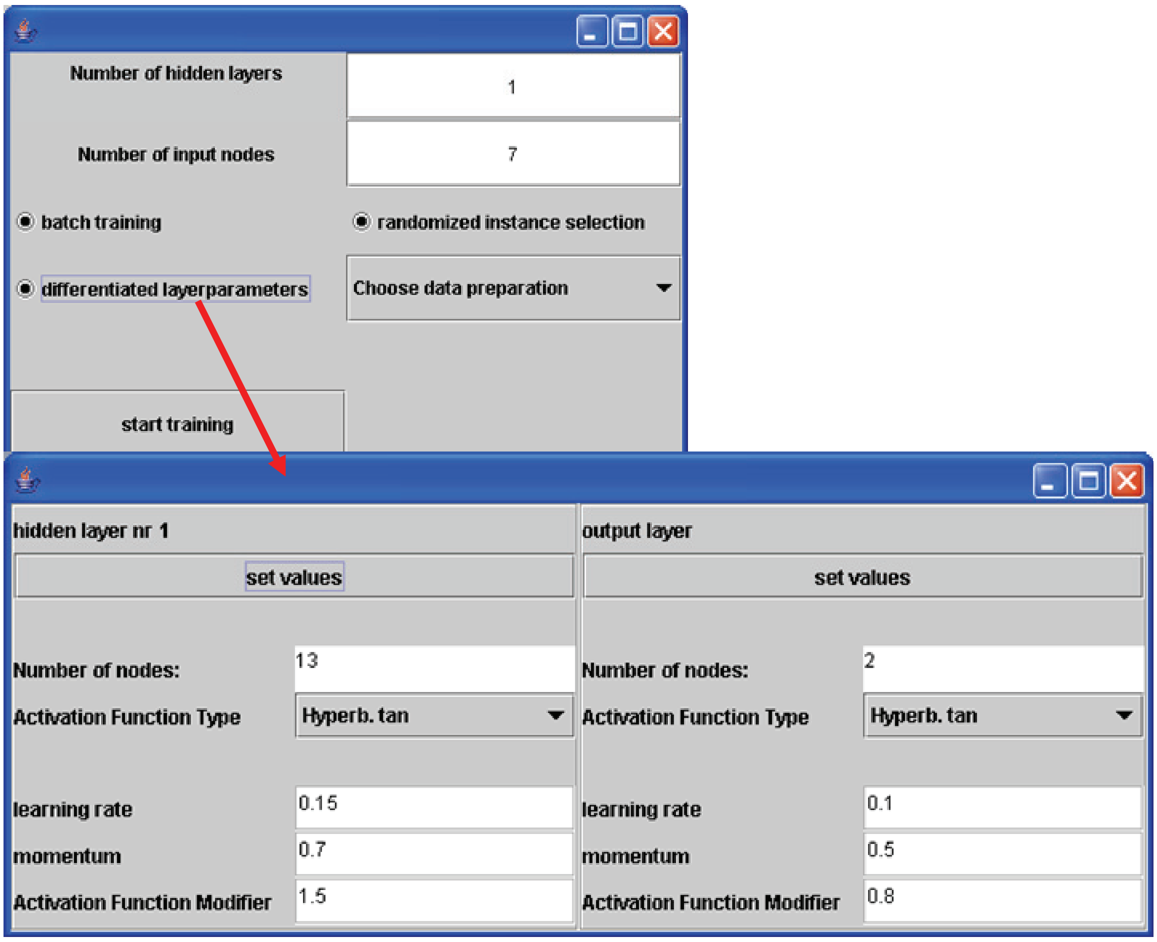
Secondly, training choices are made. Choosing batch training triggers an input box asking for percentage of instances per batch (Figure 9.12):


Number of hidden layers: 1
Number of input nodes: 7
● batch training
○ randomized instance selection
○ differentiated layerparameters
Choose data preparation
start training

Input
Insert % of instances per batch:
15

Figure 9.12: GUI - batch training

Figure 9.13 show the tuning possibilities implemented in the e-ANN:


Number of hidden layers: 1
Number of input nodes: 7
● batch training
● randomized instance selection
● differentiated layerparameters
Choose data preparation
start training

hidden layer nr 1		output layer	
set values		set values	
Number of nodes:	13	Number of nodes:	2
Activation Function Type	Hyperb. tan	Activation Function Type	Hyperb. tan
learning rate	0.15	learning rate	0.1
momentum	0.7	momentum	0.5
Activation Function Modifier	1.5	Activation Function Modifier	0.8

Figure 9.13: GUI – ANN layer specific layer tuning

In this example, the user designs a net having 7 input nodes, 1 hidden layer of 13 nodes and two output nodes giving the 7 x 13 x 2 topology: shown in Figure 9.14

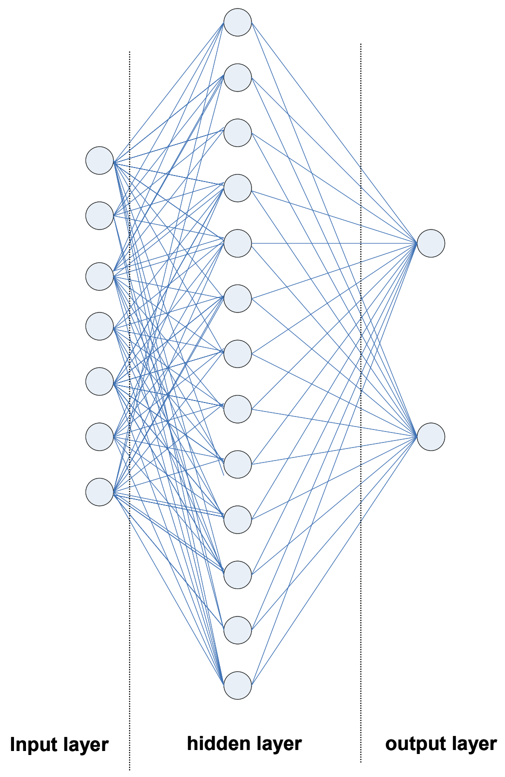


Figure 9.14: Resulting ANN topology

Moreover, the choices made in the "Layer parameters" box lead to the following differentiated tuning parameters:

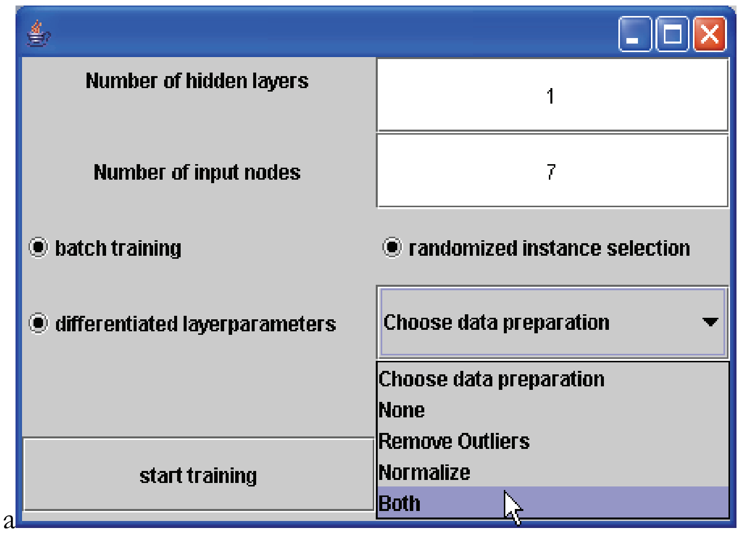
Hidden layer 1:

- Momentum = 0.7
- Learning rate = 0.15
- hyperbolic tangent activation function
- activation function "flatness": 1.5

Output layer:

- Momentum = 0.5
- Learning rate = 0.1
- sigmoid activation function
- activation function "flatness": 0.8

The last training parameter is whether to use sequential or random selection of instances, in Figure 9.15 randomized selection was chosen:

a

Number of hidden layers	1
Number of input nodes	7
● batch training	● randomized instance selection
● differentiated layerparameters	Choose data preparation

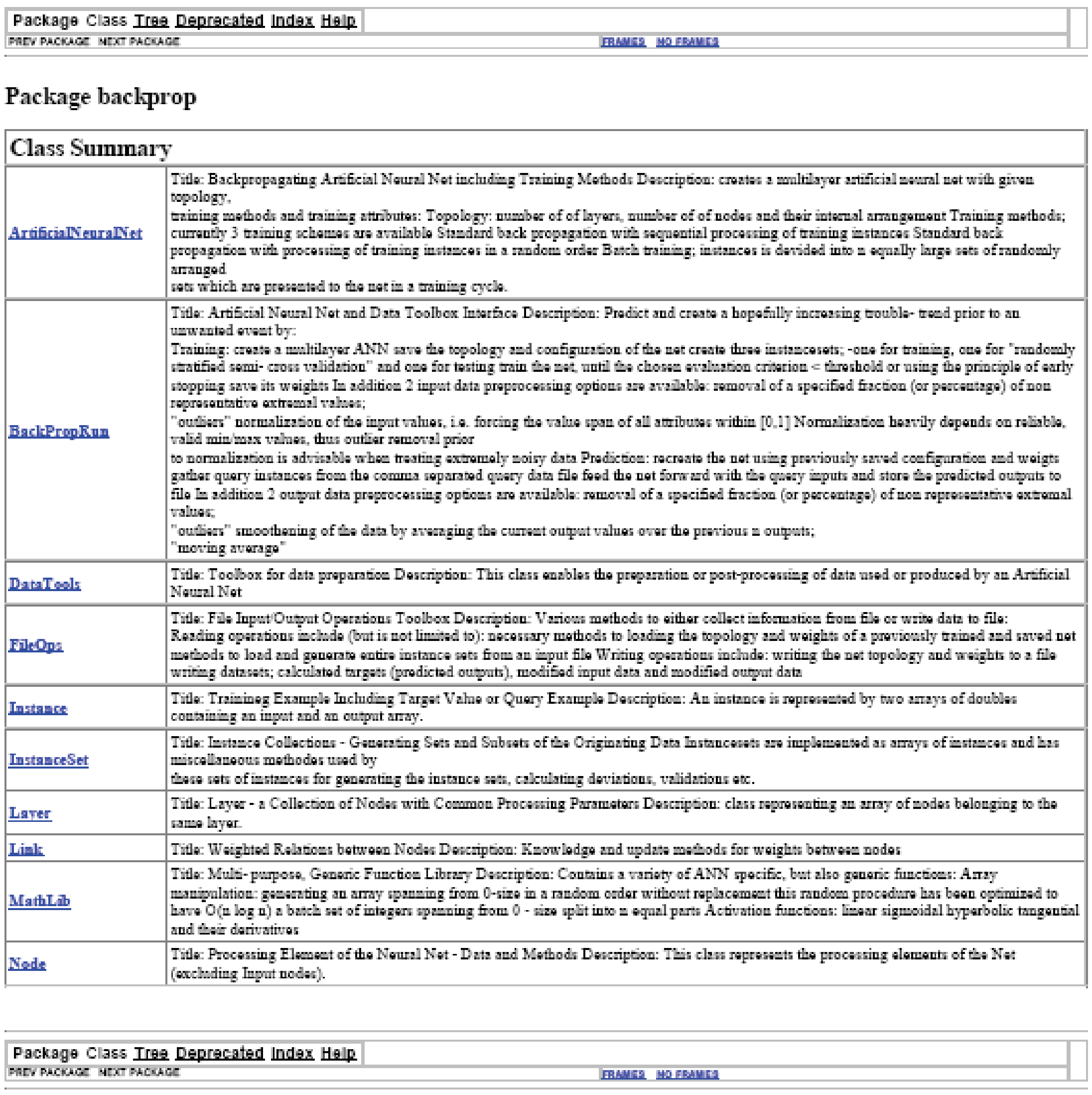
Choose data preparation
None
Remove Outliers
Normalize
Both

Figure 9.15: GUI - customizing the training algorithm

This figure also show selection of data pre- processing: Here both outlier removal and normalization has been chosen.

9.5 ANN and data processing code documentation

Following the Java programming JavaDoc code documentation standard this chapter describes all classes, methods and parameters implemented in the "Predict" phase of the PEARL cycle.

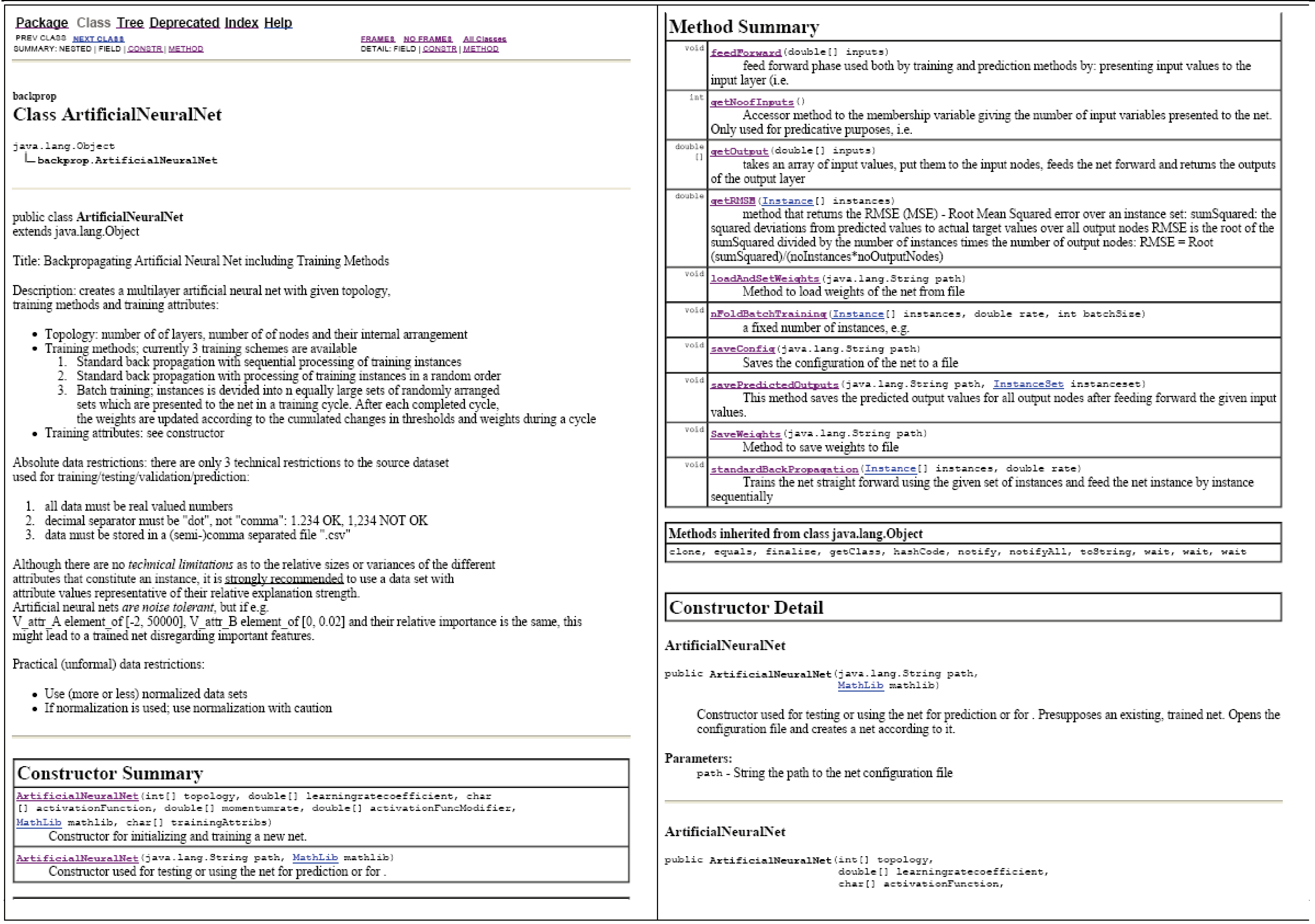


Package Class Tree Deprecated Index Help
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package backprop

Class Summary	
ArtificialNeuralNet	Title: Backpropagating Artificial Neural Net including Training Methods Description: creates a multilayer artificial neural net with given topology, training methods and training attributes: Topology: number of of layers, number of of nodes and their internal arrangement Training methods; currently 3 training schemes are available Standard back propagation with sequential processing of training instances Standard back propagation with processing of training instances in a random order Batch training; instances is devided into n equally large sets of randomly arranged sets which are presented to the net in a training cycle.
BackPropRun	Title: Artificial Neural Net and Data Toolbox Interface Description: Predict and create a hopefully increasing trouble- trend prior to an unwanted event by: Training: create a multilayer ANN save the topology and configuration of the net create three instancesets; -one for training, one for "randomly stratified semi- cross validation" and one for testing train the net, until the chosen evaluation criterion < threshold or using the principle of early stopping save its weights In addition 2 input data preprocessing options are available: removal of a specified fraction (or percentage) of non representative extremal values; "outliers" normalization of the input values, i.e. forcing the value span of all attributes within [0,1] Normalization heavily depends on reliable, valid min/max values, thus outlier removal prior to normalization is advisable when treating extremely noisy data Prediction: recreate the net using previously saved configuration and weigts gather query instances from the comma separated query data file feed the net forward with the query inputs and store the predicted outputs to file In addition 2 output data preprocessing options are available: removal of a specified fraction (or percentage) of non representative extremal values; "outliers" smoothening of the data by averaging the current output values over the previous n outputs; "moving average"
DataTools	Title: Toolbox for data preparation Description: This class enables the preparation or post-processing of data used or produced by an Artificial Neural Net
FileOps	Title: File Input/Output Operations Toolbox Description: Various methods to either collect information from file or write data to file: Reading operations include (but is not limited to): necessary methods to loading the topology and weights of a previously trained and saved net methods to load and generate entire instance sets from an input file Writing operations include: writing the net topology and weights to a file writing datasets; calculated targets (predicted outputs), modified input data and modified output data
Instance	Title: Trainineg Example Including Target Value or Query Example Description: An instance is represented by two arrays of doubles containing an input and an output array.
InstanceSet	Title: Instance Collections - Generating Sets and Subsets of the Originating Data Instancesets are implemented as arrays of instances and has miscellaneous methodes used by these sets of instances for generating the instance sets, calculating deviations, validations etc.
Layer	Title: Layer - a Collection of Nodes with Common Processing Parameters Description: class representing an array of nodes belonging to the same layer.
Link	Title: Weighted Relations between Nodes Description: Knowledge and update methods for weights between nodes
MathLib	Title: Multi- purpose, Generic Function Library Description: Contains a variety of ANN specific, but also generic functions: Array manipulation: generating an array spanning from 0-size in a random order without replacement this random procedure has been optimized to have O(n log n) a batch set of integers spanning from 0 - size split into n equal parts Activation functions: linear sigmoidal hyperbolic tangential and their derivatives
Node	Title: Processing Element of the Neural Net - Data and Methods Description: This class represents the processing elements of the Net (excluding Input nodes).

Package Class Tree Deprecated Index Help
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES



Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS | FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

backprop
Class ArtificialNeuralNet

```
java.lang.Object
  └backprop.ArtificialNeuralNet
```

public class **ArtificialNeuralNet**
extends java.lang.Object

Title: Backpropagating Artificial Neural Net including Training Methods

Description: creates a multilayer artificial neural net with given topology, training methods and training attributes:

- Topology: number of of layers, number of of nodes and their internal arrangement
- Training methods; currently 3 training schemes are available
  1. Standard back propagation with sequential processing of training instances
  2. Standard back propagation with processing of training instances in a random order
  3. Batch training; instances is devided into n equally large sets of randomly arranged sets which are presented to the net in a training cycle. After each completed cycle, the weights are updated according to the cumulated changes in thresholds and weights during a cycle
- Training attributes: see constructor

Absolute data restrictions: there are only 3 technical restrictions to the source dataset used for training/testing/validation/prediction:

1. all data must be real valued numbers
2. decimal separator must be "dot", not "comma": 1.234 OK, 1,234 NOT OK
3. data must be stored in a (semi-)comma separated file ".csv"

Although there are no *technical limitations* as to the relative sizes or variances of the different attributes that constitute an instance, it is strongly recommended to use a data set with attribute values representative of their relative explanation strength.
Artificial neural nets *are noise tolerant*, but if e.g.
V_attr_A element_of [-2, 50000], V_attr_B element_of [0, 0.02] and their relative importance is the same, this might lead to a trained net disregarding important features.

Practical (unformal) data restrictions:

- Use (more or less) normalized data sets
- If normalization is used; use normalization with caution

Constructor Summary

`ArtificialNeuralNet(int[] topology, double[] learningratecoefficient, char[] activationFunction, double[] momentumrate, double[] activationFuncModifier, MathLib mathlib, char[] trainingAttribs)` Constructor for initializing and training a new net.
`ArtificialNeuralNet(java.lang.String path, MathLib mathlib)` Constructor used for testing or using the net for prediction or for .

Method Summary

`void`	`feedForward(double[] inputs)` feed forward phase used both by training and prediction methods by: presenting input values to the input layer (i.e.
`int`	`getNoofInputs()` Accessor method to the membership variable giving the number of input variables presented to the net. Only used for predicative purposes, i.e.
`double[]`	`getOutput(double[] inputs)` takes an array of input values, put them to the input nodes, feeds the net forward and returns the outputs of the output layer
`double`	`getRMSE(Instance[] instances)` method that returns the RMSE (MSE) - Root Mean Squared error over an instance set: sumSquared: the squared deviations from predicted values to actual target values over all output nodes RMSE is the root of the sumSquared divided by the number of instances times the number of output nodes: RMSE = Root (sumSquared)/(noInstances*noOutputNodes)
`void`	`loadAndSetWeights(java.lang.String path)` Method to load weights of the net from file
`void`	`nFoldBatchTraining(Instance[] instances, double rate, int batchSize)` a fixed number of instances, e.g.
`void`	`saveConfig(java.lang.String path)` Saves the configuration of the net to a file
`void`	`savePredictedOutputs(java.lang.String path, InstanceSet instanceset)` This method saves the predicted output values for all output nodes after feeding forward the given input values.
`void`	`SaveWeights(java.lang.String path)` Method to save weights to file
`void`	`standardBackPropagation(Instance[] instances, double rate)` Trains the net straight forward using the given set of instances and feed the net instance by instance sequentially

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

ArtificialNeuralNet

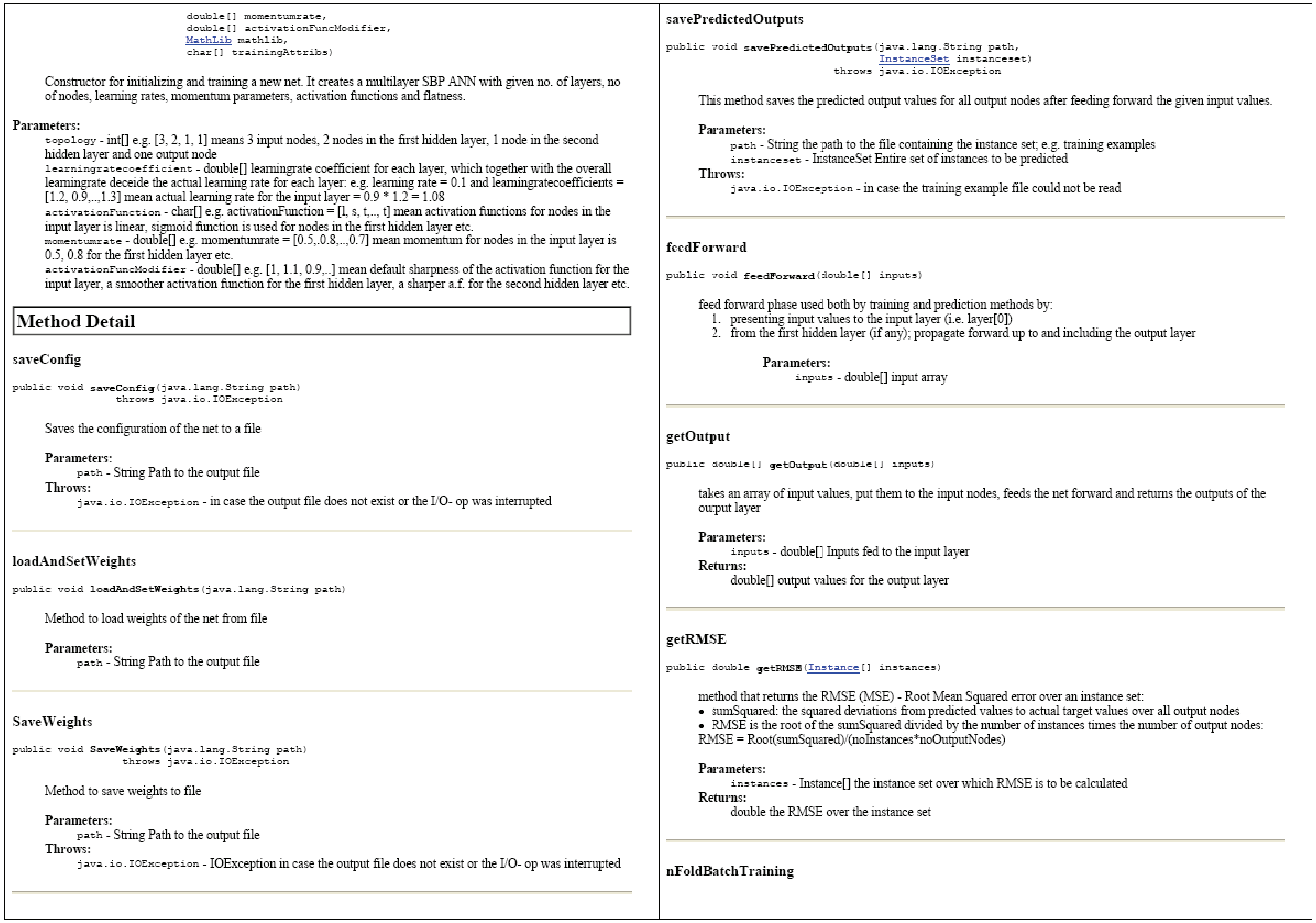
```
public ArtificialNeuralNet(java.lang.String path,
                           MathLib mathlib)
```

Constructor used for testing or using the net for prediction or for . Presupposes an existing, trained net. Opens the configuration file and creates a net according to it.

Parameters:
`path` - String the path to the net configuration file

ArtificialNeuralNet

```
public ArtificialNeuralNet(int[] topology,
                           double[] learningratecoefficient,
                           char[] activationFunction,
```



```
                                   double[] momentumrate,
                                   double[] activationFuncModifier,
                                   MathLib mathlib,
                                   char[] trainingAttribs)
```

Constructor for initializing and training a new net. It creates a multilayer SBP ANN with given no. of layers, no of nodes, learning rates, momentum parameters, activation functions and flatness.

Parameters:
`topology` - int[] e.g. [3, 2, 1, 1] means 3 input nodes, 2 nodes in the first hidden layer, 1 node in the second hidden layer and one output node
`learningratecoefficient` - double[] learningrate coefficient for each layer, which together with the overall learningrate deceide the actual learning rate for each layer: e.g. learning rate = 0.1 and learningratecoefficients = [1.2, 0.9,..,1.3] mean actual learning rate for the input layer = 0.9 * 1.2 = 1.08
`activationFunction` - char[] e.g. activationFunction = [l, s, t,.., t] mean activation functions for nodes in the input layer is linear, sigmoid function is used for nodes in the first hidden layer etc.
`momentumrate` - double[] e.g. momentumrate = [0.5,.0.8,..,0.7] mean momentum for nodes in the input layer is 0.5, 0.8 for the first hidden layer etc.
`activationFuncModifier` - double[] e.g. [1, 1.1, 0.9,..] mean default sharpness of the activation function for the input layer, a smoother activation function for the first hidden layer, a sharper a.f. for the second hidden layer etc.

Method Detail

saveConfig

```
public void saveConfig(java.lang.String path)
                throws java.io.IOException
```

Saves the configuration of the net to a file

Parameters:
`path` - String Path to the output file
Throws:
`java.io.IOException` - in case the output file does not exist or the I/O- op was interrupted

loadAndSetWeights

```
public void loadAndSetWeights(java.lang.String path)
```

Method to load weights of the net from file

Parameters:
`path` - String Path to the output file

SaveWeights

```
public void SaveWeights(java.lang.String path)
                 throws java.io.IOException
```

Method to save weights to file

Parameters:
`path` - String Path to the output file
Throws:
`java.io.IOException` - IOException in case the output file does not exist or the I/O- op was interrupted

savePredictedOutputs

```
public void savePredictedOutputs(java.lang.String path,
                                 InstanceSet instanceset)
                          throws java.io.IOException
```

This method saves the predicted output values for all output nodes after feeding forward the given input values.

Parameters:
`path` - String the path to the file containing the instance set; e.g. training examples
`instanceset` - InstanceSet Entire set of instances to be predicted
Throws:
`java.io.IOException` - in case the training example file could not be read

feedForward

```
public void feedForward(double[] inputs)
```

feed forward phase used both by training and prediction methods by:
1. presenting input values to the input layer (i.e. layer[0])
2. from the first hidden layer (if any); propagate forward up to and including the output layer

Parameters:
`inputs` - double[] input array

getOutput

```
public double[] getOutput(double[] inputs)
```

takes an array of input values, put them to the input nodes, feeds the net forward and returns the outputs of the output layer

Parameters:
`inputs` - double[] Inputs fed to the input layer
Returns:
double[] output values for the output layer

getRMSE

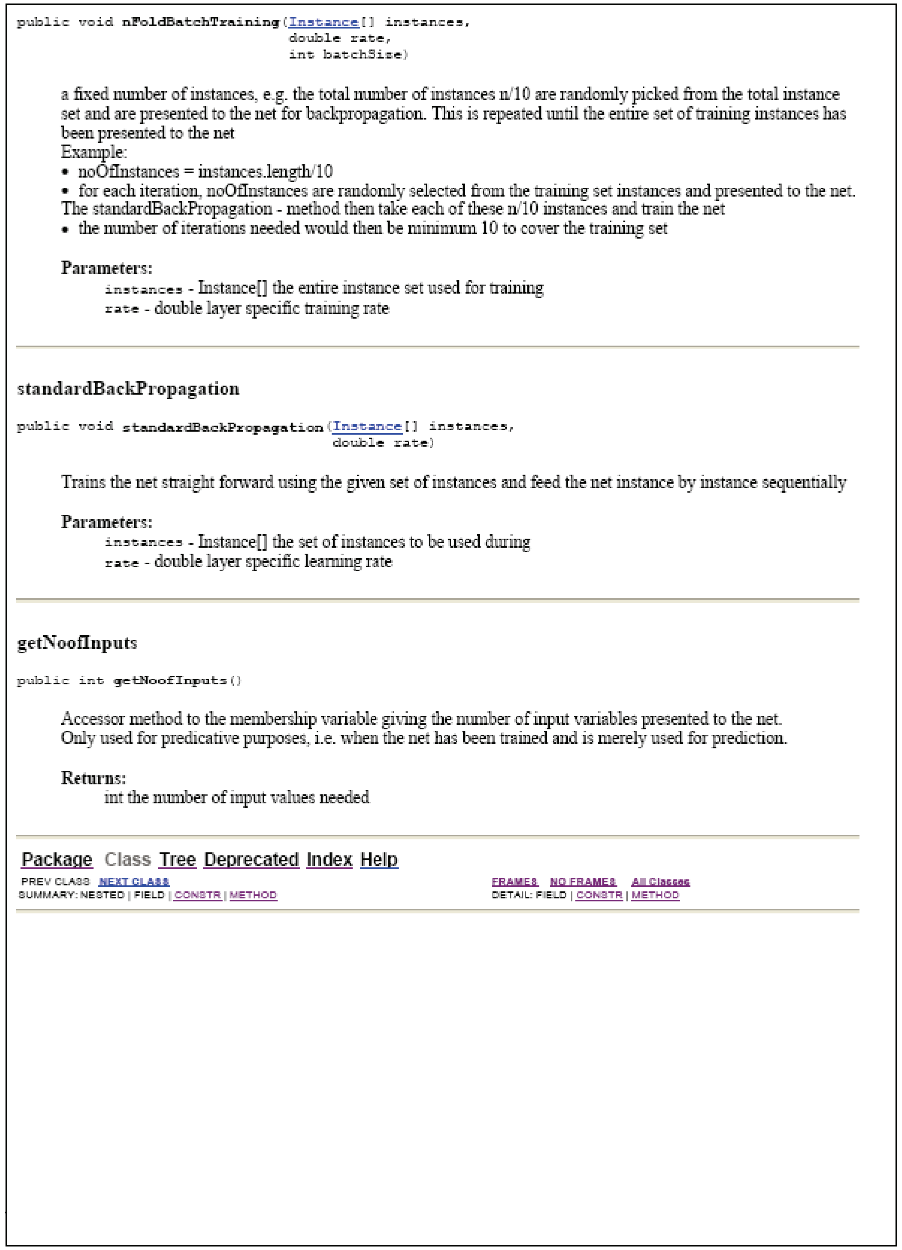
```
public double getRMSE(Instance[] instances)
```

method that returns the RMSE (MSE) - Root Mean Squared error over an instance set:
- sumSquared: the squared deviations from predicted values to actual target values over all output nodes
- RMSE is the root of the sumSquared divided by the number of instances times the number of output nodes:

RMSE = Root(sumSquared)/(noInstances*noOutputNodes)

Parameters:
`instances` - Instance[] the instance set over which RMSE is to be calculated
Returns:
double the RMSE over the instance set

nFoldBatchTraining



```
public void nFoldBatchTraining(Instance[] instances,
                               double rate,
                               int batchSize)
```

a fixed number of instances, e.g. the total number of instances n/10 are randomly picked from the total instance set and are presented to the net for backpropagation. This is repeated until the entire set of training instances has been presented to the net
Example:
- noOfInstances = instances.length/10
- for each iteration, noOfInstances are randomly selected from the training set instances and presented to the net.

The standardBackPropagation - method then take each of these n/10 instances and train the net
- the number of iterations needed would then be minimum 10 to cover the training set

Parameters:
`instances` - Instance[] the entire instance set used for training
`rate` - double layer specific training rate

standardBackPropagation

```
public void standardBackPropagation(Instance[] instances,
                                    double rate)
```

Trains the net straight forward using the given set of instances and feed the net instance by instance sequentially

Parameters:
`instances` - Instance[] the set of instances to be used during
`rate` - double layer specific learning rate

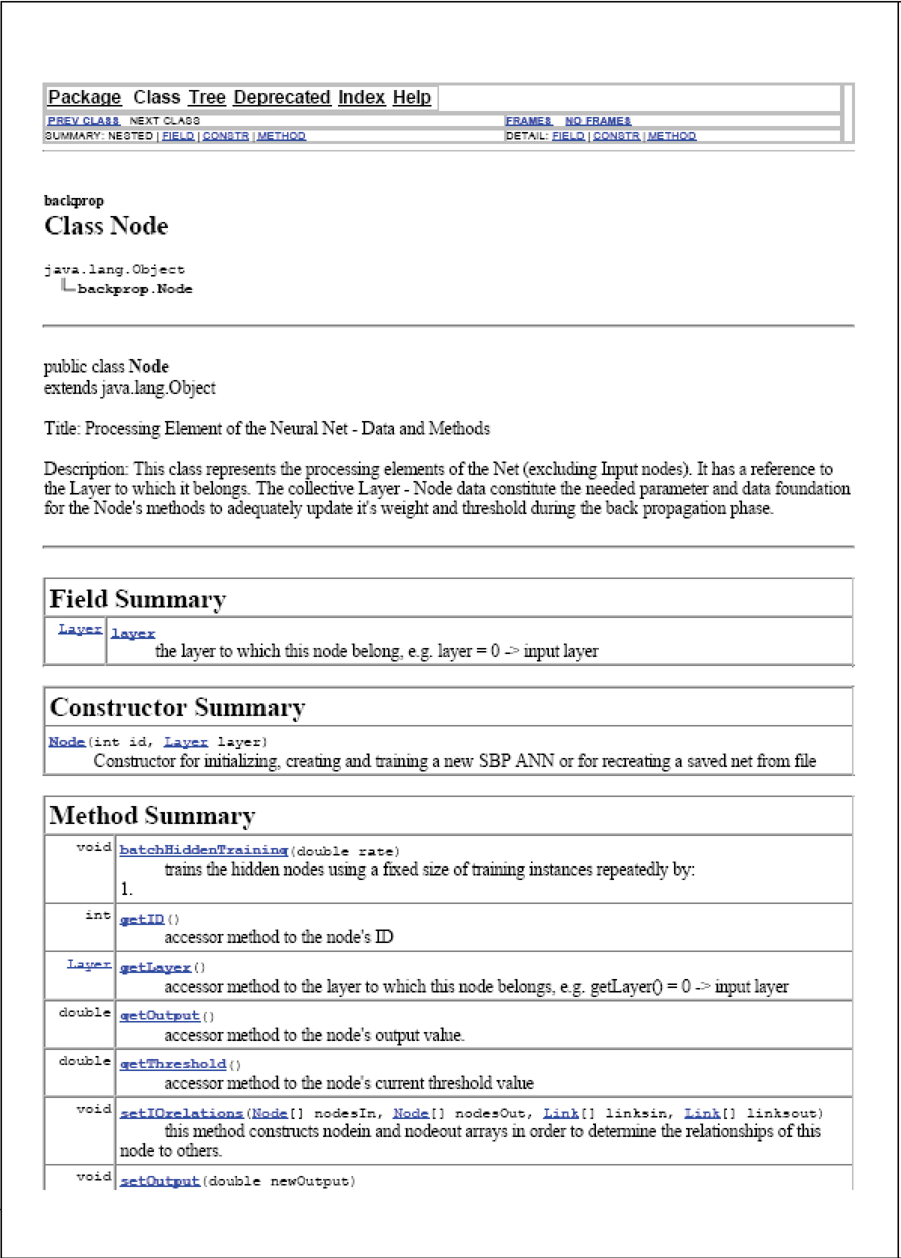
getNoofInputs

```
public int getNoofInputs()
```

Accessor method to the membership variable giving the number of input variables presented to the net.
Only used for predicative purposes, i.e. when the net has been trained and is merely used for prediction.

Returns:
int the number of input values needed

Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD
FRAMES NO FRAMES All Classes
DETAIL: FIELD | CONSTR | METHOD



Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS | FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

backprop
Class Node

```
java.lang.Object
  └backprop.Node
```

public class **Node**
extends java.lang.Object

Title: Processing Element of the Neural Net - Data and Methods

Description: This class represents the processing elements of the Net (excluding Input nodes). It has a reference to the Layer to which it belongs. The collective Layer - Node data constitute the needed parameter and data foundation for the Node's methods to adequately update it's weight and threshold during the back propagation phase.

Field Summary

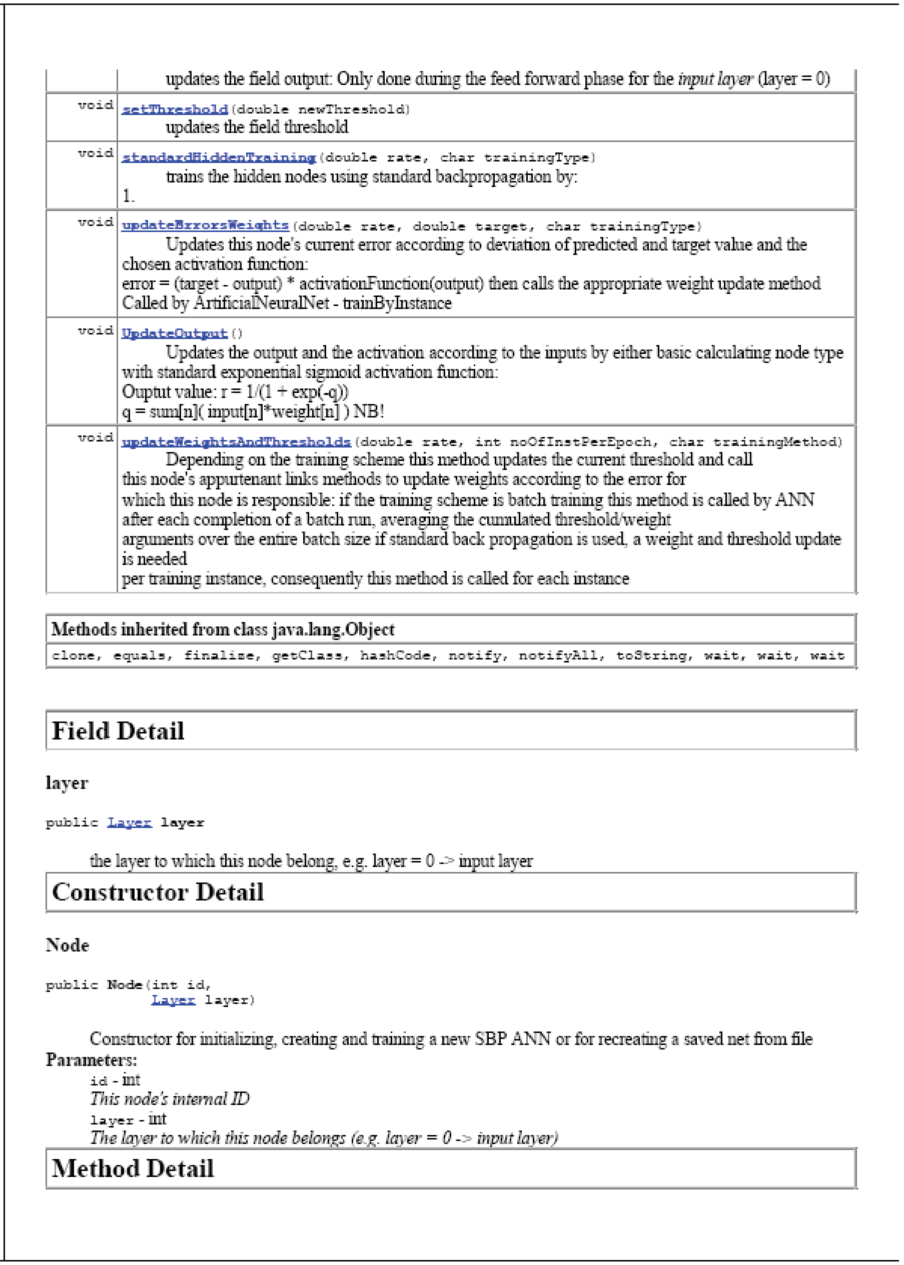
`Layer`	`layer` the layer to which this node belong, e.g. layer = 0 -> input layer

Constructor Summary

`Node(int id, Layer layer)`
Constructor for initializing, creating and training a new SBP ANN or for recreating a saved net from file

Method Summary

`void`	`batchHiddenTraining(double rate)` trains the hidden nodes using a fixed size of training instances repeatedly by: 1.
`int`	`getID()` accessor method to the node's ID
`Layer`	`getLayer()` accessor method to the layer to which this node belongs, e.g. getLayer() = 0 -> input layer
`double`	`getOutput()` accessor method to the node's output value.
`double`	`getThreshold()` accessor method to the node's current threshold value
`void`	`setIOrelations(Node[] nodesIn, Node[] nodesOut, Link[] linksin, Link[] linksout)` this method constructs nodein and nodeout arrays in order to determine the relationships of this node to others.
`void`	`setOutput(double newOutput)` updates the field output: Only done during the feed forward phase for the *input layer* (layer = 0)
`void`	`setThreshold(double newThreshold)` updates the field threshold
`void`	`standardHiddenTraining(double rate, char trainingType)` trains the hidden nodes using standard backpropagation by: 1.
`void`	`updateErrorsWeights(double rate, double target, char trainingType)` Updates this node's current error according to deviation of predicted and target value and the chosen activation function: error = (target - output) * activationFunction(output) then calls the appropriate weight update method Called by ArtificialNeuralNet - trainByInstance
`void`	`UpdateOutput()` Updates the output and the activation according to the inputs by either basic calculating node type with standard exponential sigmoid activation function: Output value: r = 1/(1 + exp(-q)) q = sum[n](input[n]*weight[n]) NB!
`void`	`updateWeightsAndThresholds(double rate, int noOfInstPerEpoch, char trainingMethod)` Depending on the training scheme this method updates the current threshold and call this node's appurtenant links methods to update weights according to the error for which this node is responsible: if the training scheme is batch training this method is called by ANN after each completion of a batch run, averaging the cumulated threshold/weight arguments over the entire batch size if standard back propagation is used, a weight and threshold update is needed per training instance, consequently this method is called for each instance



Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

layer

```
public Layer layer
```

the layer to which this node belong, e.g. layer = 0 -> input layer

Constructor Detail

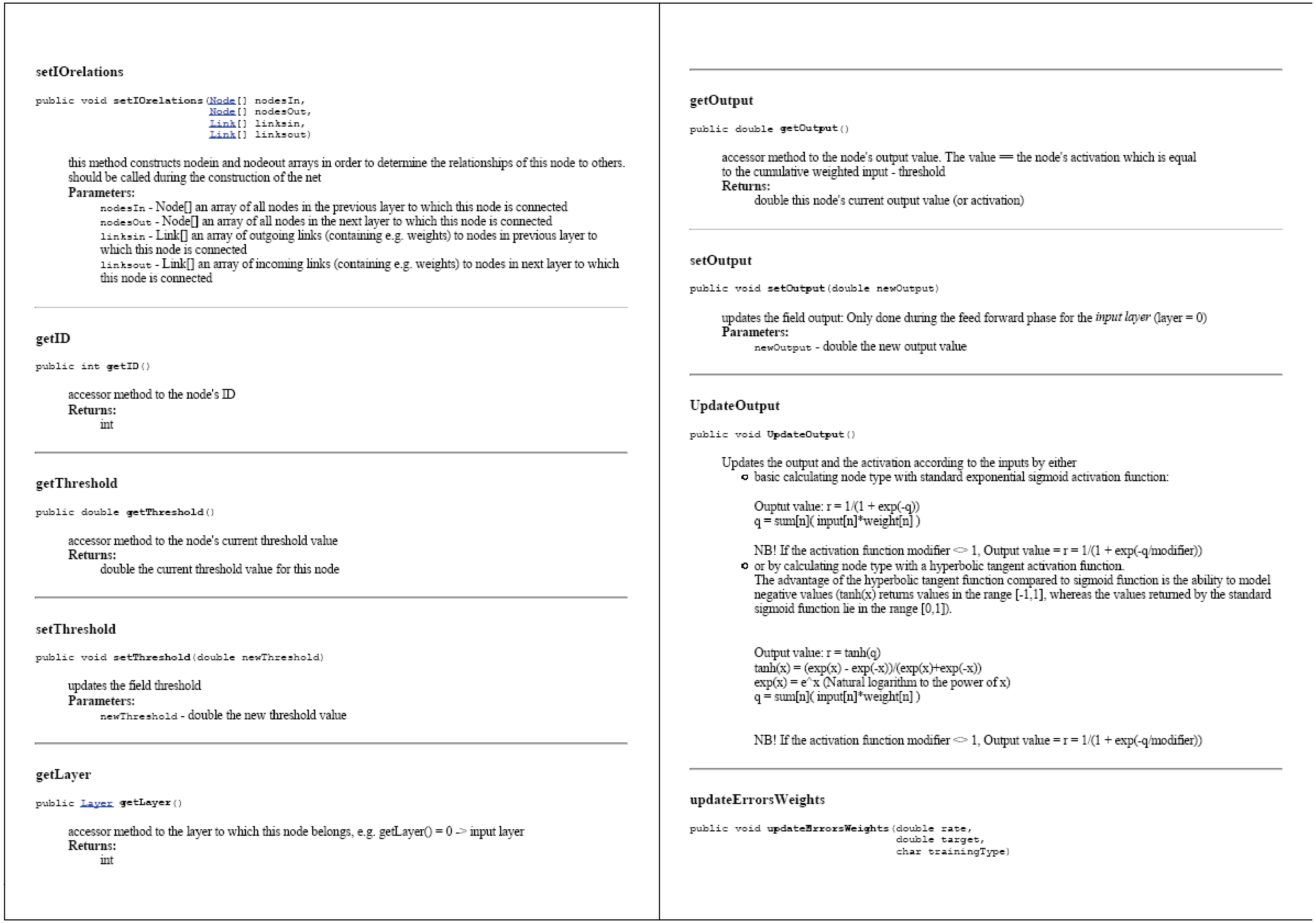
Node

```
public Node(int id,
            Layer layer)
```

Constructor for initializing, creating and training a new SBP ANN or for recreating a saved net from file

Parameters:
`id` - int
This node's internal ID
`layer` - int
The layer to which this node belongs (e.g. layer = 0 -> input layer)

Method Detail



setIOrelations

```
public void setIOrelations(Node[] nodesIn,
                           Node[] nodesOut,
                           Link[] linksin,
                           Link[] linksout)
```

this method constructs nodein and nodeout arrays in order to determine the relationships of this node to others. should be called during the construction of the net

Parameters:

`nodesIn` - Node[] an array of all nodes in the previous layer to which this node is connected
`nodesOut` - Node[] an array of all nodes in the next layer to which this node is connected
`linksin` - Link[] an array of outgoing links (containing e.g. weights) to nodes in previous layer to which this node is connected
`linksout` - Link[] an array of incoming links (containing e.g. weights) to nodes in next layer to which this node is connected

---

getID

```
public int getID()
```

accessor method to the node's ID

Returns:

int

---

getThreshold

```
public double getThreshold()
```

accessor method to the node's current threshold value

Returns:

double the current threshold value for this node

---

setThreshold

```
public void setThreshold(double newThreshold)
```

updates the field threshold

Parameters:

`newThreshold` - double the new threshold value

---

getLayer

```
public Layer getLayer()
```

accessor method to the layer to which this node belongs, e.g. getLayer() = 0 -> input layer

Returns:

int

---

getOutput

```
public double getOutput()
```

accessor method to the node's output value. The value == the node's activation which is equal to the cumulative weighted input - threshold

Returns:

double this node's current output value (or activation)

---

setOutput

```
public void setOutput(double newOutput)
```

updates the field output: Only done during the feed forward phase for the *input layer* (layer = 0)

Parameters:

`newOutput` - double the new output value

UpdateOutput

```
public void UpdateOutput()
```

Updates the output and the activation according to the inputs by either

- basic calculating node type with standard exponential sigmoid activation function:

  Ouptut value: r = 1/(1 + exp(-q))
  q = sum[n](input[n]*weight[n])

  NB! If the activation function modifier <> 1, Output value = r = 1/(1 + exp(-q/modifier))
- or by calculating node type with a hyperbolic tangent activation function.
  The advantage of the hyperbolic tangent function compared to sigmoid function is the ability to model negative values (tanh(x) returns values in the range [-1,1], whereas the values returned by the standard sigmoid function lie in the range [0,1]).

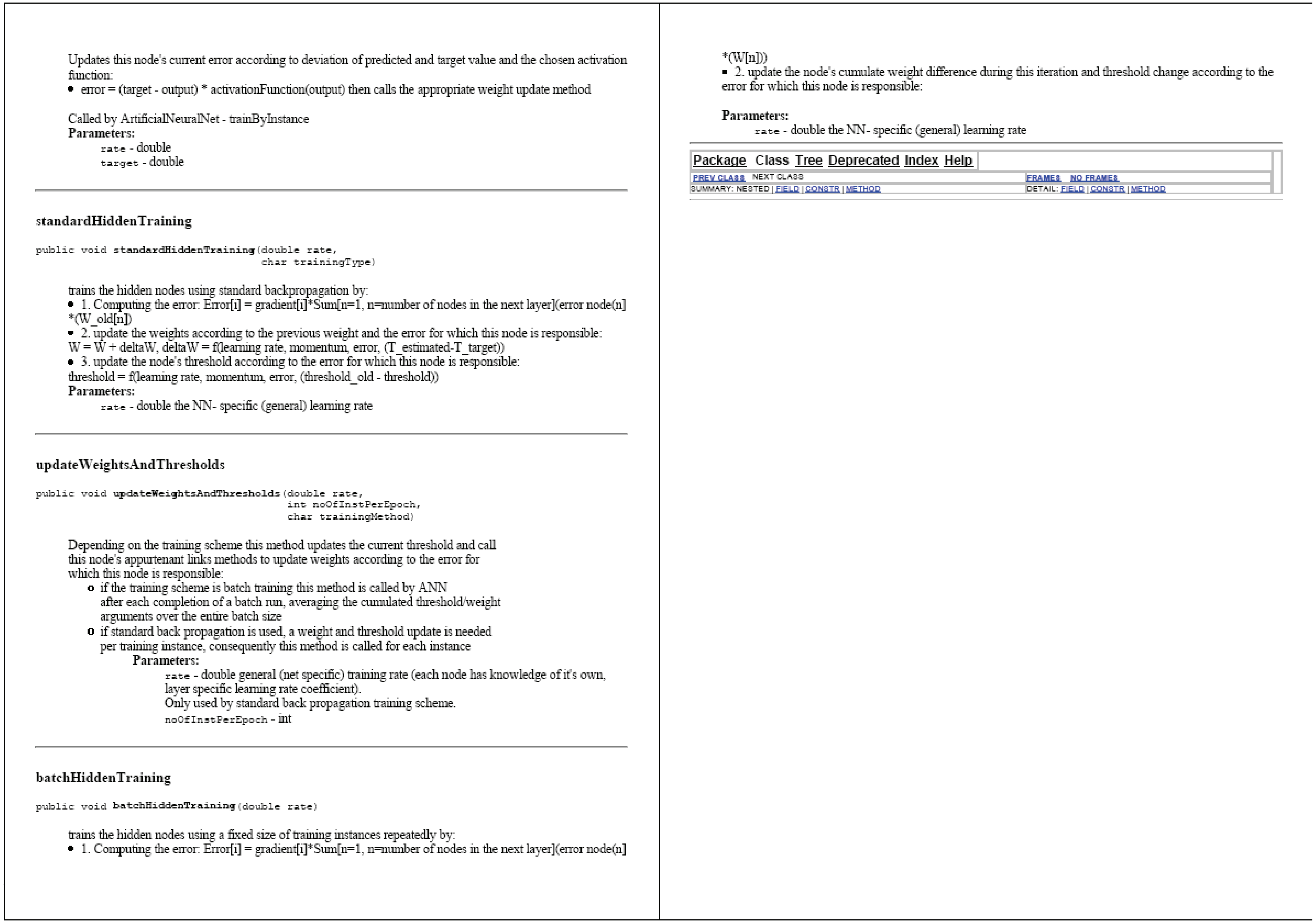
  Output value: r = tanh(q)
  tanh(x) = (exp(x) - exp(-x))/(exp(x)+exp(-x))
  exp(x) = e^x (Natural logarithm to the power of x)
  q = sum[n](input[n]*weight[n])

  NB! If the activation function modifier <> 1, Output value = r = 1/(1 + exp(-q/modifier))

---

updateErrorsWeights

```
public void updateErrorsWeights(double rate,
                                double target,
                                char trainingType)
```



Updates this node's current error according to deviation of predicted and target value and the chosen activation function:

- error = (target - output) * activationFunction(output) then calls the appropriate weight update method

Called by ArtificialNeuralNet - trainByInstance
Parameters:
`rate` - double
`target` - double

---

standardHiddenTraining

```
public void standardHiddenTraining(double rate,
                                   char trainingType)
```

trains the hidden nodes using standard backpropagation by:

- 1. Computing the error: Error[i] = gradient[i]*Sum[n=1, n=number of nodes in the next layer](error node(n] *(W_old[n])
- 2. update the weights according to the previous weight and the error for which this node is responsible: W = W + deltaW, deltaW = f(learning rate, momentum, error, (T_estimated-T_target))
- 3. update the node's threshold according to the error for which this node is responsible: threshold = f(learning rate, momentum, error, (threshold_old - threshold))

Parameters:
`rate` - double the NN- specific (general) learning rate

---

updateWeightsAndThresholds

```
public void updateWeightsAndThresholds(double rate,
                                       int noOfInstPerEpoch,
                                       char trainingMethod)
```

Depending on the training scheme this method updates the current threshold and call this node's appurtenant links methods to update weights according to the error for which this node is responsible:

- if the training scheme is batch training this method is called by ANN after each completion of a batch run, averaging the cumulated threshold/weight arguments over the entire batch size
- if standard back propagation is used, a weight and threshold update is needed per training instance, consequently this method is called for each instance

Parameters:
`rate` - double general (net specific) training rate (each node has knowledge of it's own, layer specific learning rate coefficient).
Only used by standard back propagation training scheme.
`noOfInstPerEpoch` - int

---

batchHiddenTraining

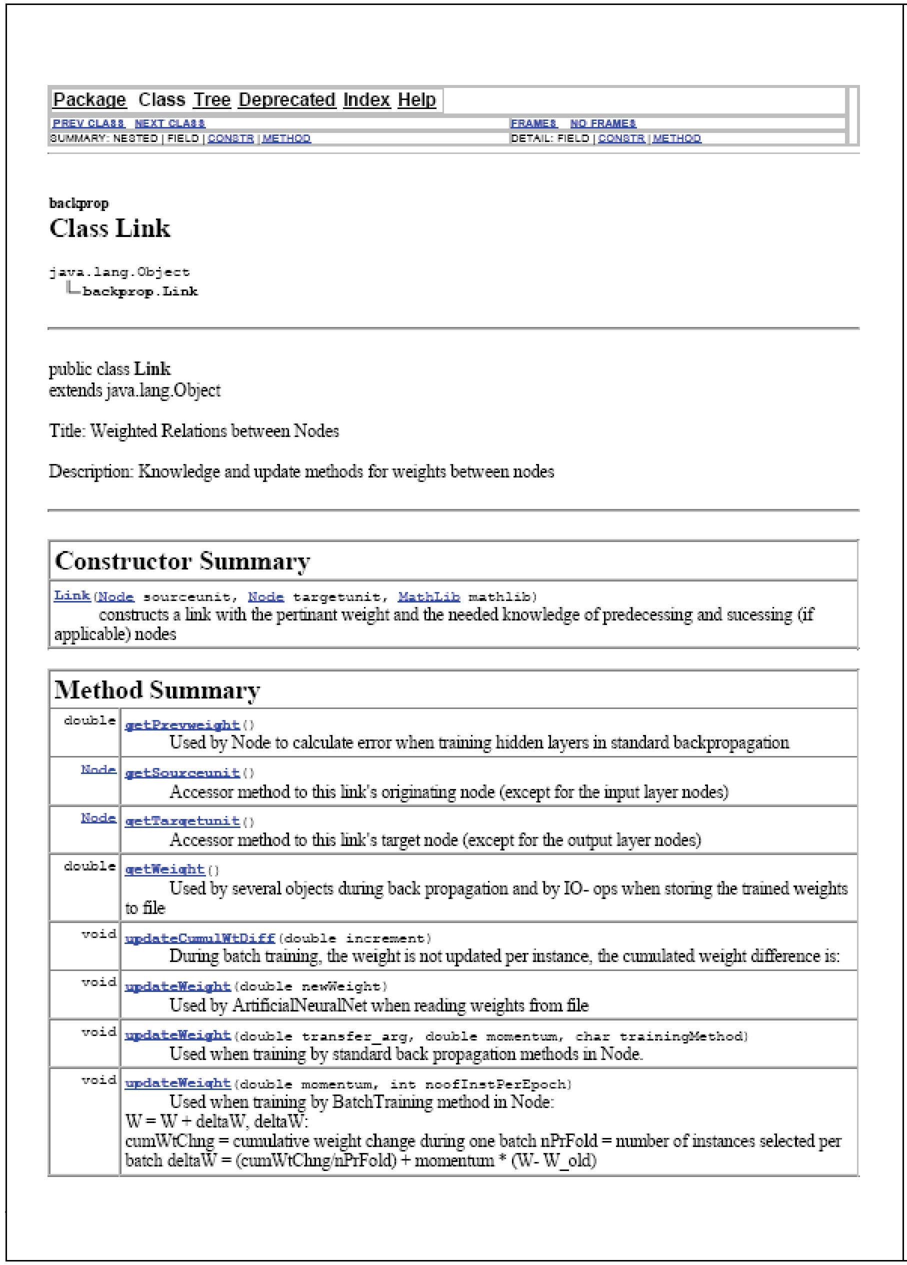
```
public void batchHiddenTraining(double rate)
```

trains the hidden nodes using a fixed size of training instances repeatedly by:

- 1. Computing the error: Error[i] = gradient[i]*Sum[n=1, n=number of nodes in the next layer](error node(n] *(W[n]))
- 2. update the node's cumulate weight difference during this iteration and threshold change according to the error for which this node is responsible:

Parameters:
`rate` - double the NN- specific (general) learning rate

Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS | FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD



Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

backprop
Class Link

```
java.lang.Object
  └backprop.Link
```

public class **Link**
extends java.lang.Object

Title: Weighted Relations between Nodes

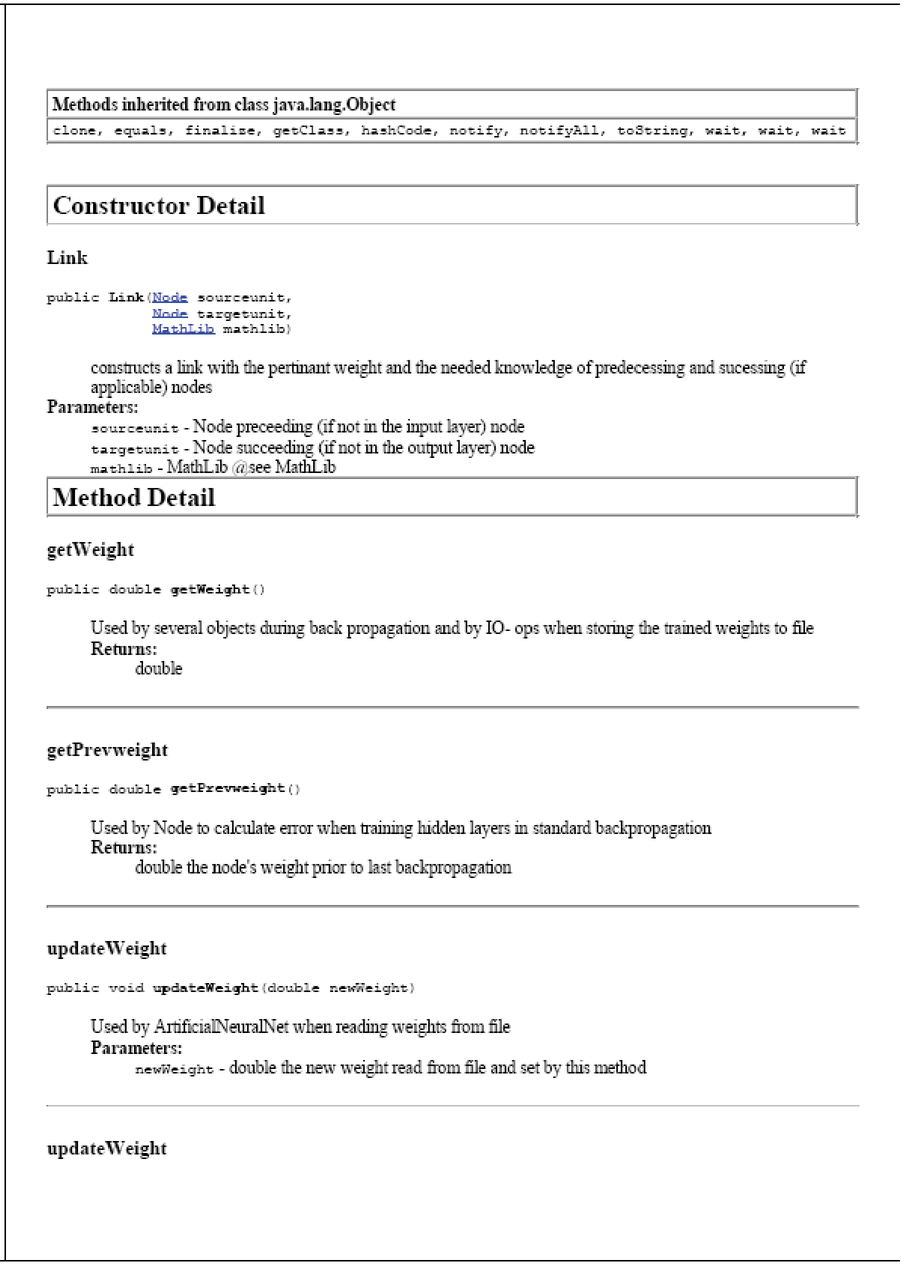
Description: Knowledge and update methods for weights between nodes

Constructor Summary

`Link(Node sourceunit, Node targetunit, MathLib mathlib)`
constructs a link with the pertinant weight and the needed knowledge of predecessing and sucessing (if applicable) nodes

Method Summary

`double`	`getPrevweight()` Used by Node to calculate error when training hidden layers in standard backpropagation
`Node`	`getSourceunit()` Accessor method to this link's originating node (except for the input layer nodes)
`Node`	`getTargetunit()` Accessor method to this link's target node (except for the output layer nodes)
`double`	`getWeight()` Used by several objects during back propagation and by IO- ops when storing the trained weights to file
`void`	`updateCumulWtDiff(double increment)` During batch training, the weight is not updated per instance, the cumulated weight difference is:
`void`	`updateWeight(double newWeight)` Used by ArtificialNeuralNet when reading weights from file
`void`	`updateWeight(double transfer_arg, double momentum, char trainingMethod)` Used when training by standard back propagation methods in Node.
`void`	`updateWeight(double momentum, int noofInstPerEpoch)` Used when training by BatchTraining method in Node: W = W + deltaW, deltaW: cumWtChng = cumulative weight change during one batch nPrFold = number of instances selected per batch deltaW = (cumWtChng/nPrFold) + momentum * (W- W_old)



Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Link

```
public Link(Node sourceunit,
            Node targetunit,
            MathLib mathlib)
```

constructs a link with the pertinant weight and the needed knowledge of predecessing and sucessing (if applicable) nodes
Parameters:
`sourceunit` - Node preceeding (if not in the input layer) node
`targetunit` - Node succeeding (if not in the output layer) node
`mathlib` - MathLib @see MathLib

Method Detail

getWeight

`public double getWeight()`

Used by several objects during back propagation and by IO- ops when storing the trained weights to file
Returns:
double

getPrevweight

`public double getPrevweight()`

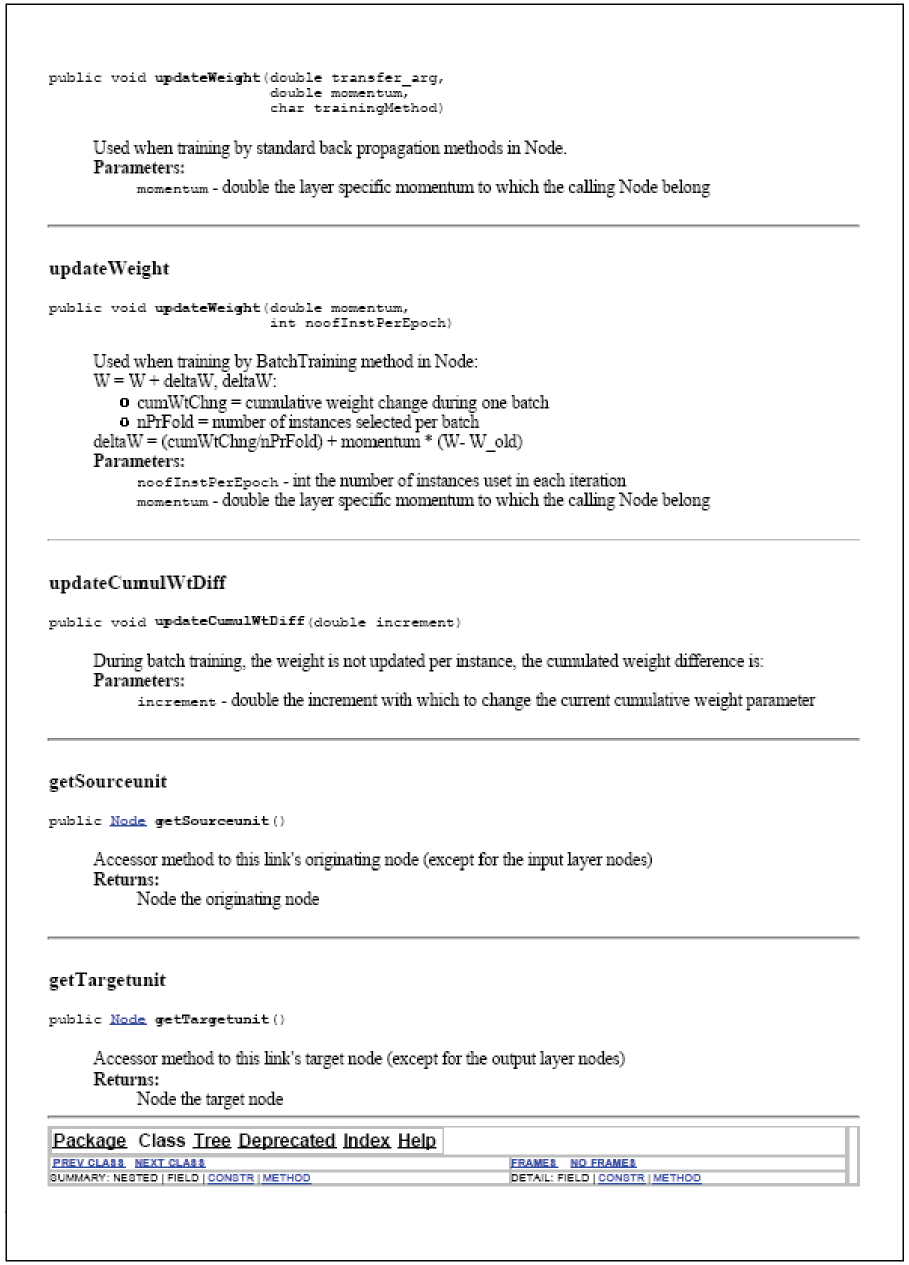
Used by Node to calculate error when training hidden layers in standard backpropagation
Returns:
double the node's weight prior to last backpropagation

updateWeight

`public void updateWeight(double newWeight)`

Used by ArtificialNeuralNet when reading weights from file
Parameters:
`newWeight` - double the new weight read from file and set by this method

updateWeight



```
public void updateWeight(double transfer_arg,
                         double momentum,
                         char trainingMethod)
```

Used when training by standard back propagation methods in Node.
Parameters:
`momentum` - double the layer specific momentum to which the calling Node belong

updateWeight

```
public void updateWeight(double momentum,
                         int noofInstPerEpoch)
```

Used when training by BatchTraining method in Node:
W = W + deltaW, deltaW:
- cumWtChng = cumulative weight change during one batch
- nPrFold = number of instances selected per batch

deltaW = (cumWtChng/nPrFold) + momentum * (W- W_old)
Parameters:
`noofInstPerEpoch` - int the number of instances uset in each iteration
`momentum` - double the layer specific momentum to which the calling Node belong

updateCumulWtDiff

```
public void updateCumulWtDiff(double increment)
```

During batch training, the weight is not updated per instance, the cumulated weight difference is:
Parameters:
`increment` - double the increment with which to change the current cumulative weight parameter

getSourceunit

```
public Node getSourceunit()
```

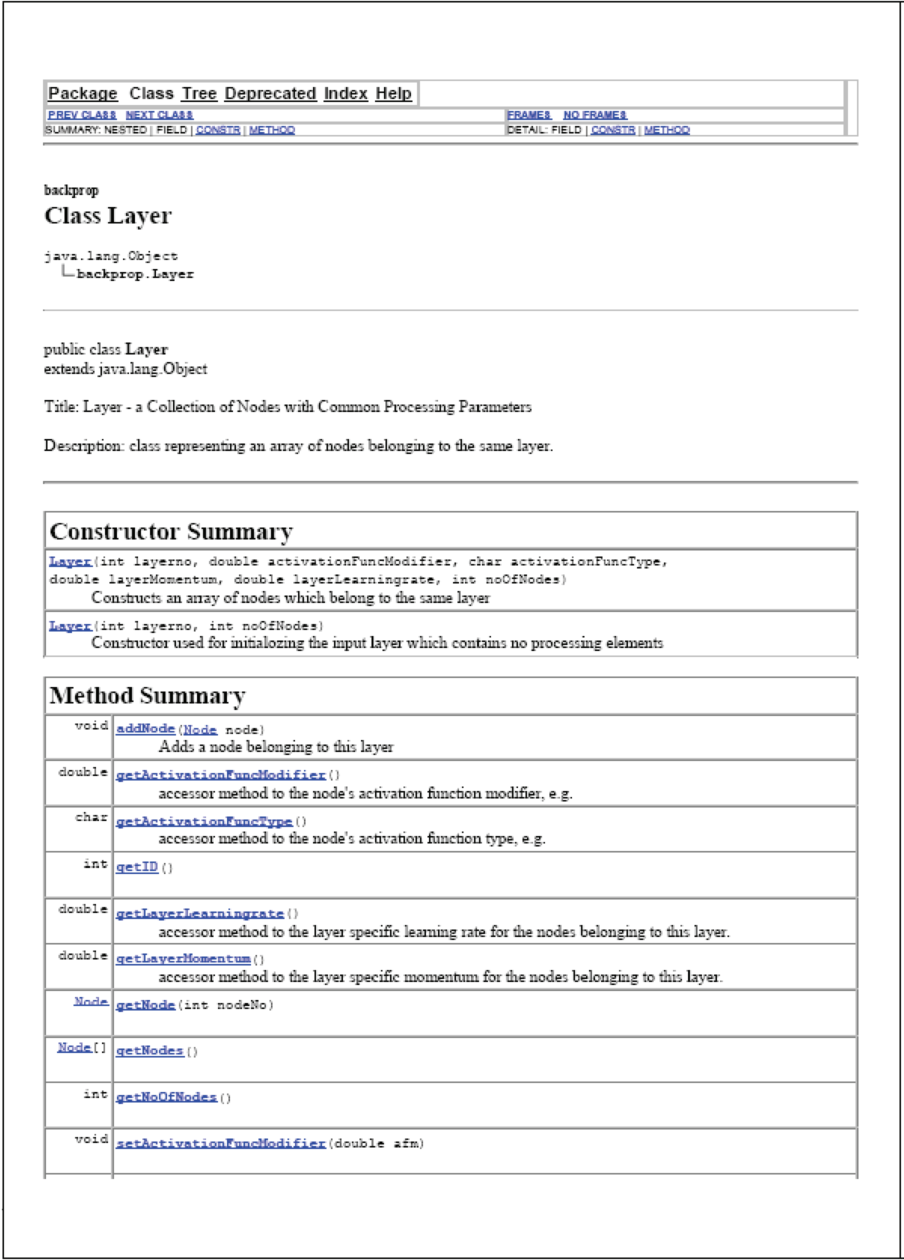
Accessor method to this link's originating node (except for the input layer nodes)
Returns:
Node the originating node

getTargetunit

```
public Node getTargetunit()
```

Accessor method to this link's target node (except for the output layer nodes)
Returns:
Node the target node

Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD



Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS | FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

backprop
Class Layer

```
java.lang.Object
  └backprop.Layer
```

public class **Layer**
extends java.lang.Object

Title: Layer - a Collection of Nodes with Common Processing Parameters

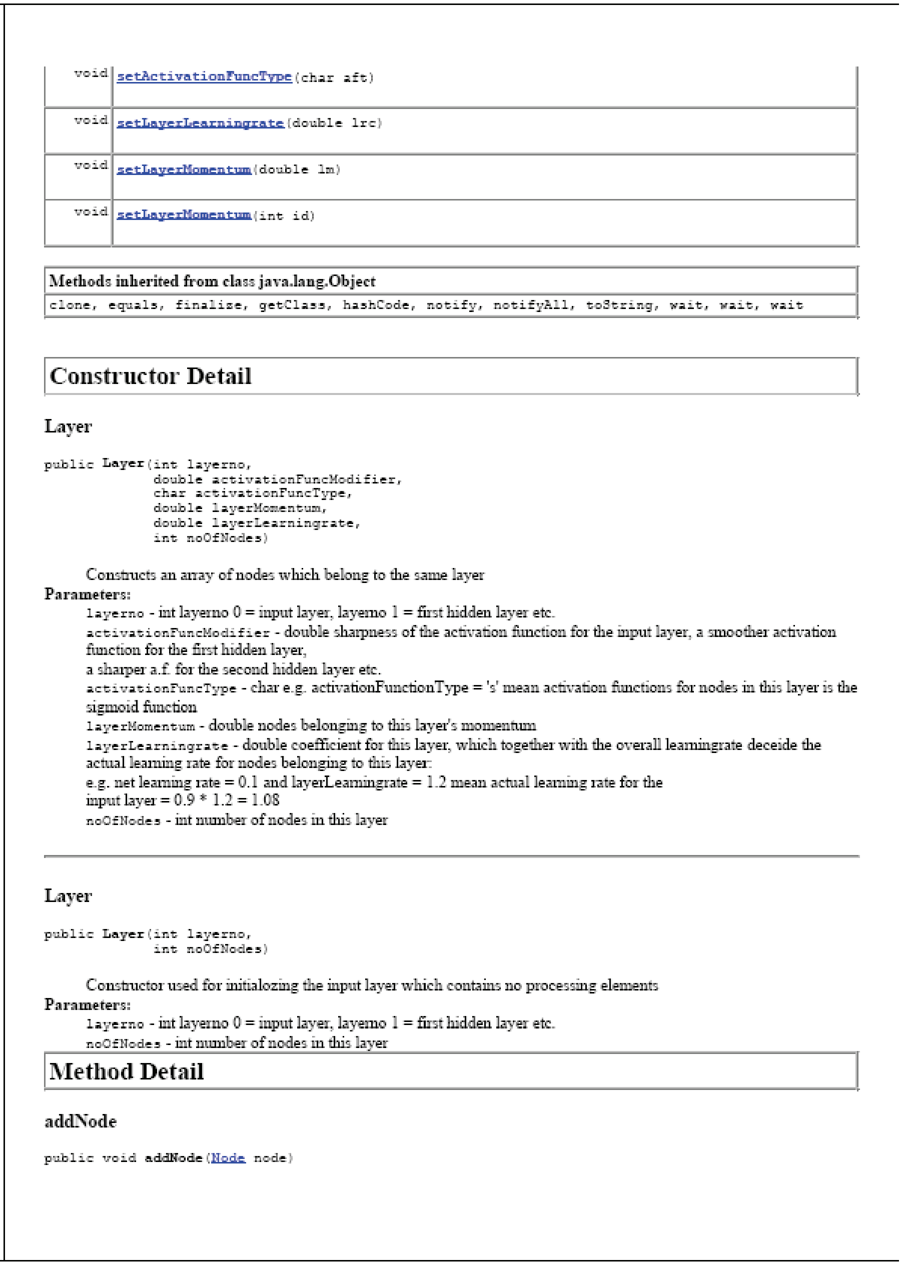
Description: class representing an array of nodes belonging to the same layer.

Constructor Summary

`Layer(int layerno, double activationFuncModifier, char activationFuncType, double layerMomentum, double layerLearningrate, int noOfNodes)` Constructs an array of nodes which belong to the same layer
`Layer(int layerno, int noOfNodes)` Constructor used for initialozing the input layer which contains no processing elements

Method Summary

`void`	`addNode(Node node)` Adds a node belonging to this layer
`double`	`getActivationFuncModifier()` accessor method to the node's activation function modifier, e.g.
`char`	`getActivationFuncType()` accessor method to the node's activation function type, e.g.
`int`	`getID()`
`double`	`getLayerLearningrate()` accessor method to the layer specific learning rate for the nodes belonging to this layer.
`double`	`getLayerMomentum()` accessor method to the layer specific momentum for the nodes belonging to this layer.
`Node`	`getNode(int nodeNo)`
`Node[]`	`getNodes()`
`int`	`getNoOfNodes()`
`void`	`setActivationFuncModifier(double afm)`



`void`	`setActivationFuncType(char aft)`
`void`	`setLayerLearningrate(double lrc)`
`void`	`setLayerMomentum(double lm)`
`void`	`setLayerMomentum(int id)`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Layer

```
public Layer(int layerno,
             double activationFuncModifier,
             char activationFuncType,
             double layerMomentum,
             double layerLearningrate,
             int noOfNodes)
```

Constructs an array of nodes which belong to the same layer

Parameters:
`layerno` - int layerno 0 = input layer, layerno 1 = first hidden layer etc.
`activationFuncModifier` - double sharpness of the activation function for the input layer, a smoother activation function for the first hidden layer,
a sharper a.f. for the second hidden layer etc.
`activationFuncType` - char e.g. activationFunctionType = 's' mean activation functions for nodes in this layer is the sigmoid function
`layerMomentum` - double nodes belonging to this layer's momentum
`layerLearningrate` - double coefficient for this layer, which together with the overall learningrate deceide the actual learning rate for nodes belonging to this layer:
e.g. net learning rate = 0.1 and layerLearningrate = 1.2 mean actual learning rate for the input layer = 0.9 * 1.2 = 1.08
`noOfNodes` - int number of nodes in this layer

Layer

```
public Layer(int layerno,
             int noOfNodes)
```

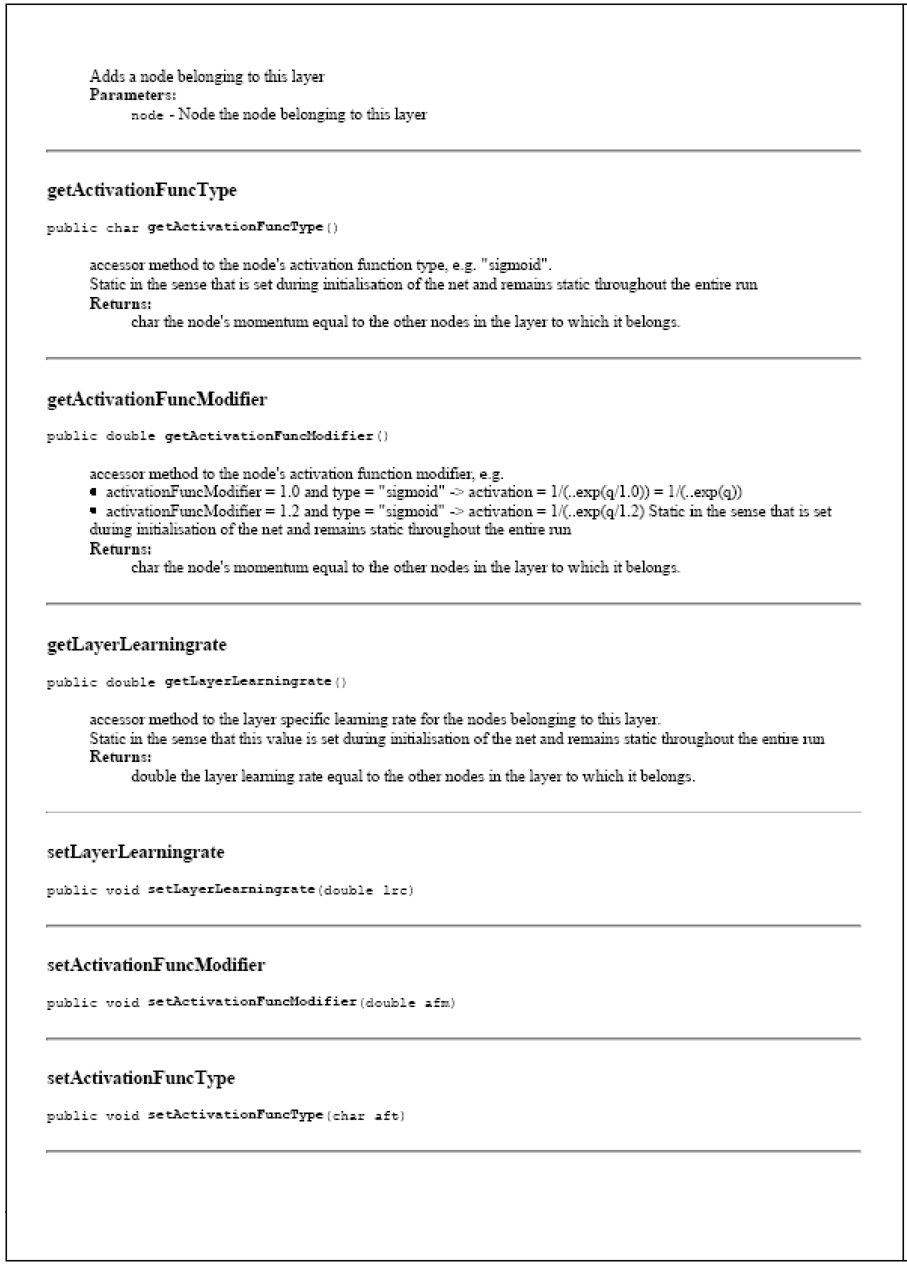
Constructor used for initialozing the input layer which contains no processing elements

Parameters:
`layerno` - int layerno 0 = input layer, layerno 1 = first hidden layer etc.
`noOfNodes` - int number of nodes in this layer

Method Detail

addNode

```
public void addNode(Node node)
```



Adds a node belonging to this layer
Parameters:
`node` - Node the node belonging to this layer

---

getActivationFuncType

```
public char getActivationFuncType()
```

accessor method to the node's activation function type, e.g. "sigmoid".
Static in the sense that is set during initialisation of the net and remains static throughout the entire run
Returns:
char the node's momentum equal to the other nodes in the layer to which it belongs.

---

getActivationFuncModifier

```
public double getActivationFuncModifier()
```

accessor method to the node's activation function modifier, e.g.
- activationFuncModifier = 1.0 and type = "sigmoid" -> activation = 1/(..exp(q/1.0)) = 1/(..exp(q))
- activationFuncModifier = 1.2 and type = "sigmoid" -> activation = 1/(..exp(q/1.2) Static in the sense that is set during initialisation of the net and remains static throughout the entire run

Returns:
char the node's momentum equal to the other nodes in the layer to which it belongs.

---

getLayerLearningrate

```
public double getLayerLearningrate()
```

accessor method to the layer specific learning rate for the nodes belonging to this layer.
Static in the sense that this value is set during initialisation of the net and remains static throughout the entire run
Returns:
double the layer learning rate equal to the other nodes in the layer to which it belongs.

---

setLayerLearningrate

```
public void setLayerLearningrate(double lrc)
```

---

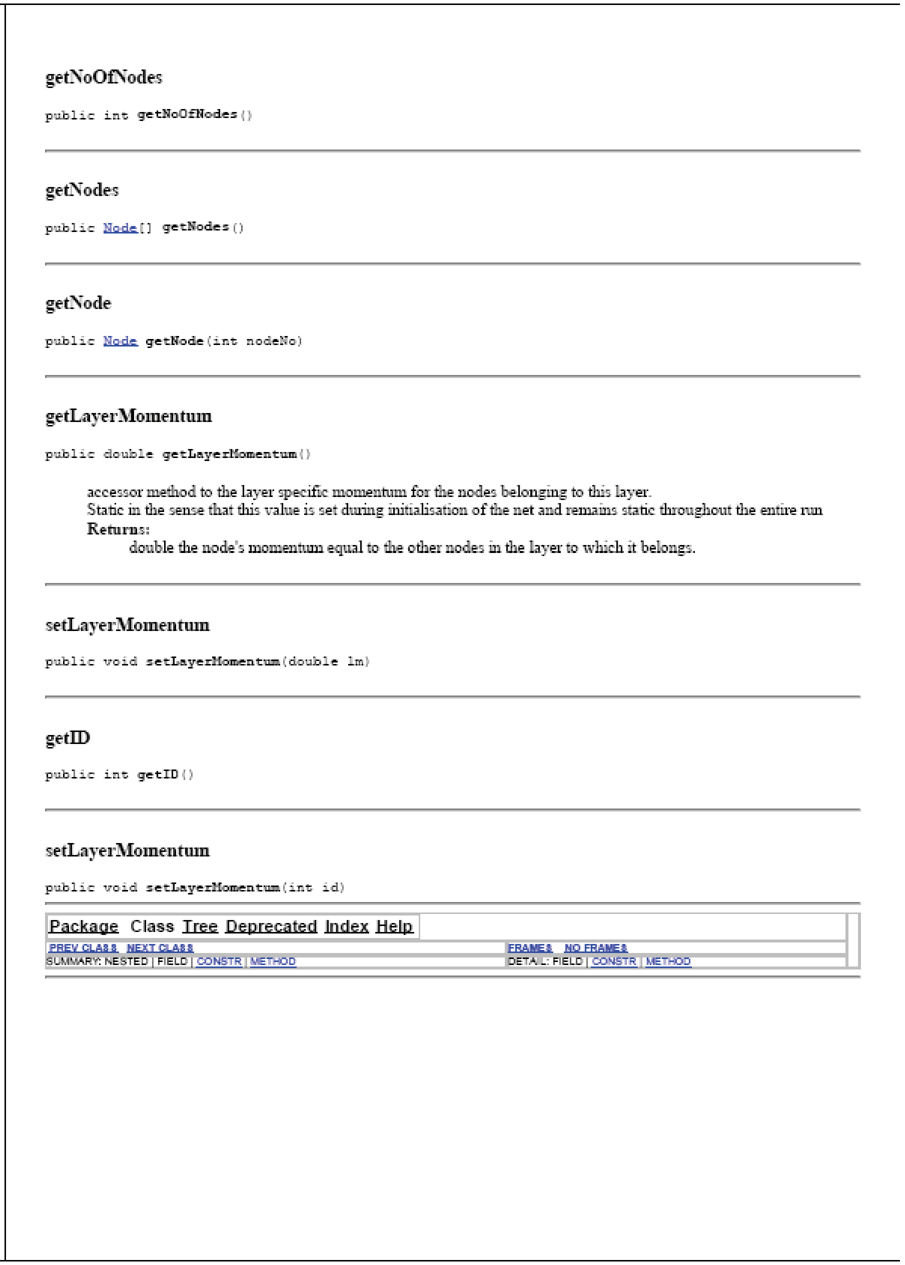
setActivationFuncModifier

```
public void setActivationFuncModifier(double afm)
```

---

setActivationFuncType

```
public void setActivationFuncType(char aft)
```



getNoOfNodes

```
public int getNoOfNodes()
```

---

getNodes

```
public Node[] getNodes()
```

---

getNode

```
public Node getNode(int nodeNo)
```

---

getLayerMomentum

```
public double getLayerMomentum()
```

accessor method to the layer specific momentum for the nodes belonging to this layer.
Static in the sense that this value is set during initialisation of the net and remains static throughout the entire run
Returns:
double the node's momentum equal to the other nodes in the layer to which it belongs.

---

setLayerMomentum

```
public void setLayerMomentum(double lm)
```

---

getID

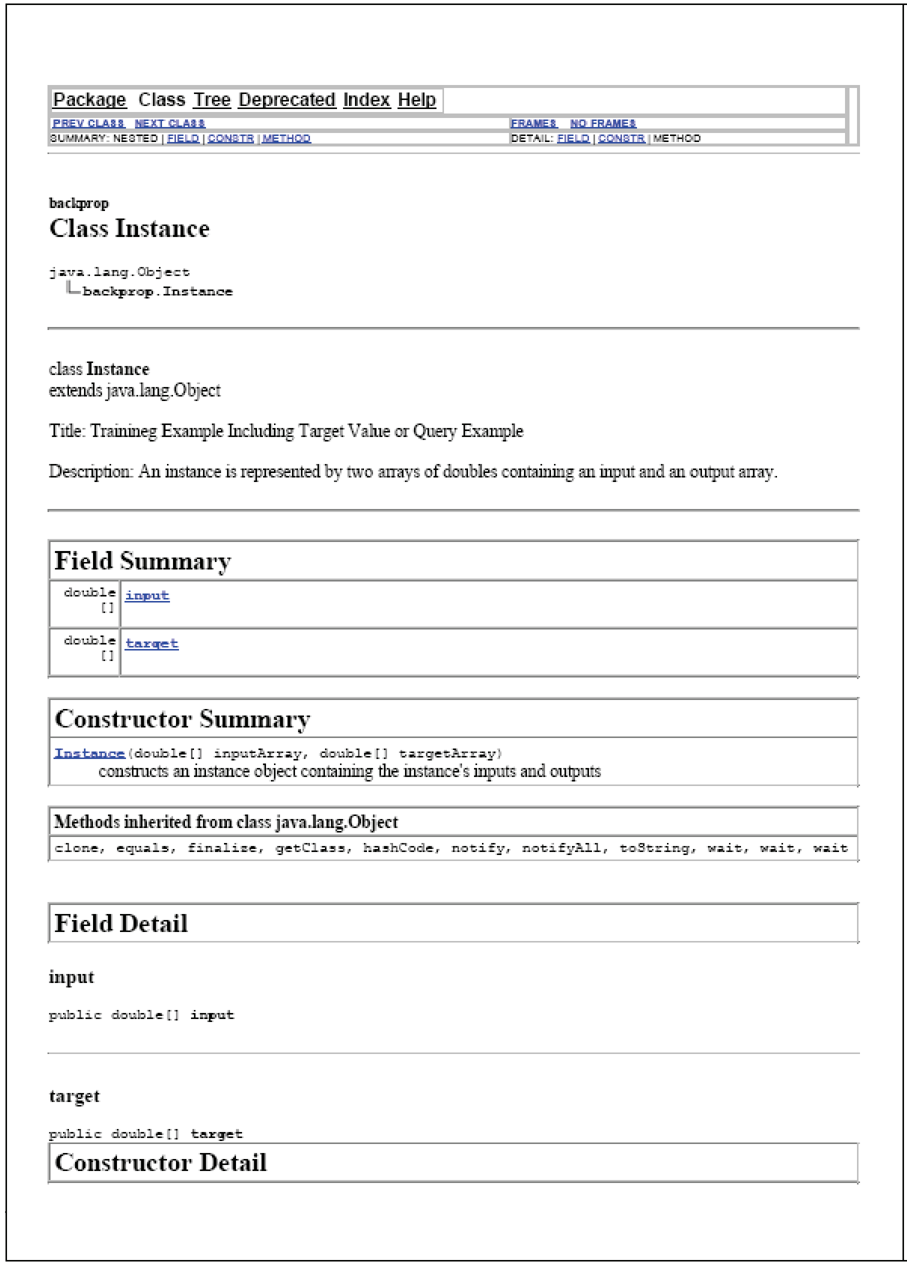
```
public int getID()
```

---

setLayerMomentum

```
public void setLayerMomentum(int id)
```

Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD



Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

backprop
Class Instance

```
java.lang.Object
  └backprop.Instance
```

class **Instance**
extends java.lang.Object

Title: Traininеg Example Including Target Value or Query Example

Description: An instance is represented by two arrays of doubles containing an input and an output array.

Field Summary

`double[]`	`input`
`double[]`	`target`

Constructor Summary

`Instance(double[] inputArray, double[] targetArray)`
constructs an instance object containing the instance's inputs and outputs

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

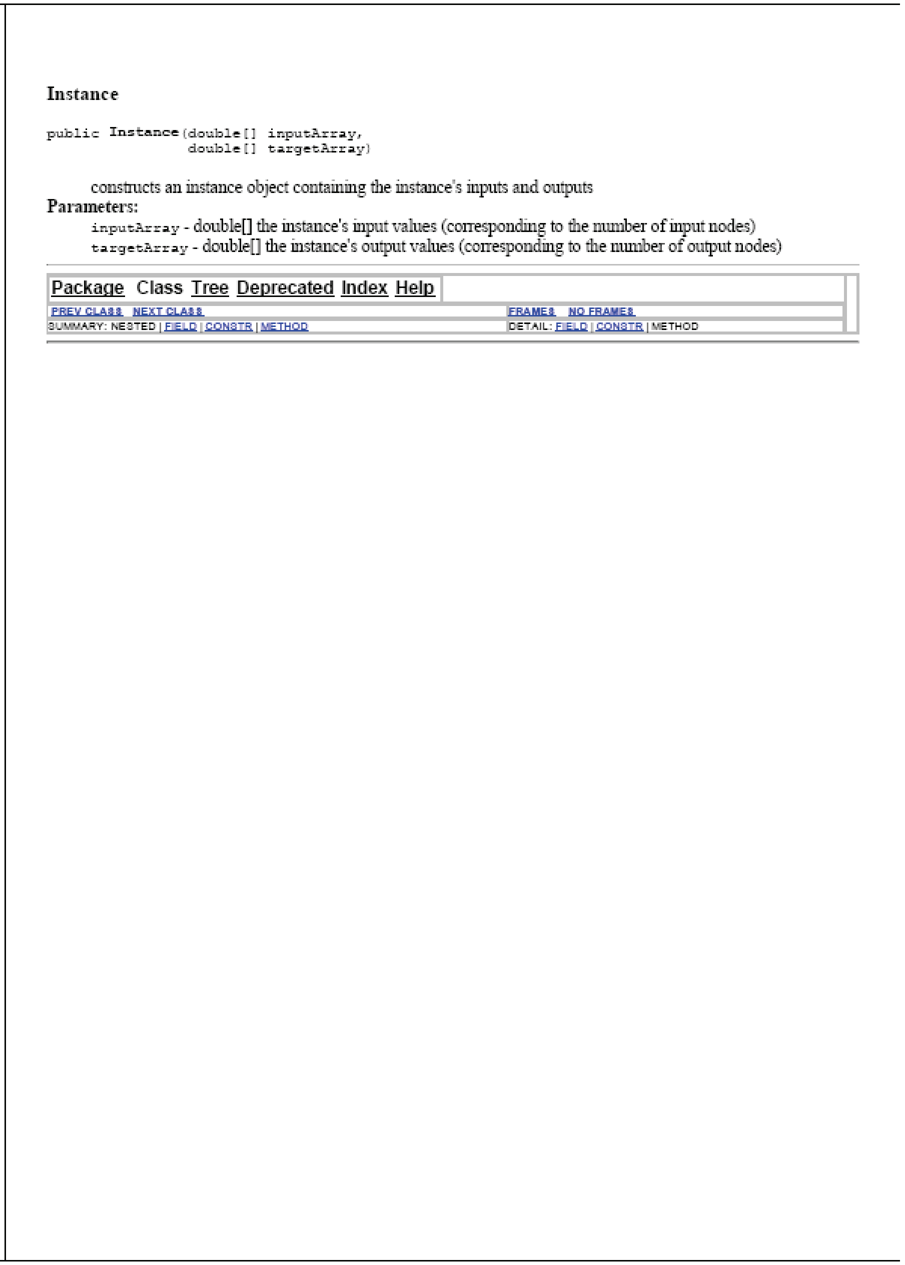
input

`public double[] input`

target

`public double[] target`

Constructor Detail



Instance

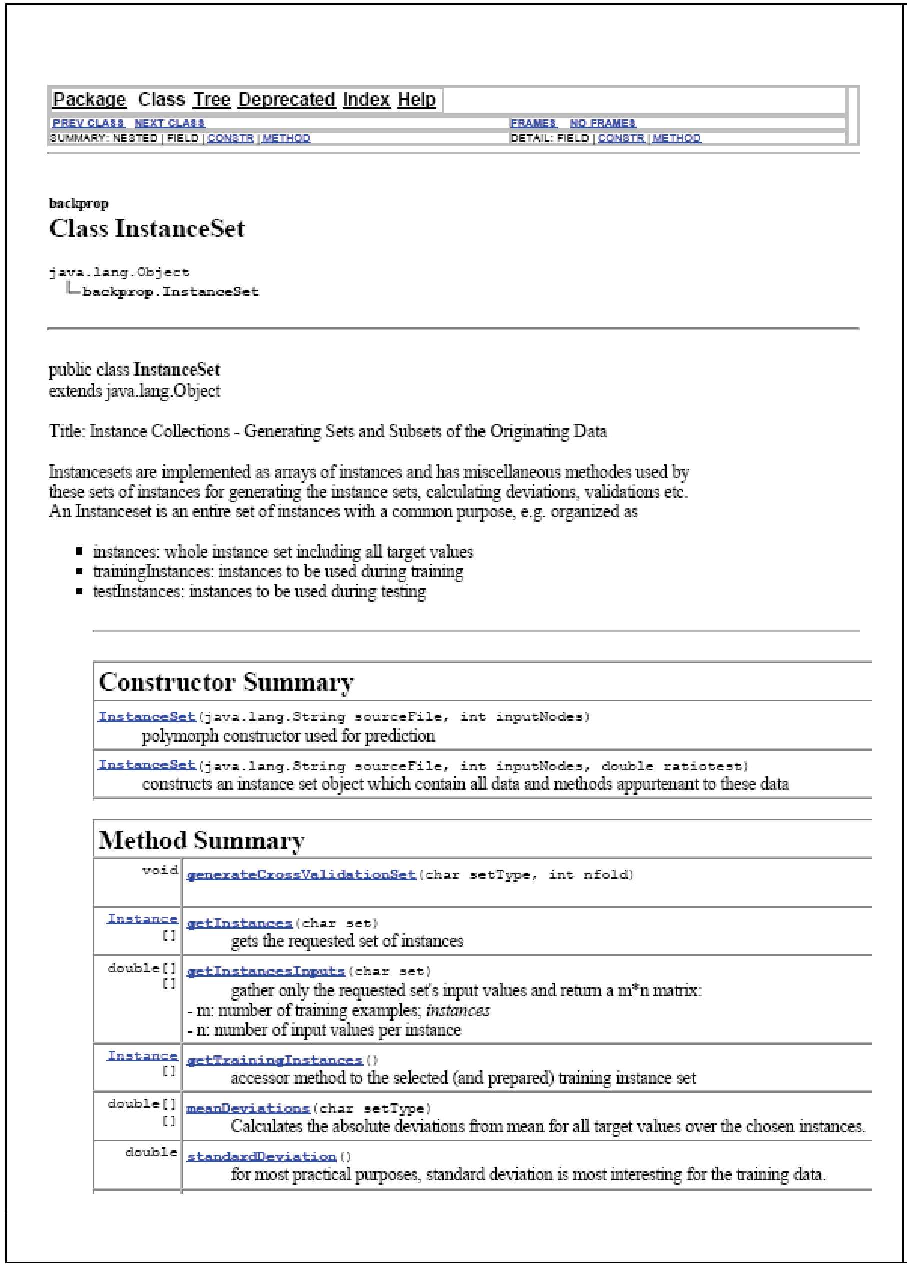
```
public Instance(double[] inputArray,
                double[] targetArray)
```

constructs an instance object containing the instance's inputs and outputs

Parameters:
`inputArray` - double[] the instance's input values (corresponding to the number of input nodes)
`targetArray` - double[] the instance's output values (corresponding to the number of output nodes)

Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD



Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

backprop
Class InstanceSet

```
java.lang.Object
  └backprop.InstanceSet
```

public class **InstanceSet**
extends java.lang.Object

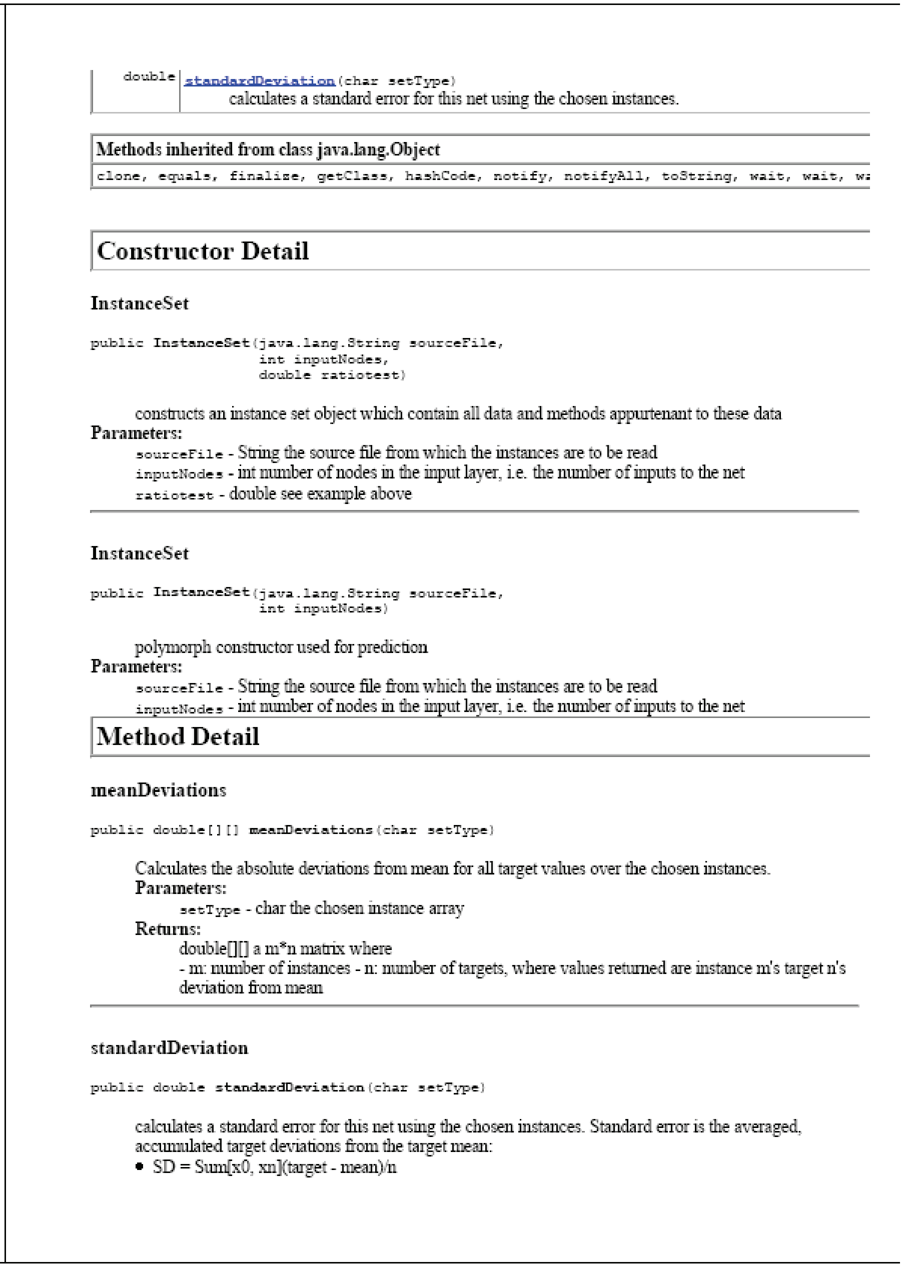
Title: Instance Collections - Generating Sets and Subsets of the Originating Data

Instancesets are implemented as arrays of instances and has miscellaneous methodes used by these sets of instances for generating the instance sets, calculating deviations, validations etc. An Instanceset is an entire set of instances with a common purpose, e.g. organized as

- instances: whole instance set including all target values
- trainingInstances: instances to be used during training
- testInstances: instances to be used during testing

Constructor Summary
`InstanceSet(java.lang.String sourceFile, int inputNodes)` polymorph constructor used for prediction
`InstanceSet(java.lang.String sourceFile, int inputNodes, double ratiotest)` constructs an instance set object which contain all data and methods appurtenant to these data

Method Summary	
`void`	`generateCrossValidationSet(char setType, int nfold)`
`Instance[]`	`getInstances(char set)` gets the requested set of instances
`double[][]`	`getInstancesInputs(char set)` gather only the requested set's input values and return a m*n matrix: - m: number of training examples; *instances* - n: number of input values per instance
`Instance[]`	`getTrainingInstances()` accessor method to the selected (and prepared) training instance set
`double[][]`	`meanDeviations(char setType)` Calculates the absolute deviations from mean for all target values over the chosen instances.
`double`	`standardDeviation()` for most practical purposes, standard deviation is most interesting for the training data.



`double`	`standardDeviation(char setType)` calculates a standard error for this net using the chosen instances.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wa`

Constructor Detail

InstanceSet

```
public InstanceSet(java.lang.String sourceFile,
                   int inputNodes,
                   double ratiotest)
```

constructs an instance set object which contain all data and methods appurtenant to these data
Parameters:
`sourceFile` - String the source file from which the instances are to be read
`inputNodes` - int number of nodes in the input layer, i.e. the number of inputs to the net
`ratiotest` - double see example above

InstanceSet

```
public InstanceSet(java.lang.String sourceFile,
                   int inputNodes)
```

polymorph constructor used for prediction
Parameters:
`sourceFile` - String the source file from which the instances are to be read
`inputNodes` - int number of nodes in the input layer, i.e. the number of inputs to the net

Method Detail

meanDeviations

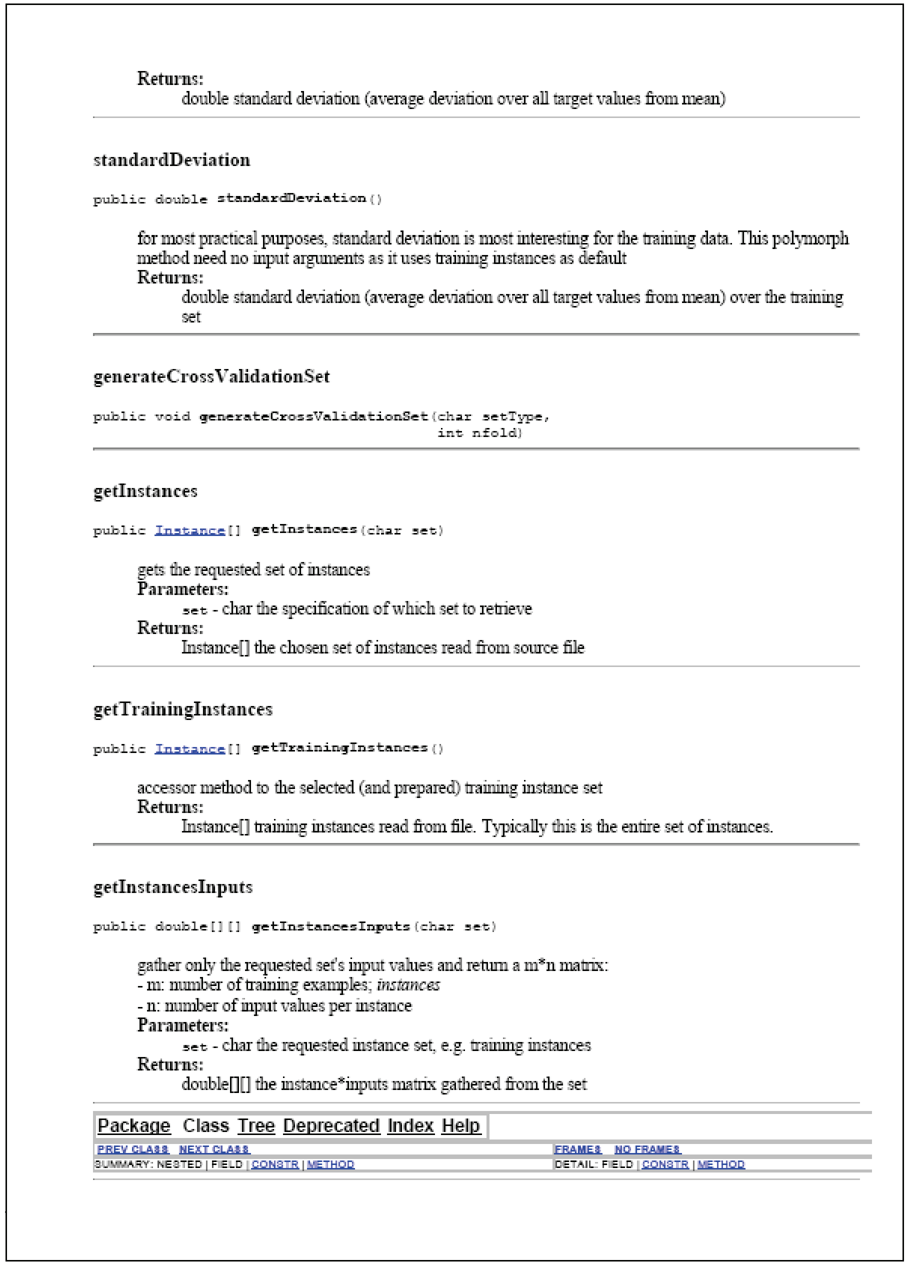
```
public double[][] meanDeviations(char setType)
```

Calculates the absolute deviations from mean for all target values over the chosen instances.
Parameters:
`setType` - char the chosen instance array
Returns:
double[][] a m*n matrix where
- m: number of instances - n: number of targets, where values returned are instance m's target n's deviation from mean

standardDeviation

```
public double standardDeviation(char setType)
```

calculates a standard error for this net using the chosen instances. Standard error is the averaged, accumulated target deviations from the target mean:
- SD = Sum[x0, xn](target - mean)/n



Returns:
double standard deviation (average deviation over all target values from mean)

---

standardDeviation

```
public double standardDeviation()
```

for most practical purposes, standard deviation is most interesting for the training data. This polymorph method need no input arguments as it uses training instances as default
Returns:
double standard deviation (average deviation over all target values from mean) over the training set

---

generateCrossValidationSet

```
public void generateCrossValidationSet(char setType,
                                       int nfold)
```

---

getInstances

```
public Instance[] getInstances(char set)
```

gets the requested set of instances
Parameters:
`set` - char the specification of which set to retrieve
Returns:
Instance[] the chosen set of instances read from source file

---

getTrainingInstances

```
public Instance[] getTrainingInstances()
```

accessor method to the selected (and prepared) training instance set
Returns:
Instance[] training instances read from file. Typically this is the entire set of instances.

---

getInstancesInputs

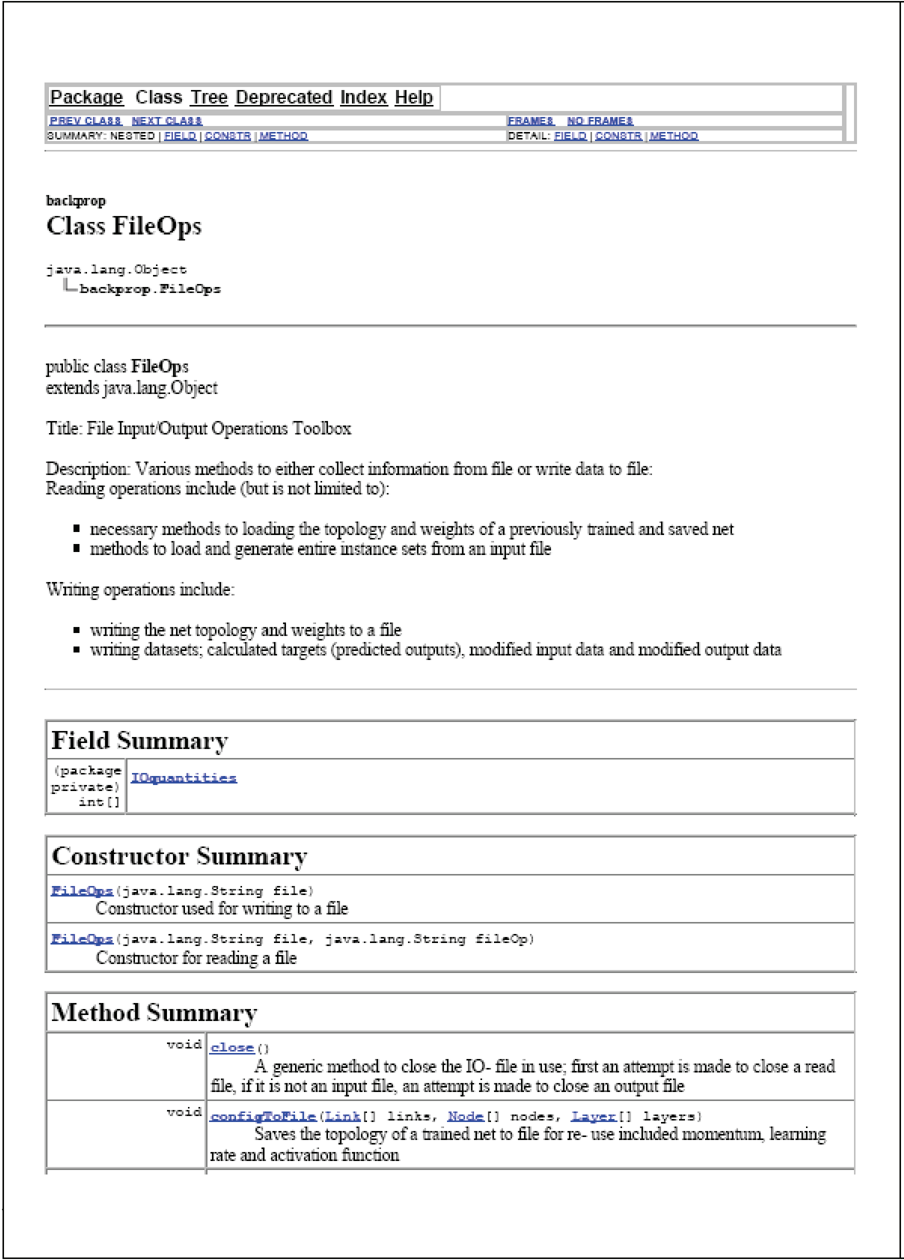
```
public double[][] getInstancesInputs(char set)
```

gather only the requested set's input values and return a m*n matrix:
- m: number of training examples; *instances*
- n: number of input values per instance
Parameters:
`set` - char the requested instance set, e.g. training instances
Returns:
double[][] the instance*inputs matrix gathered from the set

Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD



Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

backprop
Class FileOps

```
java.lang.Object
  └backprop.FileOps
```

public class **FileOps**
extends java.lang.Object

Title: File Input/Output Operations Toolbox

Description: Various methods to either collect information from file or write data to file:
Reading operations include (but is not limited to):

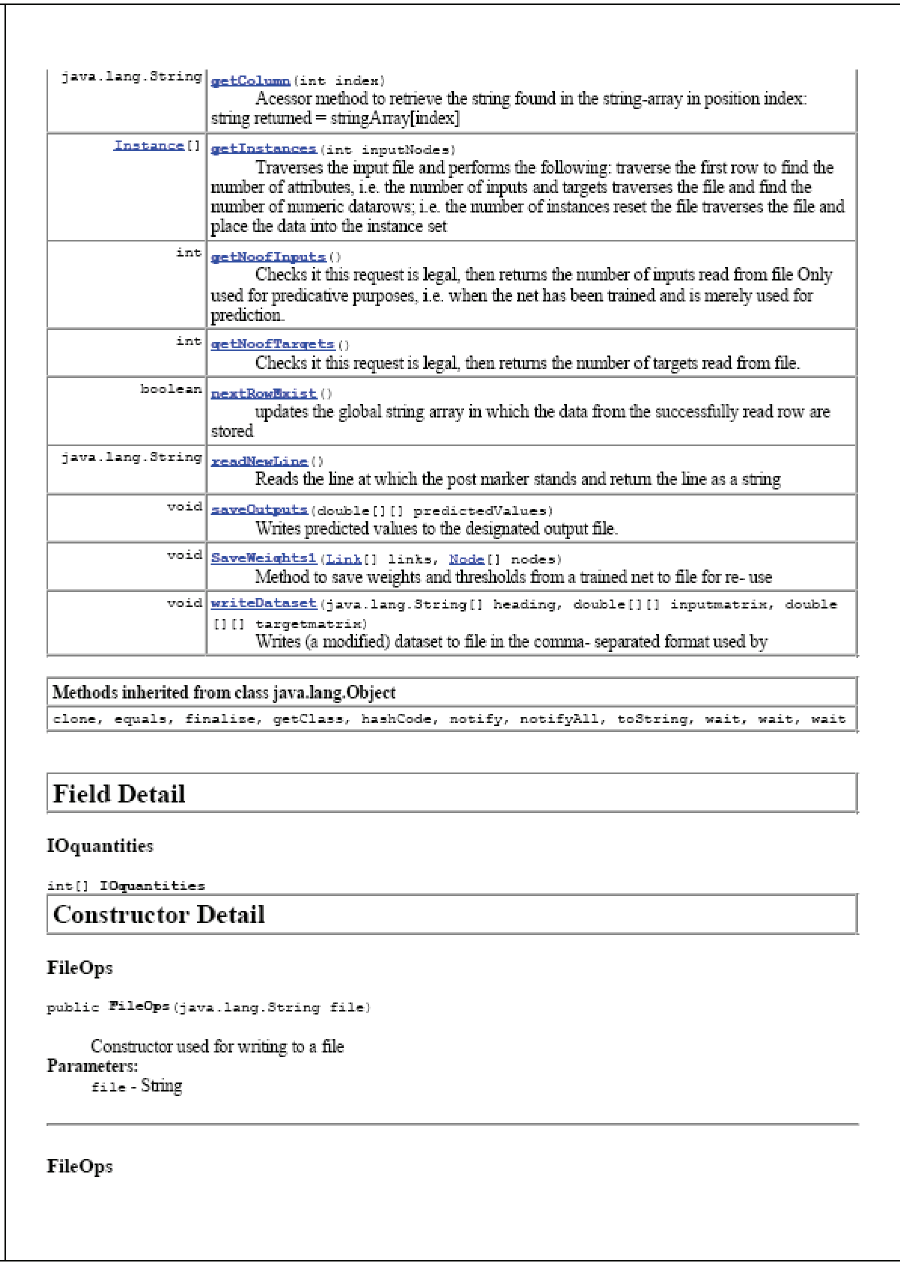
- necessary methods to loading the topology and weights of a previously trained and saved net
- methods to load and generate entire instance sets from an input file

Writing operations include:

- writing the net topology and weights to a file
- writing datasets; calculated targets (predicted outputs), modified input data and modified output data

Field Summary	
`(package private) int[]`	`IOquantities`

Constructor Summary
`FileOps(java.lang.String file)` Constructor used for writing to a file
`FileOps(java.lang.String file, java.lang.String fileOp)` Constructor for reading a file



Method Summary	
`void`	`close()` A generic method to close the IO- file in use; first an attempt is made to close a read file, if it is not an input file, an attempt is made to close an output file
`void`	`configToFile(Link[] links, Node[] nodes, Layer[] layers)` Saves the topology of a trained net to file for re- use included momentum, learning rate and activation function
`java.lang.String`	`getColumn(int index)` Acessor method to retrieve the string found in the string-array in position index: string returned = stringArray[index]
`Instance[]`	`getInstances(int inputNodes)` Traverses the input file and performs the following: traverse the first row to find the number of attributes, i.e. the number of inputs and targets traverses the file and find the number of numeric datarows; i.e. the number of instances reset the file traverses the file and place the data into the instance set
`int`	`getNoofInputs()` Checks it this request is legal, then returns the number of inputs read from file Only used for predicative purposes, i.e. when the net has been trained and is merely used for prediction.
`int`	`getNoofTargets()` Checks it this request is legal, then returns the number of targets read from file.
`boolean`	`nextRowExist()` updates the global string array in which the data from the successfully read row are stored
`java.lang.String`	`readNewLine()` Reads the line at which the post marker stands and return the line as a string
`void`	`saveOutputs(double[][] predictedValues)` Writes predicted values to the designated output file.
`void`	`SaveWeights1(Link[] links, Node[] nodes)` Method to save weights and thresholds from a trained net to file for re- use
`void`	`writeDataset(java.lang.String[] heading, double[][] inputmatrix, double[][] targetmatrix)` Writes (a modified) dataset to file in the comma- separated format used by

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

IOquantities

```
int[] IOquantities
```

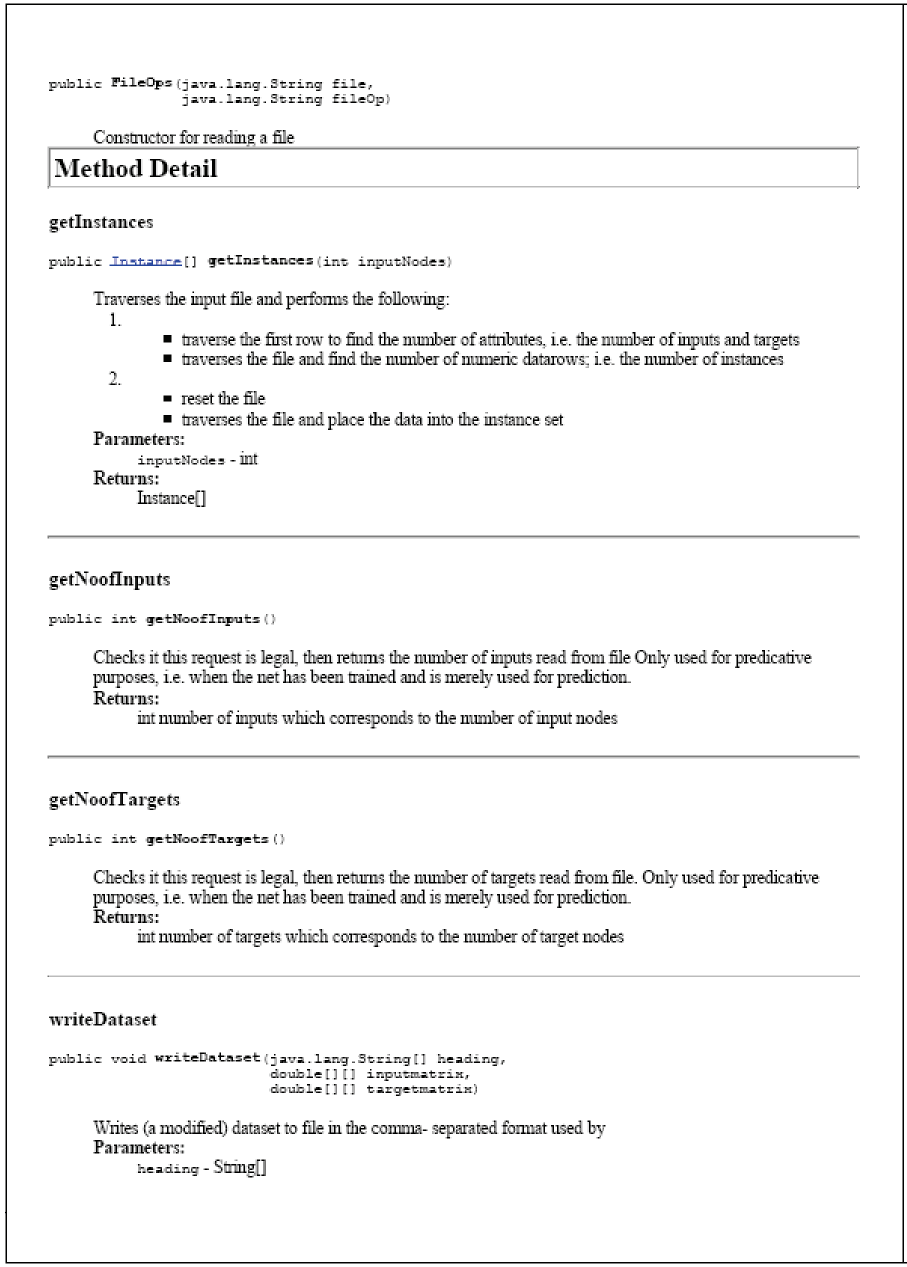
Constructor Detail

FileOps

```
public FileOps(java.lang.String file)
```

Constructor used for writing to a file
Parameters:
`file` - String

FileOps



```
public FileOps(java.lang.String file,
               java.lang.String fileOp)
```

Constructor for reading a file

Method Detail

getInstances

```
public Instance[] getInstances(int inputNodes)
```

Traverses the input file and performs the following:

1.
   - traverse the first row to find the number of attributes, i.e. the number of inputs and targets
   - traverses the file and find the number of numeric datarows; i.e. the number of instances
2.
   - reset the file
   - traverses the file and place the data into the instance set

Parameters:
`inputNodes` - int
Returns:
Instance[]

getNoofInputs

```
public int getNoofInputs()
```

Checks it this request is legal, then returns the number of inputs read from file Only used for predicative purposes, i.e. when the net has been trained and is merely used for prediction.
Returns:
int number of inputs which corresponds to the number of input nodes

getNoofTargets

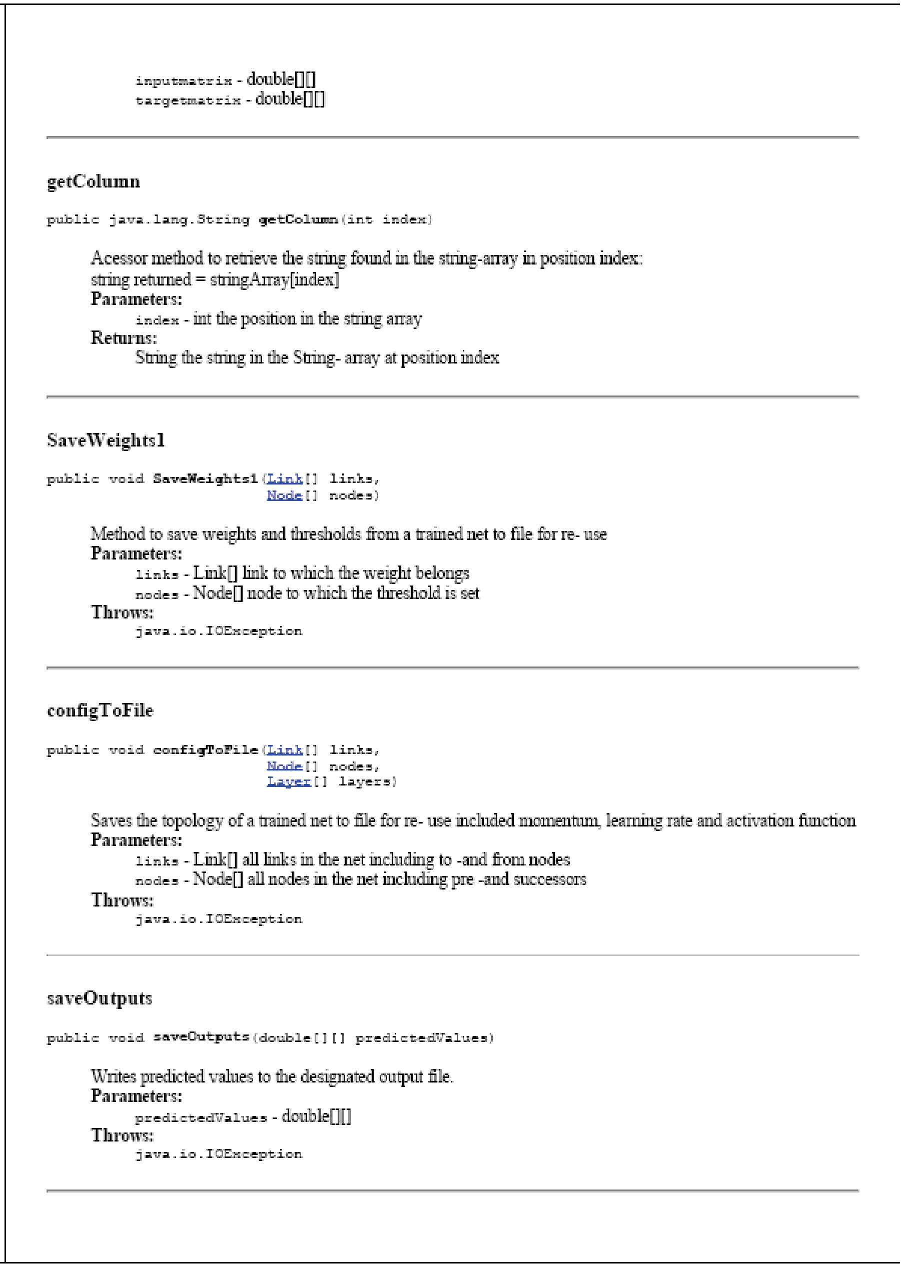
```
public int getNoofTargets()
```

Checks it this request is legal, then returns the number of targets read from file. Only used for predicative purposes, i.e. when the net has been trained and is merely used for prediction.
Returns:
int number of targets which corresponds to the number of target nodes

writeDataset

```
public void writeDataset(java.lang.String[] heading,
                         double[][] inputmatrix,
                         double[][] targetmatrix)
```

Writes (a modified) dataset to file in the comma- separated format used by
Parameters:
`heading` - String[]



`inputmatrix` - double[][]
`targetmatrix` - double[][]

getColumn

```
public java.lang.String getColumn(int index)
```

Acessor method to retrieve the string found in the string-array in position index:
string returned = stringArray[index]
Parameters:
`index` - int the position in the string array
Returns:
String the string in the String- array at position index

SaveWeights1

```
public void SaveWeights1(Link[] links,
                         Node[] nodes)
```

Method to save weights and thresholds from a trained net to file for re- use
Parameters:
`links` - Link[] link to which the weight belongs
`nodes` - Node[] node to which the threshold is set
Throws:
`java.io.IOException`

configToFile

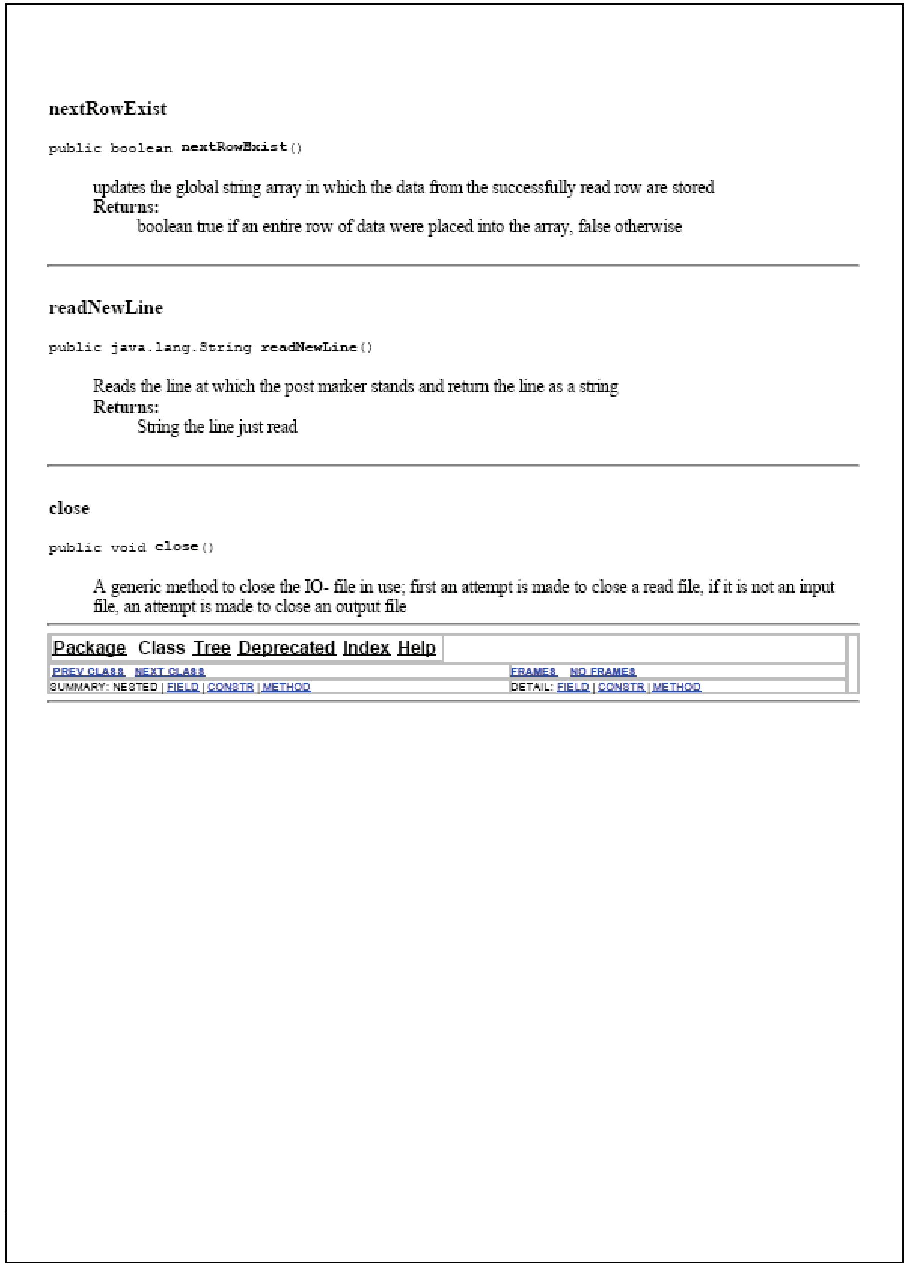
```
public void configToFile(Link[] links,
                         Node[] nodes,
                         Layer[] layers)
```

Saves the topology of a trained net to file for re- use included momentum, learning rate and activation function
Parameters:
`links` - Link[] all links in the net including to -and from nodes
`nodes` - Node[] all nodes in the net including pre -and successors
Throws:
`java.io.IOException`

saveOutputs

```
public void saveOutputs(double[][] predictedValues)
```

Writes predicted values to the designated output file.
Parameters:
`predictedValues` - double[][]
Throws:
`java.io.IOException`



nextRowExist

```
public boolean nextRowExist()
```

updates the global string array in which the data from the successfully read row are stored
Returns:
boolean true if an entire row of data were placed into the array, false otherwise

---

readNewLine

```
public java.lang.String readNewLine()
```

Reads the line at which the post marker stands and return the line as a string
Returns:
String the line just read

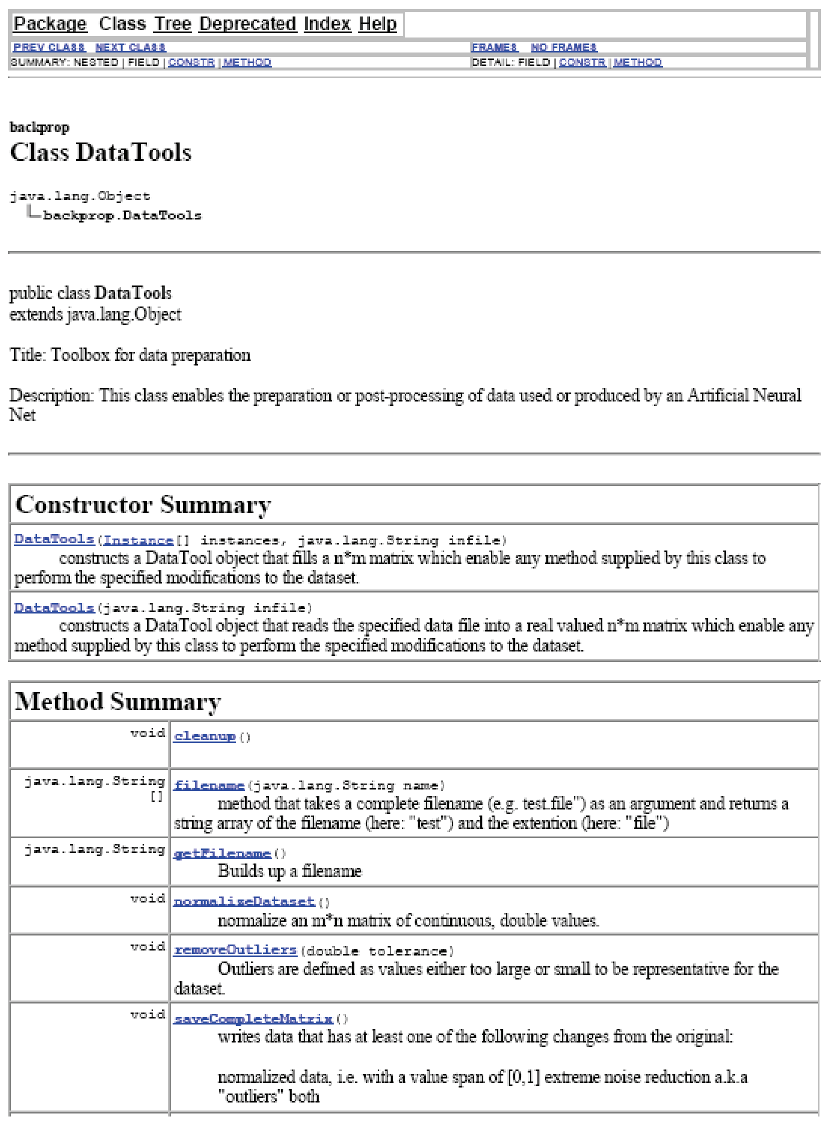
---

close

```
public void close()
```

A generic method to close the IO- file in use; first an attempt is made to close a read file, if it is not an input file, an attempt is made to close an output file

Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD



Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

backprop
Class DataTools

```
java.lang.Object
  └backprop.DataTools
```

public class **DataTools**
extends java.lang.Object

Title: Toolbox for data preparation

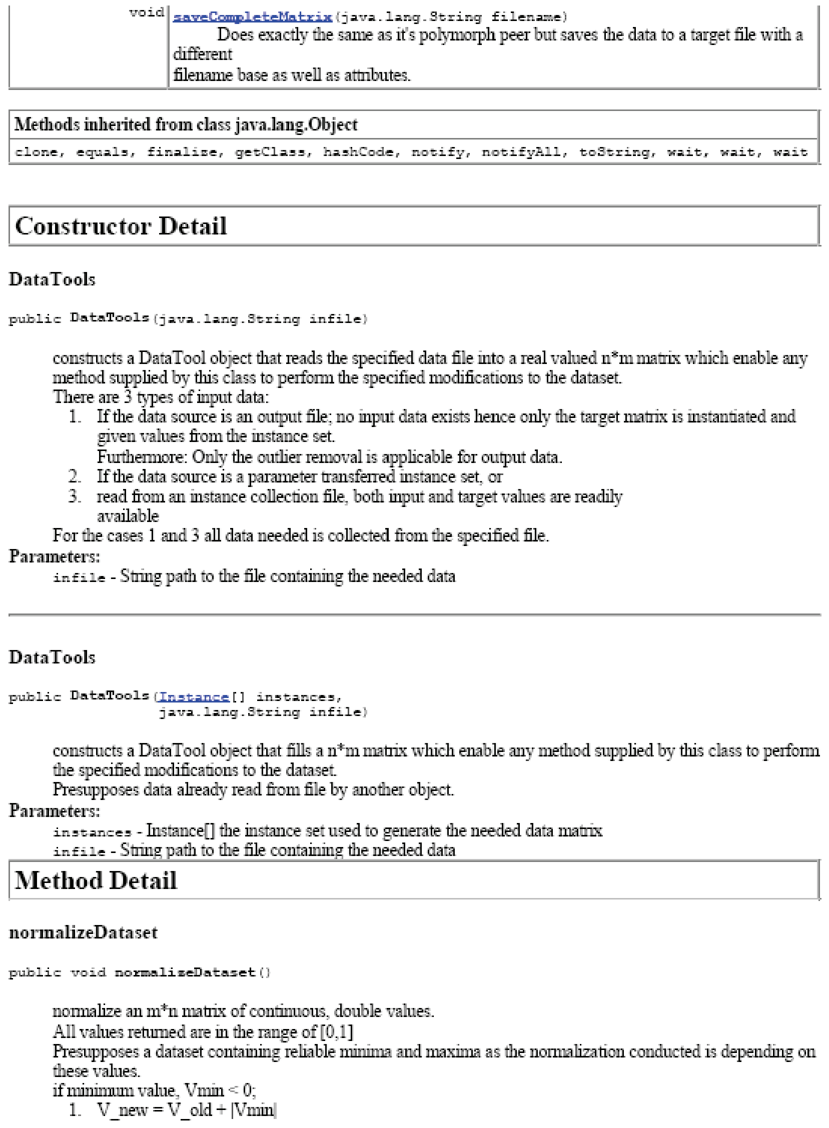
Description: This class enables the preparation or post-processing of data used or produced by an Artificial Neural Net

Constructor Summary

`DataTools(Instance[] instances, java.lang.String infile)`
constructs a DataTool object that fills a n*m matrix which enable any method supplied by this class to perform the specified modifications to the dataset.

`DataTools(java.lang.String infile)`
constructs a DataTool object that reads the specified data file into a real valued n*m matrix which enable any method supplied by this class to perform the specified modifications to the dataset.

Method Summary	
void	`cleanup()`
java.lang.String[]	`filename(java.lang.String name)` method that takes a complete filename (e.g. test.file") as an argument and returns a string array of the filename (here: "test") and the extention (here: "file")
java.lang.String	`getFilename()` Builds up a filename
void	`normalizeDataset()` normalize an m*n matrix of continuous, double values.
void	`removeOutliers(double tolerance)` Outliers are defined as values either too large or small to be representative for the dataset.
void	`saveCompleteMatrix()` writes data that has at least one of the following changes from the original: normalized data, i.e. with a value span of [0,1] extreme noise reduction a.k.a "outliers" both



void	`saveCompleteMatrix(java.lang.String filename)` Does exactly the same as it's polymorph peer but saves the data to a target file with a different filename base as well as attributes.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

DataTools

`public DataTools(java.lang.String infile)`

constructs a DataTool object that reads the specified data file into a real valued n*m matrix which enable any method supplied by this class to perform the specified modifications to the dataset.
There are 3 types of input data:
1. If the data source is an output file; no input data exists hence only the target matrix is instantiated and given values from the instance set.
   Furthermore: Only the outlier removal is applicable for output data.
2. If the data source is a parameter transferred instance set, or
3. read from an instance collection file, both input and target values are readily available

For the cases 1 and 3 all data needed is collected from the specified file.
Parameters:
`infile` - String path to the file containing the needed data

DataTools

```
public DataTools(Instance[] instances,
                 java.lang.String infile)
```

constructs a DataTool object that fills a n*m matrix which enable any method supplied by this class to perform the specified modifications to the dataset.
Presupposes data already read from file by another object.
Parameters:
`instances` - Instance[] the instance set used to generate the needed data matrix
`infile` - String path to the file containing the needed data

Method Detail

normalizeDataset

`public void normalizeDataset()`

normalize an m*n matrix of continuous, double values.
All values returned are in the range of [0,1]
Presupposes a dataset containing reliable minima and maxima as the normalization conducted is depending on these values.
if minimum value, Vmin < 0;
1. V_new = V_old + |Vmin|

2. ->Vmax=Vmax + |Vmin|
3. V_new = V_new/Vmax

if minimum value, Vmin > 0;

1. V_new = V_old - Vmin
2. ->Vmax=Vmax - Vmin
3. V_new = V_new/Vmax

Attributes with a "narrow" value span, e.g. V_attrib element_of [0.01, 0.012]
or more precisely: 0 >max_value >1 there is a need to "stretch" the attribute or expand the value span.
A rough description of the procedure follows:

- V_max = V_max + increment (if needed)
- V_new = 1/Vmax

The new data are stored in a comma- separated file specified by calling this method.

---

removeOutliers

```
public void removeOutliers(double tolerance)
```

Outliers are defined as values either too large or small to be representative for the dataset.
The technique implemented in this method selects a percentage of the data range, e.g. input to node 1 and replaces the specified percentage of

- n% of the highest values by replacing the existing values by the remaining maximum value, e.g. using 5% tolerance in a dataset of 100 instances, the new maximum value vill be that of the (1-0.05)*100 = 95'th highest attribute value. All values higher than this new maxima of the remaining 5 instances with higher attribute values vill be replaced by this new maxima
- n% of the lowest values by replacing the existing values by the remaining minimum value

the new data are stored in a comma- separated file specified by calling this method

Parameters:
`tolerance` - int the percentage of extremal values replaced by the remaining extremal values

---

saveCompleteMatrix

```
public void saveCompleteMatrix()
```

writes data that has at least one of the following changes from the original:

- normalized data, i.e. with a value span of [0,1]
- extreme noise reduction a.k.a "outliers"
- both

---

saveCompleteMatrix

```
public void saveCompleteMatrix(java.lang.String filename)
```

Does exactly the same as it's polymorph peer but saves the data to a target file with a different filename base as well as attributes.

Parameters:
`filename` - String filename different than the data source file which constitutes the basis for the complete target file name including change attributes

---

getFilename

```
public java.lang.String getFilename()
```

Builds up a filename
Returns:
String

---

filename

```
public java.lang.String[] filename(java.lang.String name)
```

method that takes a complete filename (e.g. test.file") as an argument and returns a string array of

1. the filename (here: "test")
2. and the extention (here: "file")

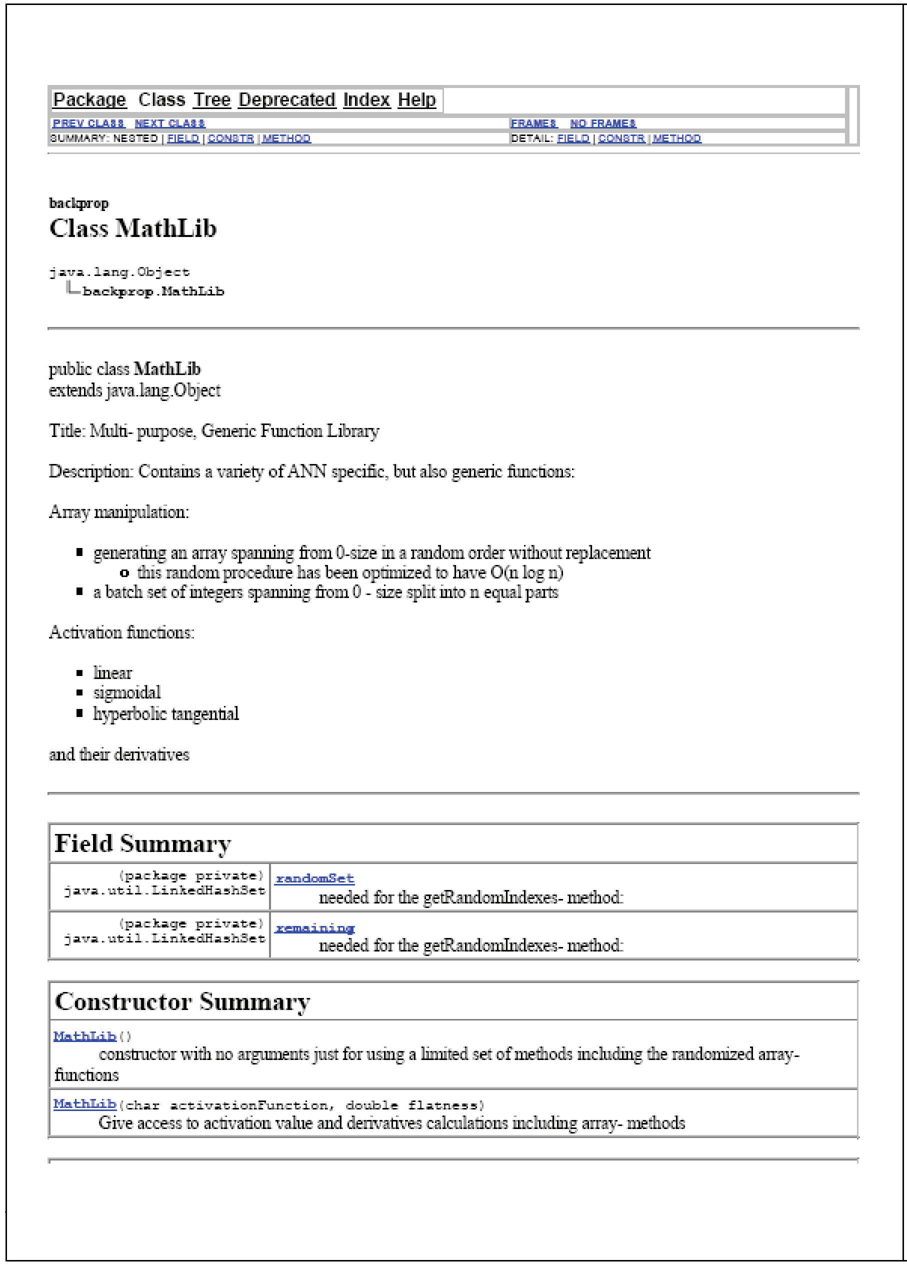
Parameters:
`name` - String the filename to split
Returns:
String[] the array containing file pre and post fix

---

cleanup

```
public void cleanup()
```

Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD



Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

backprop

Class MathLib

```
java.lang.Object
  └backprop.MathLib
```

public class **MathLib**
extends java.lang.Object

Title: Multi- purpose, Generic Function Library

Description: Contains a variety of ANN specific, but also generic functions:

Array manipulation:

- generating an array spanning from 0-size in a random order without replacement
  - this random procedure has been optimized to have O(n log n)
- a batch set of integers spanning from 0 - size split into n equal parts

Activation functions:

- linear
- sigmoidal
- hyperbolic tangential

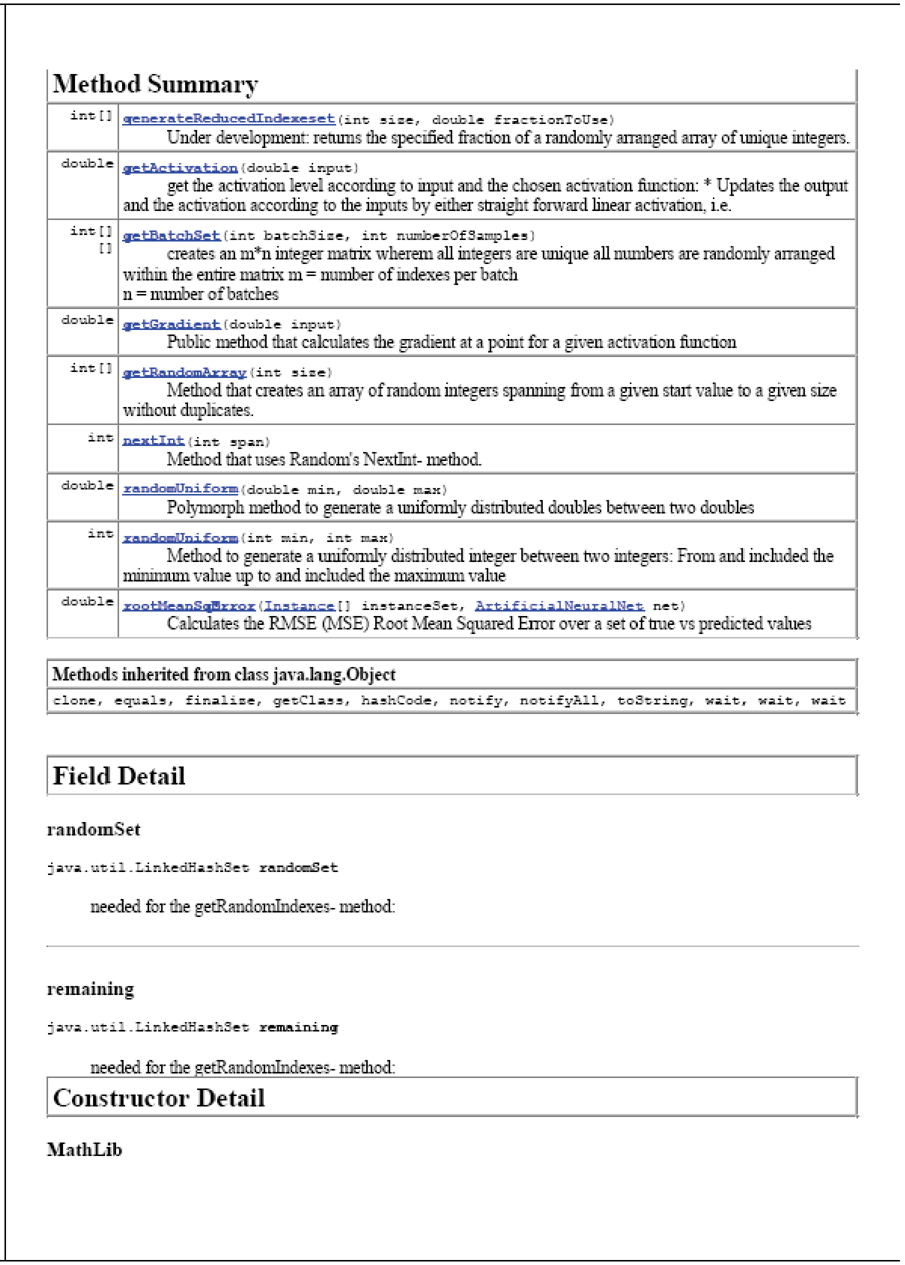
and their derivatives

Field Summary

`(package private) java.util.LinkedHashSet`	`randomSet` needed for the getRandomIndexes- method:
`(package private) java.util.LinkedHashSet`	`remaining` needed for the getRandomIndexes- method:

Constructor Summary

`MathLib()`   constructor with no arguments just for using a limited set of methods including the randomized array-functions
`MathLib(char activationFunction, double flatness)`   Give access to activation value and derivatives calculations including array- methods



Method Summary

`int[]`	`generateReducedIndexeset(int size, double fractionToUse)` Under development: returns the specified fraction of a randomly arranged array of unique integers.
`double`	`getActivation(double input)` get the activation level according to input and the chosen activation function: * Updates the output and the activation according to the inputs by either straight forward linear activation, i.e.
`int[][]`	`getBatchSet(int batchSize, int numberOfSamples)` creates an m*n integer matrix wherem all integers are unique all numbers are randomly arranged within the entire matrix m = number of indexes per batch n = number of batches
`double`	`getGradient(double input)` Public method that calculates the gradient at a point for a given activation function
`int[]`	`getRandomArray(int size)` Method that creates an array of random integers spanning from a given start value to a given size without duplicates.
`int`	`nextInt(int span)` Method that uses Random's NextInt- method.
`double`	`randomUniform(double min, double max)` Polymorph method to generate a uniformly distributed doubles between two doubles
`int`	`randomUniform(int min, int max)` Method to generate a uniformly distributed integer between two integers: From and included the minimum value up to and included the maximum value
`double`	`rootMeanSqError(Instance[] instanceSet, ArtificialNeuralNet net)` Calculates the RMSE (MSE) Root Mean Squared Error over a set of true vs predicted values

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

randomSet

`java.util.LinkedHashSet randomSet`

needed for the getRandomIndexes- method:

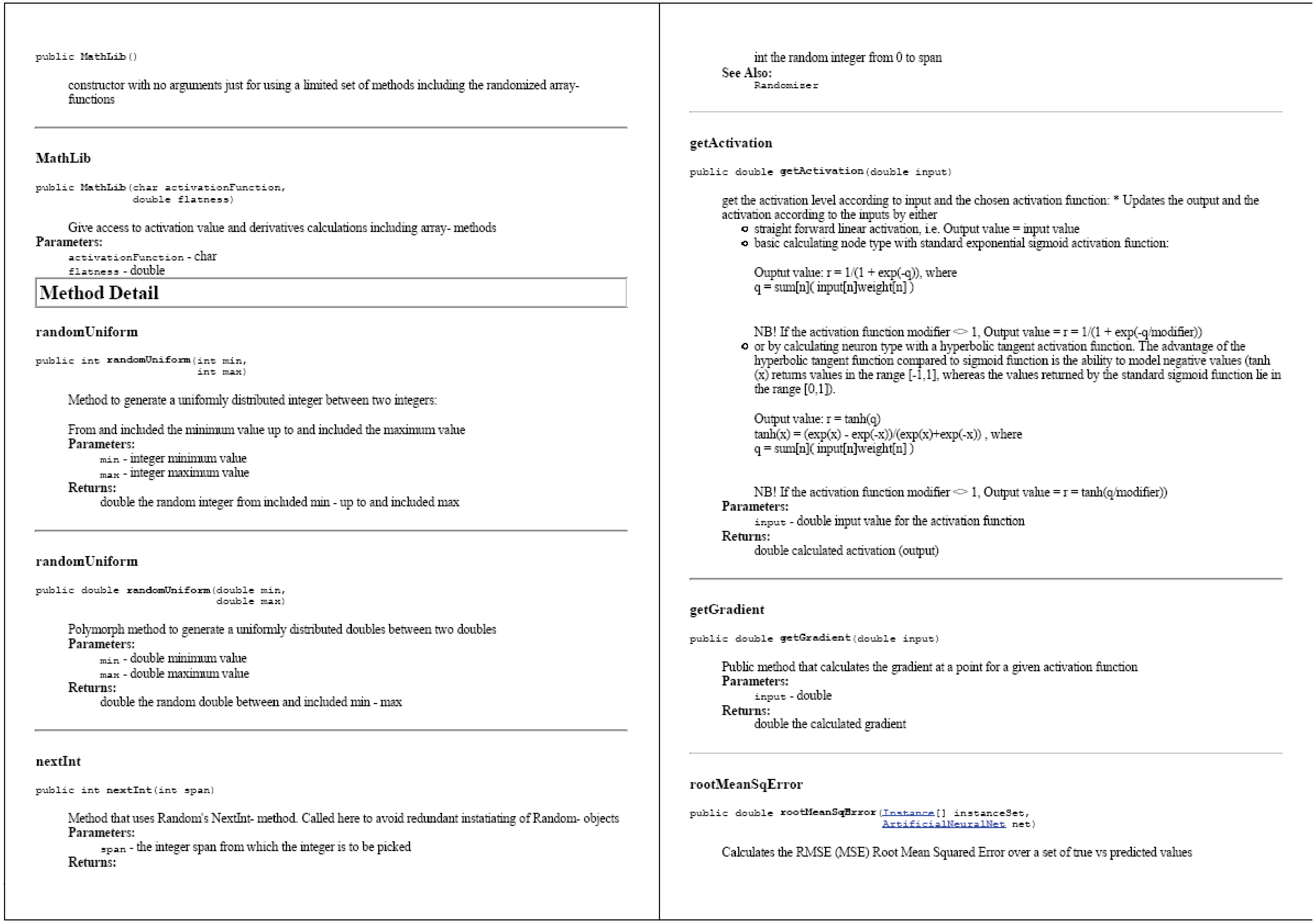
remaining

`java.util.LinkedHashSet remaining`

needed for the getRandomIndexes- method:

Constructor Detail

MathLib



`public MathLib()`

constructor with no arguments just for using a limited set of methods including the randomized array-functions

MathLib

```
public MathLib(char activationFunction,
               double flatness)
```

Give access to activation value and derivatives calculations including array- methods

Parameters:
`activationFunction` - char
`flatness` - double

Method Detail

randomUniform

```
public int randomUniform(int min,
                         int max)
```

Method to generate a uniformly distributed integer between two integers:

From and included the minimum value up to and included the maximum value

Parameters:
`min` - integer minimum value
`max` - integer maximum value

Returns:
double the random integer from included min - up to and included max

randomUniform

```
public double randomUniform(double min,
                            double max)
```

Polymorph method to generate a uniformly distributed doubles between two doubles

Parameters:
`min` - double minimum value
`max` - double maximum value

Returns:
double the random double between and included min - max

nextInt

`public int nextInt(int span)`

Method that uses Random's NextInt- method. Called here to avoid redundant instatiating of Random- objects

Parameters:
`span` - the integer span from which the integer is to be picked

Returns:
int the random integer from 0 to span

See Also:
`Randomizer`

getActivation

`public double getActivation(double input)`

get the activation level according to input and the chosen activation function: * Updates the output and the activation according to the inputs by either

- straight forward linear activation, i.e. Output value = input value
- basic calculating node type with standard exponential sigmoid activation function:

  Ouptut value: r = 1/(1 + exp(-q)), where
  q = sum[n](input[n]weight[n])

  NB! If the activation function modifier <> 1, Output value = r = 1/(1 + exp(-q/modifier))
- or by calculating neuron type with a hyperbolic tangent activation function. The advantage of the hyperbolic tangent function compared to sigmoid function is the ability to model negative values (tanh (x) returns values in the range [-1,1], whereas the values returned by the standard sigmoid function lie in the range [0,1]).

  Output value: r = tanh(q)
  tanh(x) = (exp(x) - exp(-x))/(exp(x)+exp(-x)) , where
  q = sum[n](input[n]weight[n])

  NB! If the activation function modifier <> 1, Output value = r = tanh(q/modifier))

Parameters:
`input` - double input value for the activation function

Returns:
double calculated activation (output)

getGradient

`public double getGradient(double input)`

Public method that calculates the gradient at a point for a given activation function

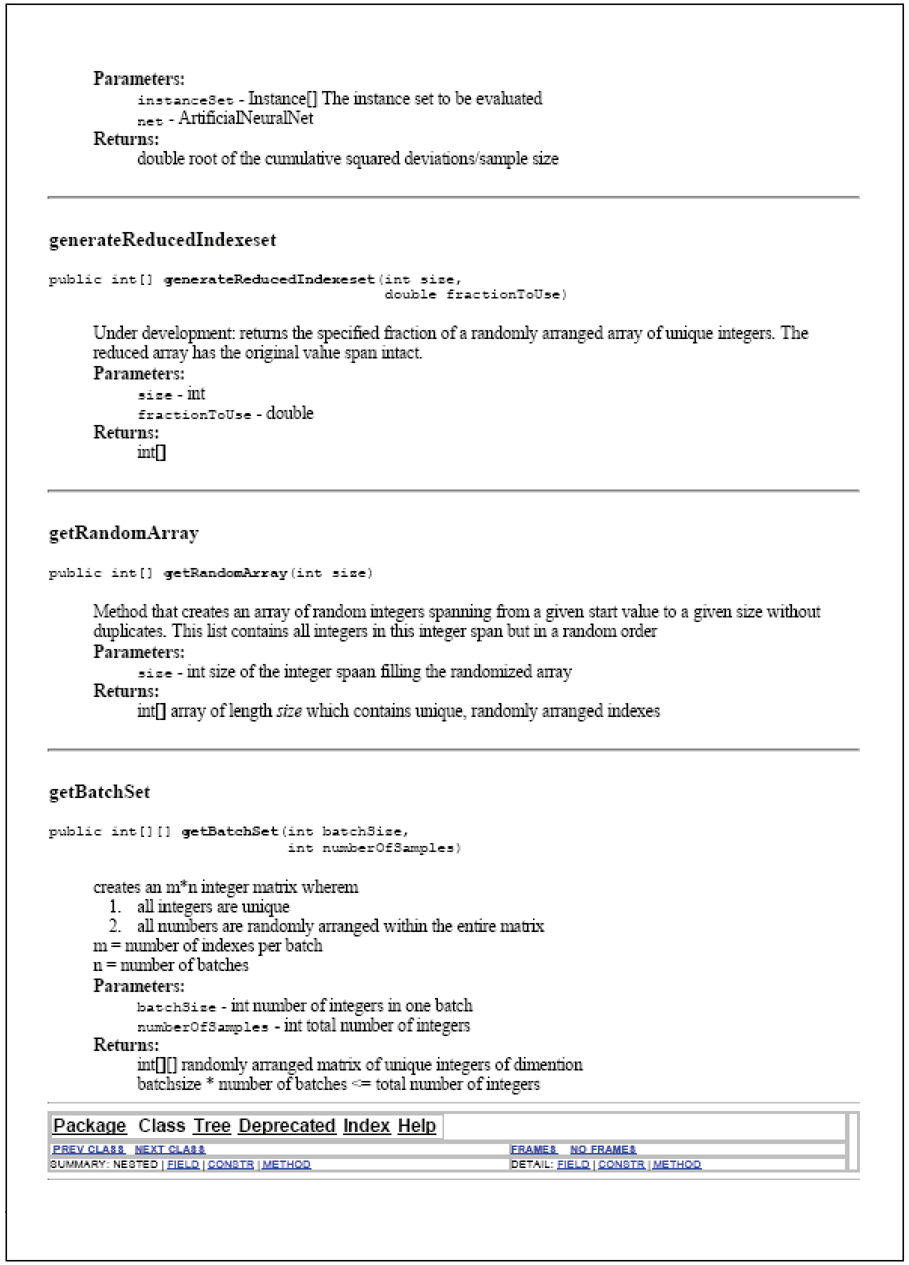
Parameters:
`input` - double

Returns:
double the calculated gradient

rootMeanSqError

```
public double rootMeanSqError(Instance[] instanceSet,
                              ArtificialNeuralNet net)
```

Calculates the RMSE (MSE) Root Mean Squared Error over a set of true vs predicted values



Parameters:
`instanceSet` - Instance[] The instance set to be evaluated
`net` - ArtificialNeuralNet
Returns:
double root of the cumulative squared deviations/sample size

---

generateReducedIndexeset

```
public int[] generateReducedIndexeset(int size,
                                      double fractionToUse)
```

Under development: returns the specified fraction of a randomly arranged array of unique integers. The reduced array has the original value span intact.
Parameters:
`size` - int
`fractionToUse` - double
Returns:
int[]

---

getRandomArray

```
public int[] getRandomArray(int size)
```

Method that creates an array of random integers spanning from a given start value to a given size without duplicates. This list contains all integers in this integer span but in a random order
Parameters:
`size` - int size of the integer spaan filling the randomized array
Returns:
int[] array of length *size* which contains unique, randomly arranged indexes

---

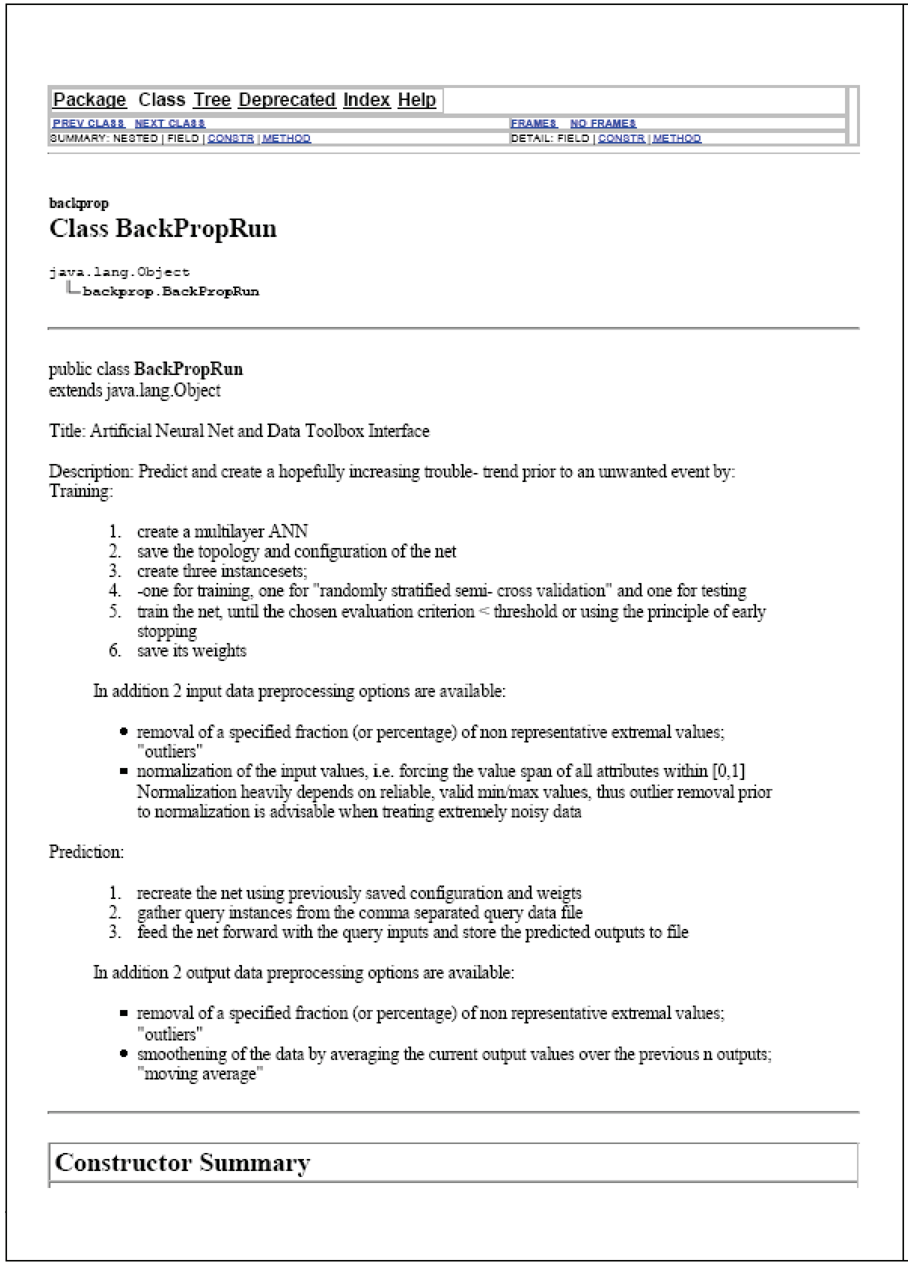
getBatchSet

```
public int[][] getBatchSet(int batchSize,
                           int numberOfSamples)
```

creates an m*n integer matrix wherem
1. all integers are unique
2. all numbers are randomly arranged within the entire matrix

m = number of indexes per batch
n = number of batches
Parameters:
`batchSize` - int number of integers in one batch
`numberOfSamples` - int total number of integers
Returns:
int[][] randomly arranged matrix of unique integers of dimention
batchsize * number of batches <= total number of integers

Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD



Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

backprop

Class BackPropRun

```
java.lang.Object
  └backprop.BackPropRun
```

public class **BackPropRun**
extends java.lang.Object

Title: Artificial Neural Net and Data Toolbox Interface

Description: Predict and create a hopefully increasing trouble- trend prior to an unwanted event by:
Training:

1. create a multilayer ANN
2. save the topology and configuration of the net
3. create three instancesets;
4. -one for training, one for "randomly stratified semi- cross validation" and one for testing
5. train the net, until the chosen evaluation criterion < threshold or using the principle of early stopping
6. save its weights

In addition 2 input data preprocessing options are available:

- removal of a specified fraction (or percentage) of non representative extremal values; "outliers"
- normalization of the input values, i.e. forcing the value span of all attributes within [0,1] Normalization heavily depends on reliable, valid min/max values, thus outlier removal prior to normalization is advisable when treating extremely noisy data

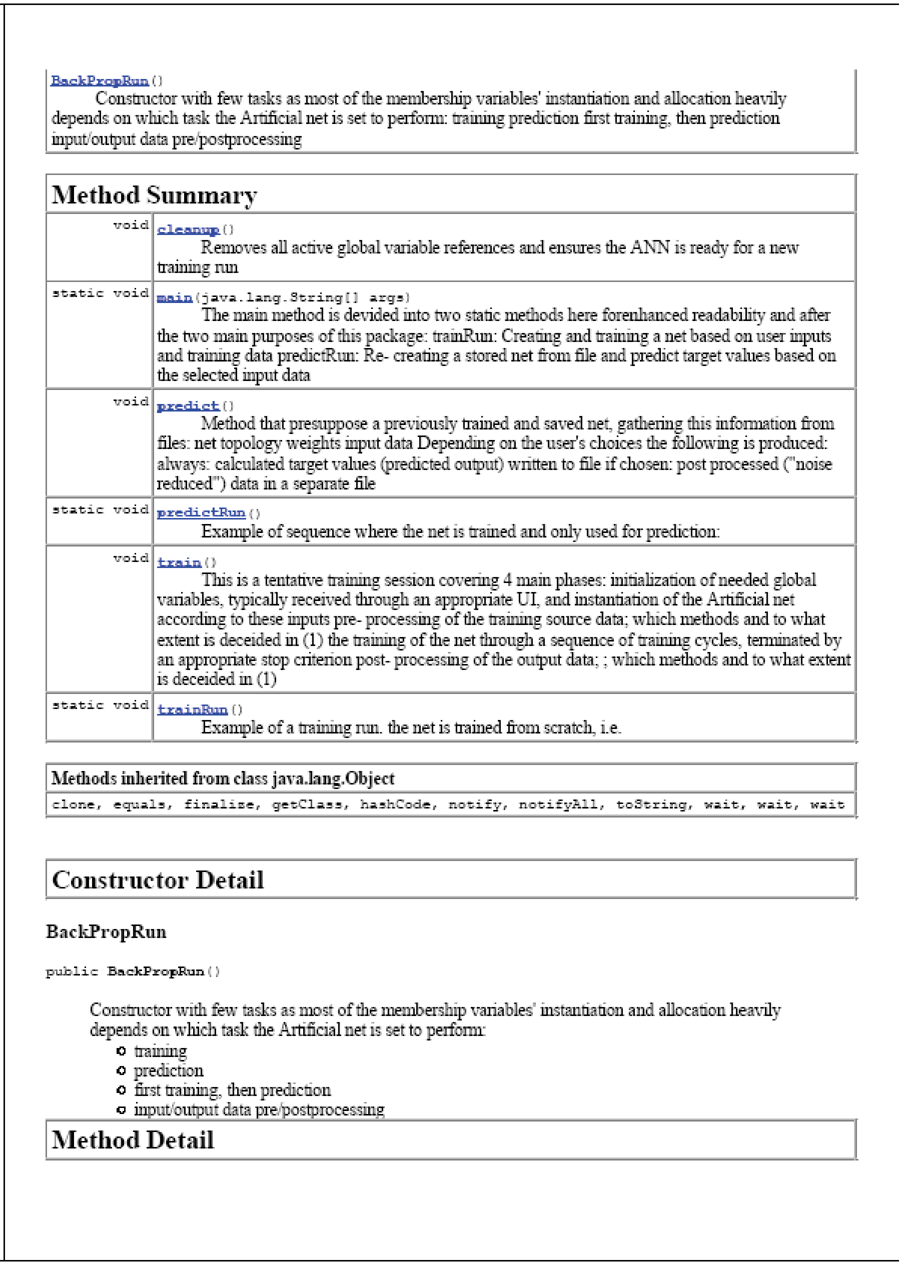
Prediction:

1. recreate the net using previously saved configuration and weigts
2. gather query instances from the comma separated query data file
3. feed the net forward with the query inputs and store the predicted outputs to file

In addition 2 output data preprocessing options are available:

- removal of a specified fraction (or percentage) of non representative extremal values; "outliers"
- smoothening of the data by averaging the current output values over the previous n outputs; "moving average"

Constructor Summary



`BackPropRun()`
Constructor with few tasks as most of the membership variables' instantiation and allocation heavily depends on which task the Artificial net is set to perform: training prediction first training, then prediction input/output data pre/postprocessing

Method Summary

`void`	`cleanup()` Removes all active global variable references and ensures the ANN is ready for a new training run
`static void`	`main(java.lang.String[] args)` The main method is devided into two static methods here forenhanced readability and after the two main purposes of this package: trainRun: Creating and training a net based on user inputs and training data predictRun: Re- creating a stored net from file and predict target values based on the selected input data
`void`	`predict()` Method that presuppose a previously trained and saved net, gathering this information from files: net topology weights input data Depending on the user's choices the following is produced: always: calculated target values (predicted output) written to file if chosen: post processed ("noise reduced") data in a separate file
`static void`	`predictRun()` Example of sequence where the net is trained and only used for prediction:
`void`	`train()` This is a tentative training session covering 4 main phases: initialization of needed global variables, typically received through an appropriate UI, and instantiation of the Artificial net according to these inputs pre- processing of the training source data; which methods and to what extent is deceided in (1) the training of the net through a sequence of training cycles, terminated by an appropriate stop criterion post- processing of the output data; ; which methods and to what extent is deceided in (1)
`static void`	`trainRun()` Example of a training run. the net is trained from scratch, i.e.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

BackPropRun

`public BackPropRun()`

Constructor with few tasks as most of the membership variables' instantiation and allocation heavily depends on which task the Artificial net is set to perform:
- training
- prediction
- first training, then prediction
- input/output data pre/postprocessing

Method Detail

Error! Objects cannot be created from editing field codes.

9.6 ANN Implementation

Attached is a code selection of classes referred. in the thesis. The code is implemented using the Java JDK 1.4 platform, but no library post 1.1 is used methods therefore the code is considered backwards compatible to Java JDK 1.1.
The following classes are attached:

1. **ArtificialNet.java**
2. **Node.java**
3. **MathLib.java**
4. **FileOps.java**
5. **DataTools.java**

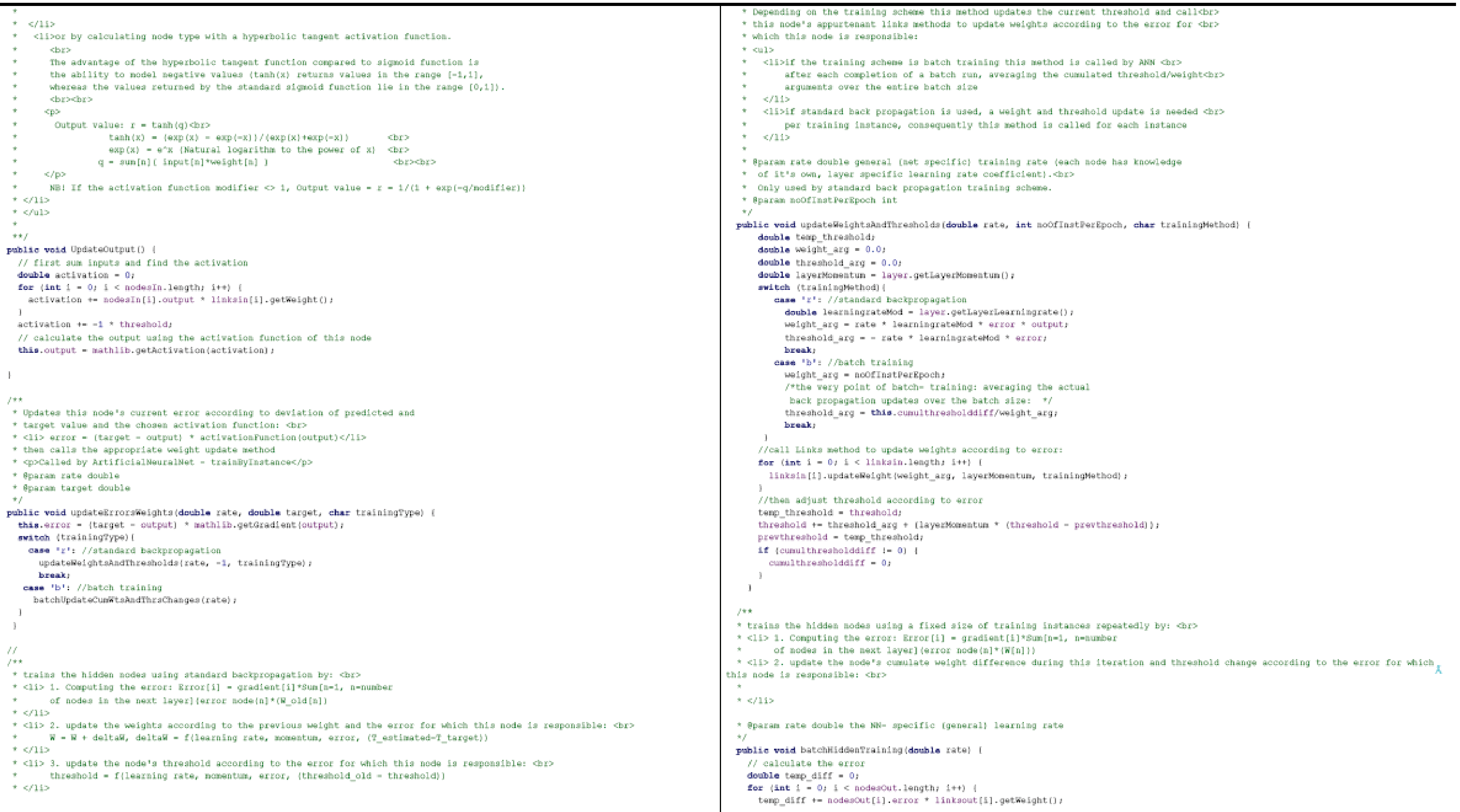
```
  * Constructor for initializing and training a new net. It creates a multilayer SBP ANN with
  * given no. of layers, no of nodes, learning rates, momentum parameters,
  * activation functions and flatness.
  * @param topology int[] e.g. [3, 2, 1, 1] means 3 input nodes, 2 nodes in the first hidden layer,
  * 1 node in the second hidden layer and one output node
  * @param learningratecoefficient double[] learningrate coefficient for each layer, which together
  * with the overall learningrate deceide the actual learning rate for each layer: e.g. learning rate = 0.1 and
  * learningratecoefficients = [1.2, 0.9,..,1.3] mean actual learning rate for the input layer = 0.9 * 1.2 = 1.08
  * @param activationFunction char[] e.g. activationFunction = [l, s, t,.., t] mean activation functions for
  *  nodes in the input layer is linear, sigmoid function is used for nodes in the first hidden layer etc.
  * @param momentumrate double[] e.g. momentumrate = [0.5,.0.8,..,0.7] mean momentum for nodes in the input
  * layer is 0.5, 0.8 for the first hidden layer etc.
  * @param activationFuncModifier double[] e.g. [1, 1.1, 0.9,..] mean default sharpness of the activation function
  * for the input layer, a smoother activation function for the first hidden layer, a sharper a.f. for the second
  * hidden layer etc.
   */
public ArtificialNeuralNet(int[] topology, double[] learningratecoefficient,
                           char[] activationFunction, double[] momentumrate,
                           double[] activationFuncModifier, MathLib mathlib,
                           char[] trainingAttribs) {
  this.mathlib = mathlib;
  this.trainingAttributes = trainingAttribs;
  //nolayers = topology.length;

  /**
   * instantiate the Layer- objects with their appurtenant membership variables:
   */
  layers = new Layer[topology.length];
  /**
   * temporary layer- object used when instantiating and adding new layer- objects
   * into the global layer- array
   */
  Layer new_layer = null;
  //first instantiate the input layer
  layers[0] = new Layer(0, topology[0]);
  //then instantiate layers containing processing elements
  for (int i = 1; i < topology.length; i++) {
    new_layer = new Layer(i, activationFuncModifier[i], activationFunction[i], momentumrate[i],
                          learningratecoefficient[i], topology[i]);
    this.layers[i] = new_layer;
  }
  /**
   * determine the number of nodes by traversing the layer- array and calculate
   * the number of nodes that constitute this net, then
   * instantiate the Node array:
   */
  int noOfNodes = 0;
  for (int i = 0; i < topology.length; i++) {
    noOfNodes += topology[i];
  }
  nodes = new Node[noOfNodes];

  /**
   * determine the number of links:
   * <li>n_links = Sum[layer=0, n-1]((nodes in layer i)*(nodes in next layer))</li>
   * then instantiate the Link- array:
   */
  int noOfLinks = 0;
  for (int i = 0; i < layers.length - 1; i++) {
    noOfLinks += topology[i] * topology[i + 1];
  }
  links = new Link[noOfLinks];
  /**
   * Now that the number of nodes is known, the Node- array is instantiated and all needed Node- data is in place: <br>
   * first instantiate the input nodes and add them to their Layer- object
   */
  int nodeNumber = 0;
  // nodes in the input layer first as they have no processing tasks:
  for (int i = 0; i < topology[0]; i++) {
    Node new_node = new Node(nodeNumber, layers[0]);
    nodes[nodeNumber] = new_node;
    layers[0].addNode(new_node);
    nodeNumber++;
  }
  /**
   * instantiate the remaining Node- objects (i.e. nodes in the hidden and output
   * layers:
```

```
   *   <li>traverse the entire Layer- array and for each layer;</li>
   *   <li>instantiate a Node- object for each node belonging to the current layer</li>
   *   <li>add the new node to it's Layer- object</li>
   * <ol>
   */
  for (int i = 1; i < topology.length; i++) {
    for (int j = 0; j < topology[i]; j++) {
      Node new_node = new Node(
          nodeNumber, layers[i]);
      nodes[nodeNumber] = new_node;
      layers[i].addNode(new_node);
      nodeNumber++;
    }
  }
  /**
   * instantiate links by
   * -instantiate a link from each node from current layer to each node in the next layer
   * -starting from the input layer to, but excluding, the output layer (which has no next layer)
   */

  int linkNo = 0;
  for (int i = 0; i < topology.length - 1; i++) {
    for (int j = 0; j < topology[i]; j++) {
      for (int k = 0; k < topology[i + 1]; k++) {
        links[linkNo++] = new Link(layers[i].getNode(j),
                                   layers[i + 1].getNode(k), mathlib);
      }
    }
  }
  //establish predecessor/successor relationships
  setNodeIOs();
}

// This method is used by constructors only.
// It determines the incoming and outgoing nodes / links for each node
// and set them in the node. This information is to be used later during feed forward and back propagation.
/**
 *each node needs knowledge about
 * <ul>
 *   <li>it's successors (except for the output layer) depending on this node's output, e.g. for cumulative input</li>
 *   <li>and it's predecessor (except for the inputlayer) depending on links origining from
 *    this node, e.g. for cumulative weighted input</li>
 */
private void setNodeIOs() {

  Node[] currentNodesin;
  Node[] currentNodesout;
  Link[] currentLinksin;
  Link[] currentLinksout;

  int occurencesIn = 0;
  int occurencesOut = 0;
  for (int i = 0; i < nodes.length; i++) {
    // first determine the dimension of the arrays
    currentNodesin = null;
    currentNodesout = null;
    occurencesIn = 0;
    occurencesOut = 0;
    for (int j = 0; j < links.length; j++) {
      if (links[j].getSourceunit() == nodes[i]) {
        occurencesOut++;
      }
      if (links[j].getTargetunit()== nodes[i]) {
        occurencesIn++;
      }
    }
    currentNodesin = new Node[occurencesIn];
    currentLinksin = new Link[occurencesIn];
    currentNodesout = new Node[occurencesOut];
    currentLinksout = new Link[occurencesOut];
    // then fill each array
    occurencesIn = 0;
    occurencesOut = 0;
    for (int j = 0; j < links.length; j++) {
      if (links[j].getSourceunit() == nodes[i]) {
        currentNodesout[occurencesOut] = links[j].getTargetunit();
        currentLinksout[occurencesOut++] = links[j];
      }
```

```java
      // set them in the node
      nodes[i].setIOrelations(currentNodesin, currentNodesout, currentLinksin,
                              currentLinksout);
    }
  }

  //
  /**
   * Saves the configuration of the net to a file
   * @param path String Path to the output file
   * @throws IOException in case the output file does not exist or the I/O= op was interrupted
   */
  public void saveConfig(String path) throws IOException {
    FileOps fileops = new FileOps(path);
    //fileops.configToFile(links, nodes);
    fileops.configToFile(links, nodes, layers);
    fileops.close();
  }

  /**
   * Method to load weights of the net from file
   * @param path String Path to the output file
   */
  public void loadAndSetWeights(String path) {
    FileOps fileRead = new FileOps(path, "Read");
    //LineReader linereader = new LineReader(path);
    double newWeight = 0.0;
    double newThreshold = 0.0;
    int linkID = 0;
    int nodeID = 0;
    while (fileRead.nextRowExist()) {
      // if it's a link weight
      if (fileRead.getColumn(0).compareTo("w") == 0) {
        newWeight = Double.parseDouble(fileRead.getColumn(2));
        linkID = Integer.parseInt(fileRead.getColumn(1));
        links[linkID].updateWeight(newWeight);
      }
      // if it's a node threshold
      if (fileRead.getColumn(0).compareTo("t") == 0) {
        newThreshold = Double.parseDouble(fileRead.getColumn(2));
        nodeID = Integer.parseInt(fileRead.getColumn(1));
        nodes[nodeID].setThreshold(newThreshold);
      }
    }
    fileRead.close();
  }

  /**
   * Method to save weights to file
   * @param path String Path to the output file
   * @throws IOException IOException in case the output file does not exist or the I/O= op was interrupted
   */

  public void SaveWeights(String path) throws IOException {
    FileOps fileops = new FileOps(path);
    fileops.SaveWeights1(links, nodes);

  }
  /**
   * This method saves the predicted output values for all output nodes after feeding
   * forward the given input values.
   * @param path String the path to the file containing the instance set; e.g. training
   * examples
   * @param instanceset InstanceSet Entire set of instances to be predicted
   * @throws IOException in case the training example file could not be read
   */
  public void savePredictedOutputs(String path, InstanceSet instanceset) throws
      IOException {
    /***
     * when asking for instance inputs for prediction, only the entire loaded
     * instance set is used as input for prediction:
     */
    char setType = 'i';
```

```java
    //the matrix of predicted targets to be written to file:
    double[][] predictedTargets = new double[inputMatrix.length][no_targets];
    for (int instanceNr = 0; instanceNr < inputMatrix.length; instanceNr++) {
      predictedTargets[instanceNr] = getOutput(inputMatrix[instanceNr]);
    }
    FileOps fileops = new FileOps(path);
    fileops.saveOutputs(predictedTargets);
    fileops.close();
  }

  /**
   * feed forward phase used both by training and prediction methods by:
   * <ol>
   *   <li>presenting input values to the input layer (i.e. layer[0])</li>
   *   <li>from the first hidden layer (if any); propagate forward up to and including the output layer</li>
   * @param inputs double[] input array
   */
  public void feedForward(double[] inputs) {
    // Input values are set to the input layer (i.e. layer[0]:
    for (int i = 0; i < layers[0].getNoOfNodes(); i++) {
      Node nodeToUpdate = layers[0].getNode(i);
      nodeToUpdate.setOutput(inputs[i]);
    }
    // from the first hidden layer (if any); propagate forward up to and including the output layer
    for (int i = 1; i < layers.length; i++) {
      // update the output of each node in this layer
      for (int j = 0; j < layers[i].getNoOfNodes(); j++) {
        Node nodeToUpdate = layers[i].getNode(j);
        nodeToUpdate.UpdateOutput();
      }
    }
  }

  /**
   * takes an array of input values, put them to the input nodes, feeds the net forward
   * and returns the outputs of the output layer
   * @param inputs double[] Inputs fed to the input layer
   * @return double[] output values for the output layer
   */
  public double[] getOutput(double[] inputs) {
    feedForward(inputs);
    //number of outputs is found in the last layer:
    Layer outputLayer = layers[layers.length - 1];
    int noOutputs = outputLayer.getNoOfNodes();
    double[] outputArray = new double[noOutputs];
    for (int i = 0; i < noOutputs; i++) {
      outputArray[i] = outputLayer.getNode(i).getOutput();
    }
    return outputArray;
  }

  /**
   * method that returns the RMSE (MSE) - Root Mean Squared error over an instance set:
   * <li> sumSquared: the squared deviations from predicted values to actual
   *      target values over all output nodes
   * </li>
   * <li> RMSE is the root of the sumSquared divided by the number of instances
   *      times the number of output nodes: RMSE = Root(sumSquared)/(noInstances*noOutputNodes)
   * </li>
   *
   * @param instances Instance[] the instance set over which RMSE is to be calculated
   * @return double the RMSE over the instance set
   */
  public double getRMSE(Instance[] instances) {
    return mathlib.rootMeanSqError(instances, this);
  }

    /**
    * a fixed number of instances, e.g. the total number of instances n/10 are randomly picked from
    * the total instance set and are presented to the net for backpropagation. This is repeated until the
    * entire set of training instances has been presented to the net
    * <br> Example: <br>
    * <li> noOfInstances = instances.length/10</li>
    * <li> for each iteration, noOfInstances are randomly selected from the training set instances and
    *      presented to the net. The standardBackPropagation - method then take each of these n/10 instances and train the net </li>
   * <li> the number of iterations needed would then be 10-11 to cover the training set giving 10-11 weight updates</li>
   * @param instances Instance[] the entire instance set used for training
   * @param rate double layer specific training rate
   * @param noOfInstances int the number of instances selected for each training iteration
   * <li> e.g. noOfInstances = instances.length/10</li>
   */
  public void nFoldBatchTraining(Instance[] instances, double rate,
                                 int batchSize) {
    Instance[] batchOfInstances;
    int[][] batchSet = mathlib.getBatchSet(batchSize, instances.length);
    int[] batchArray;
    int randomIndex = -1;
    for (int batchNo = 0; batchNo < batchSet.length; batchNo++) {
      batchArray = batchSet[batchNo];
      batchOfInstances = new Instance[batchArray.length];
      for (int i = 0; i < batchArray.length; i++) {
        randomIndex = batchArray[i];
        batchOfInstances[i] = instances[randomIndex];
      }
      standardBackPropagation(batchOfInstances, rate);
      batchArray = null;
      batchOfInstances = null;
    }
  }

  /**
   * Trains the net straight forward using the given set of instances and feed
   * the net instance by instance sequentially
   * @param instances Instance[] the set of instances to be used during
   * @param rate double layer specific learning rate
   */
  public void standardBackPropagation(Instance[] instances, double rate) {
    int noofInstances = instances.length;
    //index of randomly arranged (without replacement) indexes of length noofInstances
    int[] randomindexes = null;
    //standard back propagation, randomized instance presentation, batch training?
    char trainingType = this.trainingAttributes[1];
    int instance_index = -1;
    if (trainingType == 'r') {
      randomindexes = mathlib.getRandomArray(noofInstances);
    }
    for (int i = 0; i < noofInstances; i++) {
      switch (trainingType) {
        case 'r': //randomized instance presentation
          instance_index = randomindexes[i];
          break;
        default: //standard back propagation, batch training (instance set already randomized)
          instance_index = i;
      }
      instanceTraining(rate, instances[instance_index]);
    }
    /**
     *if batch training is used, training and weight/threshold updates occur
     *only when a batch cycle has completed, hence these values need to be propagated
     * through the layers when the standardBackPropagation is called:
     */
    if (trainingType == 'b') {
      // update weights using cumulative values obtained during batch training
      for (int i = 0; i < nodes.length; i++) {
        nodes[i].updateWeightsAndThresholds(rate, instances.length,
                                            trainingType);
      }
    }
  }
  /**
   * used by standardBackPropagation method to train the net using one selected instance
   * @param rate double net specific training rate
   * @param instance Instance instance used for training
   */
  private void instanceTraining(double rate, Instance instance) {
    double output;
    Node nodeToUpdate;
    char trainingType = this.trainingAttributes[1];
    feedForward(instance.input);
    // training last layer, i.e. the output layer first
    for (int j = 0; j < layers[layers.length - 1].getNoOfNodes(); j++) {
      output = instance.target[j];
      nodeToUpdate = layers[layers.length - 1].getNode(j);
      nodeToUpdate.updateErrorsWeights(rate, output, trainingType);
    }
    //then train the hidden layers
    for (int i = layers.length - 2; i > 0; i--) {
      for (int j = 0; j < layers[i].getNoOfNodes(); j++) {
        nodeToUpdate = layers[i].getNode(j);
        switch (trainingType) {
          case 'r': //standard back propagation
            nodeToUpdate.standardHiddenTraining(rate, trainingType);
            break;
          case 'b': //batch training, only cumulatively changing weights
            nodeToUpdate.batchHiddenTraining(rate);
            break;
        }
      }
    }
  }


  /**
   * Accessor method to the membership variable giving the number of input
   * variables presented to the net.<br>
   * Only used for predicative purposes, i.e. when the net has been trained and
   * is merely used for prediction.
   * @return int the number of input values needed
   */
  public int getNoofInputs(){
    return this.noofInputs;
  }


}
```

```java
package backprop;
/**
 *
 * <p>Title: Processing Element of the Neural Net - Data and Methods</p>
 *<p>Description: This class represents the processing elements of the Net (excluding Input nodes). It has a reference
 * to the Layer to which it belongs. The collective Layer - Node data constitute the needed parameter and data foundation
 *  for the Node's methods to adequately update it's weight and threshold during the back propagation phase.
 *</p>
 *
 * @author Inge Valaas
 * @version 1.0
 */
public class Node {
  /**
   * this node's unique ID, used when (re-)constructing a net from file or when saved to file.
   */
  private int id; //
  /**
   * the layer specific threshold value
   */
  private double threshold;
  /**
   * the layer to which this node belong, e.g. layer = 0 -> input layer
   */
   public Layer layer;
  /**
   * this node's current output, given the cumulative weighted inputs and threshold.
   */
  private double output;
  /**
   * threshold from previous run. Used during back propagation and threshold update
   */
  private double prevthreshold;

  /**
   * an array of all nodes in the next layer to which this node is connected.<br>
   * Not applicable for the output layer. Used during back propagation.
   */
  private Node[] nodesOut;
  /**
   * an array of all nodes in the previous layer to which this node is connected and <br>
   * thus take inputs from. Used during feed forward.
   */
  private Node[] nodesIn;
  /**
   * an array of outgoing links (containing e.g. weights) to nodes in previous
   * layer <br> to which this node is connected thus take this node's output.
   * Used during back propagation
   */
  private Link[] linksout;
  /**
   * an array of incoming links (containing e.g. weights) to nodes in previous
   * layer <br> to which this node is connected, and from which this node takes outputs.
   * Used during feed forward.
   */
  private Link[] linksin; // array of links from which this node takes outputs. To be used during feedforward
  /**
   * error = (target - output) * activationFunction(output). Used during back propagation.
   */
  private double error;
  /**
   * cumulate changes in threshold during batch training
   */
  private double cumulthresholddiff;
  /**
   * @see MathLib
   */
  private MathLib mathlib;

   /**
   * Constructor for initializing, creating and training a new SBP ANN or for recreating
   * a saved net from file
   * @param id int <br> <i>This node's internal ID </i>
   * @param layer int <br> <i> The layer to which this node belongs (e.g. layer = 0 -> input layer)</i>
   * @param activationFuncModifier double <br> <i> "Flatness" of the activation function, 1 is default </i>
   * @param activationFuncType char <br> <i> e.g. "t" -> hyperbolic tangent activation function </i>
   * @param learningratecoefficient double <br> <i> i.e.: if the learning rate is .1 and this value is
   * 1.5, actual learning rate will be .1 * 1.5 = .15</i>
 * <br> <i> </i>
 */
public Node (int id, Layer layer) {
  this.id = id;
  this.layer = layer;
  /*if it is a node belonging to the input layer, it has no processing
    tasks and initialization ends here. */
  if (layer.getID() != 0) {
    output = 0.0;
    char activationType = layer.getActivationFuncType();
    double sharpness = layer.getActivationFuncModifier();
    this.mathlib = new MathLib(activationType, sharpness);
    this.threshold = mathlib.randomUniform( (double) - 1, (double) 1);
    this.prevthreshold = threshold;
    this.cumulthresholddiff = 0;
  }
}

/**
 * this method constructs nodein and nodeout arrays in order to determine the relationships of this node to others.
 * should be called during the construction of the net
 * @param nodesIn Node[]  an array of all nodes in the previous layer to which this node is connected
 * @param nodesOut Node[]   an array of all nodes in the next layer to which this node is connected
 * @param linksin Link[]   an array of outgoing links (containing e.g. weights) to nodes in previous
 * layer to which this node is connected
 * @param linksout Link[]   an array of incoming links (containing e.g. weights) to nodes in next
 * layer to which this node is connected
 */
public void setIOrelations(Node[] nodesIn, Node[] nodesOut, Link[] linksin,
                    Link[] linksout) {

  this.nodesIn = nodesIn;
  this.nodesOut = nodesOut;
  this.linksin = linksin;
  this.linksout = linksout;
}

/**
 * accessor method to the node's ID
 * @return int
 */
public int getID(){
  return id;
}
/**
 * accessor method to the node's current threshold value
 * @return double the current threshold value for this node
 */
public double getThreshold(){
  return threshold;
}
/**
 * updates the field threshold
 * @param newThreshold double the new threshold value
 */
public void setThreshold(double newThreshold){
  this.threshold = newThreshold;
}
/**
 * accessor method to the layer to which this node belongs, e.g. getLayer() = 0 -> input layer
 * @return int
 */
public Layer getLayer(){
  return layer;
}
/**
 * accessor method to the node's output value. The value == the node's activation which is equal<br>
 * to the cumulative weighted input - threshold
 * @return double this node's current output value (or activation)
 */
public double getOutput(){
  return output;
}
/**
 * updates the field output: Only done during the feed forward
 * phase for the <i>input layer </i>(layer = 0)
 * @param newOutput double the new output value
 */
public void setOutput(double newOutput){
```



```
 *
 *  </li>
 *   <li>or by calculating node type with a hyperbolic tangent activation function.
 *      <br>
 *      The advantage of the hyperbolic tangent function compared to sigmoid function is
 *      the ability to model negative values (tanh(x) returns values in the range [-1,1],
 *      whereas the values returned by the standard sigmoid function lie in the range [0,1]).
 *      <br><br>
 *     <p>
 *      Output value: r = tanh(q)<br>
 *                tanh(x) = (exp(x) - exp(-x))/(exp(x)+exp(-x))        <br>
 *                exp(x) = e^x (Natural logarithm to the power of x)   <br>
 *              q = sum[n]( input[n]*weight[n] )                        <br><br>
 *     </p>
 *      NB! If the activation function modifier <> 1, Output value = r = 1/(1 + exp(-q/modifier))
 * </li>
 * </ul>
 *
 **/
public void UpdateOutput() {
  // first sum inputs and find the activation
  double activation = 0;
  for (int i = 0; i < nodesIn.length; i++) {
    activation += nodesIn[i].output * linksin[i].getWeight();
  }
  activation += -1 * threshold;
  // calculate the output using the activation function of this node
  this.output = mathlib.getActivation(activation);

}

/**
 * Updates this node's current error according to deviation of predicted and
 * target value and the chosen activation function: <br>
 * <li> error = (target - output) * activationFunction(output)</li>
 * then calls the appropriate weight update method
 * <p>Called by ArtificialNeuralNet - trainByInstance</p>
 * @param rate double
 * @param target double
 */
public void updateErrorsWeights(double rate, double target, char trainingType) {
  this.error = (target - output) * mathlib.getGradient(output);
  switch (trainingType){
    case 'r': //standard backpropagation
      updateWeightsAndThresholds(rate, -1, trainingType);
      break;
   case 'b': //batch training
     batchUpdateCumWtsAndThrsChanges(rate);
  }
 }

//
/**
 * trains the hidden nodes using standard backpropagation by: <br>
 * <li> 1. Computing the error: Error[i] = gradient[i]*Sum[n=1, n=number
 *      of nodes in the next layer](error node(n)*(W_old[n])
 * </li>
 * <li> 2. update the weights according to the previous weight and the error for which this node is responsible: <br>
 *      W = W + deltaW, deltaW = f(learning rate, momentum, error, (T_estimated-T_target))
 * </li>
 * <li> 3. update the node's threshold according to the error for which this node is responsible: <br>
 *      threshold = f(learning rate, momentum, error, (threshold_old - threshold))
 * </li>
```

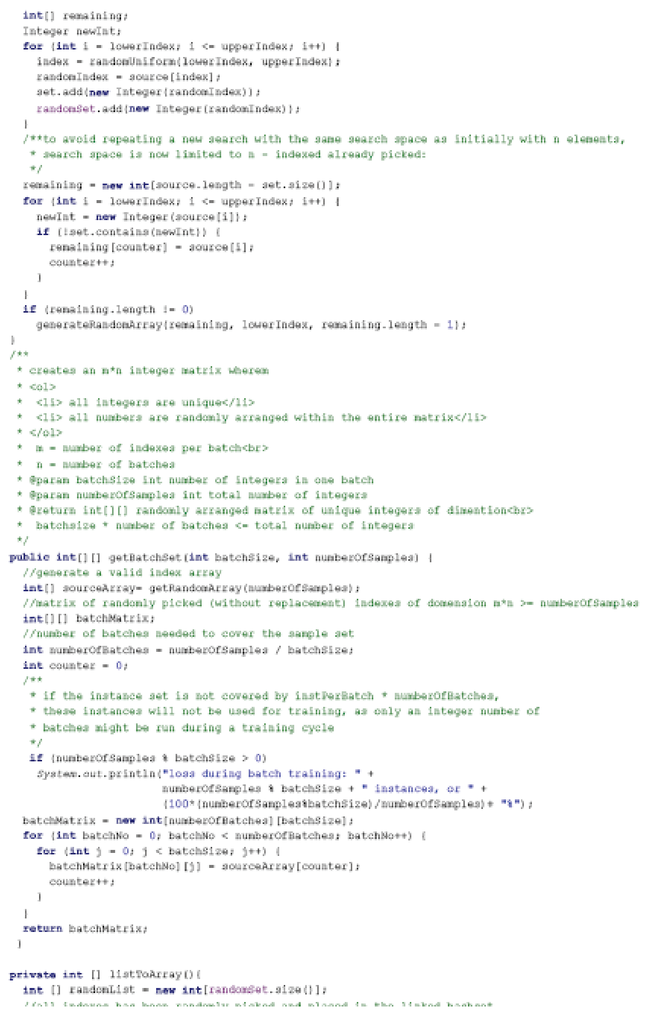
```
 * Depending on the training scheme this method updates the current threshold and call<br>
 * this node's appurtenant links methods to update weights according to the error for <br>
 * which this node is responsible:
 * <ul>
 *   <li>if the training scheme is batch training this method is called by ANN <br>
 *       after each completion of a batch run, averaging the cumulated threshold/weight<br>
 *       arguments over the entire batch size
 *   </li>
 *   <li>if standard back propagation is used, a weight and threshold update is needed <br>
 *       per training instance, consequently this method is called for each instance
 *   </li>
 *
 * @param rate double general (net specific) training rate (each node has knowledge
 *  of it's own, layer specific learning rate coefficient).<br>
 *  Only used by standard back propagation training scheme.
 * @param noOfInstPerEpoch int
 */
public void updateWeightsAndThresholds(double rate, int noOfInstPerEpoch, char trainingMethod) {
    double temp_threshold;
    double weight_arg = 0.0;
    double threshold_arg = 0.0;
    double layerMomentum = layer.getLayerMomentum();
    switch (trainingMethod){
      case 'r': //standard backpropagation
        double learningrateMod = layer.getLayerLearningrate();
        weight_arg = rate * learningrateMod * error * output;
        threshold_arg = - rate * learningrateMod * error;
        break;
      case 'b': //batch training
        weight_arg = noOfInstPerEpoch;
        /*the very point of batch- training: averaging the actual
         back propagation updates over the batch size:  */
        threshold_arg = this.cumulthresholddiff/weight_arg;
        break;
     }
    //call Links method to update weights according to error:
    for (int i = 0; i < linksin.length; i++) {
      linksin[i].updateWeight(weight_arg, layerMomentum, trainingMethod);
    }
    //then adjust threshold according to error
    temp_threshold = threshold;
    threshold += threshold_arg + (layerMomentum * (threshold - prevthreshold));
    prevthreshold = temp_threshold;
    if (cumulthresholddiff != 0) {
      cumulthresholddiff = 0;
    }
  }

 /**
 * trains the hidden nodes using a fixed size of training instances repeatedly by: <br>
 * <li> 1. Computing the error: Error[i] = gradient[i]*Sum[n=1, n=number
 *      of nodes in the next layer](error node(n)*(W[n]))
 * <li> 2. update the node's cumulate weight difference during this iteration and threshold change according to the error for which
this node is responsible: <br>
 *
 * </li>

 * @param rate double the NN- specific (general) learning rate
 */
 public void batchHiddenTraining(double rate) {
   // calculate the error
   double temp_diff = 0;
   for (int i = 0; i < nodesOut.length; i++) {
     temp_diff += nodesOut[i].error * linksout[i].getWeight();
```

```
      }
      this.error = temp_diff * mathlib.getGradient(output);
      //update weights and thresholds accordingly
      batchUpdateCumWtsAndThrsChanges(rate);
    }

    /**
     * cumulated threshold and weight differences are updated according to the error for
     * which this node is responsible and is updated for<i> each instance</i>
     * @param rate double
     */
    private void batchUpdateCumWtsAndThrsChanges(double rate) {
      double weight_increment = 0;
      double layerLearningrate = layer.getLayerLearningrate();
      for (int i = 0; i < linksin.length; i++) {
        //call each link in's method to update it's cumulative weight difference during this batch
        weight_increment = rate * layerLearningrate * error * output;
        linksin[i].updateCumulWtDiff(weight_increment);
      }
      //then update this node's cumulative threshold difference during this batch run
      cumulthresholddiff += rate * layerLearningrate * error * -1;
    }

  }
```

```java
package backprop;
/**
 * @author Inge Valaas
 * @Url http://www.idi.ntnu.no/~ingeasmu
 * @version 1.0.8
 */

import java.util.Random;
import java.util.LinkedHashSet;
import java.util.Arrays;
/**
 * <p>Title: Multi- purpose, Generic Function Library</p>
 * <p>Description: Contains a variety of ANN specific, but also generic functions:</p>
* Array manipulation:
* <ul>
*  <li>generating an array spanning from 0=size in a random order without replacement</li>
*  <ul>
*    <li>this random procedure has been optimized to have O(n log n)</li>
*  </ul>
*  <li>a batch set of integers spanning from 0 - size split into n equal parts</li>
* </ul>
* Activation functions:
* <ul>
*  <li>linear</li>
*  <li>sigmoidal</li>
*  <li>hyperbolic tangential</li>
* </ul>
* and their derivatives
*/

public class MathLib {
  /**
   * Needed to deceide wheter to use sigmoidal, linear or hyperbolic tangent activation functions and their derivatives
   */
  private char activationFunction;
  /**
   * Layer specific modificator to the activation function
   */
  private double flatness;
  /**
   * needed for the getRandomIndexes- method:
   */
  LinkedHashSet randomSet;
  /**
   * needed for the getRandomIndexes- method:
   */
  LinkedHashSet remaining;
  private Random rnd;

  /**
   * constructor with no arguments just for using a limited set of methods including the
   * randomized array- functions
   */
  public MathLib(){
    rnd = new Random();
    randomSet = new LinkedHashSet();
  }
  /**
   *Give access to activation value and derivatives calculations including
   * array- methods
   * @param activationFunction char
   * @param flatness double
   */
  public MathLib(char activationFunction, double flatness) {
  this.activationFunction = activationFunction;
  this.flatness = flatness;

  }

  /**
   * Method to generate a uniformly distributed integer between two integers: <p>
   * From and included the minimum value up to and included the maximum value
   * @param min integer minimum value
   * @param max integer maximum value
   * @return double the random integer from included min - up to and included max
   */
  public int randomUniform (int min, int max) {
    double randomDouble = randomUniform((double)min, (double)max);
    int randomInt = (int) Math.round(randomDouble);
    return randomInt;
  }


 /**
 * Polymorph method to generate a uniformly distributed doubles between two doubles
 * @param min double minimum value
 * @param max double maximum value
 * @return double the random double between and included min - max
 */
public double randomUniform (double min, double max) {
  if (rnd==null)
    rnd = new Random();
  return (rnd.nextDouble() * (max - min)) + min;
}

/**
 * Method that uses Random's NextInt- method. Called here to avoid redundant
 * instatiating of Random- objects
 * @param span the integer span from which the integer is to be picked
 * @return int the random integer from 0 to span
 * @see Randomizer
 */
public int nextInt (int span) {
    if (rnd==null)
     rnd = new Random();
    return rnd.nextInt(span);
  }

 /**
  * get the activation level according to input and the chosen activation function:
  * *  Updates the output and the activation according to the inputs by either
* <ul>
*  <li> straight forward linear activation, i.e. Output value = input value
*  </li>
* </ul>
  <ul>
       <li>basic calculating node type with standard
                 exponential sigmoid activation function:
                 <p>Ouptut value: r = 1/(1 + exp(-q)), where
          <br>     q = sum[n]( input[n]weight[n] )
       </p>
       </li>
       <br>NB! If the activation function modifier &lt;&gt; 1, Output value = r = 1/(1 + exp(-q/modifier))
  </ul>
       <ul>
        <li>or by calculating neuron type with a hyperbolic tangent activation function.
       The advantage of the hyperbolic tangent function compared to sigmoid function is
            the ability to model negative values (tanh(x) returns values in the range [-1,1],
              whereas the values returned by the standard sigmoid function lie in the range [0,1]).
              <p>Output value: r = tanh(q)<br>tanh(x) = (exp(x) - exp(-x))/(exp(x)+exp(-x)) , where
                      <br>q = sum[n]( input[n]weight[n] )
               </p>
        </li>
        <br>NB! If the activation function modifier &lt;&gt; 1, Output value = r = tanh(q/modifier))
       </ul>

  *
  * @param input double input value for the activation function
  * @return double calculated activation (output)
  */
 public double getActivation(double input){
 double output = 0.0;
   switch (activationFunction) {
                 case 's': // sigmoid
                         output = sigmoid(input);
                         break;
                 case 't': // hyperbolic tangent (tanh)
                         output = tanh(input);
                         break;
                 case 'l': // linear
                         output = activationFunction;
                         break;
             }
             return output;
}

/**
```

```
*   "S" shaped activation function implementing the exponential sigmoid function:
*          S = 1/(1 + exp(-x))
*
*   @param  x  input used by sigmoid function.
*   @return The value of the sigmoid function at x.
**/
private double sigmoid( double  input) {
        return (1 / (1 + Math.exp( -input/flatness)));
}

  /**
     *  Returns the hyperbolic tangent of the specified argument
     *
     *  The hyperbolic tangent is defined as:  <br>
     *      tanh(x) = sinh(x)/cosh(x) = 1 - 2/(exp(2*x) + 1)  <br>
     *
     *  @param  x  Value to determine the hyperbolic tangent of.
     *  @return Value of the hyperbolic tangent function at the point x.
     **/
     private double tanh(double input) {

         if (Double.isNaN(input))          return Double.NaN;
         if (input == 0)         return 0;
         if (input < -5)
                         return -1;
                 else if (input > 5)
                         return 1;

         double z = Math.abs(input);
         double s = Math.exp(2*z/flatness);
         z = 1.0 - 2.0/(s + 1.0);
         if (input < 0)
             z = -z;

         return z;
         }

/**
* Public method that calculates the gradient at a point for a given activation function
* @param input double
* @return double the calculated gradient
*/
public double getGradient (double input) {
  double temp_gradient;
  switch (activationFunction) {
    case 'g': // sigmoid
      temp_gradient = sigmoidDerivative(input) ;
      break;
    case 't': // hyperbolic tangent
      temp_gradient = tanhDerivative(input);
      break;
    case 'l': // linear
      temp_gradient = 1;
      break;
    default:
      temp_gradient= 0;
      break;
  }
  return temp_gradient;
}

  /**
   *  Returns the sigmoidal gradient of this node or node's activation
   *  function with respect to the weighted node's inputs.
   *  This function returns:
   *           dr/dq = r*(1 - r).
```

```
 *
 *  @return Value of the derivative of hyperbolic tangent function at the point fValue.
 **/
private double tanhDerivative(double input) {
  double fTemp = tanh(input);
  double fRc = 1F - fTemp*fTemp;
  return fRc;
}
/**
 * Calculates the RMSE (MSE) Root Mean Squared Error over a set of true vs predicted values
 * @param instanceSet Instance[] The instance set to be evaluated
 * @param net ArtificialNeuralNet
 * @return double root of the cumulative squared deviations/sample size
 */
public double rootMeanSqError(Instance [] instanceSet, ArtificialNeuralNet net){
   double[] outputArray;
   //find the number of output values (and correspondingly; the number of output nodes):
   int noOutputs = instanceSet[0].target.length;
   double sumSquared = 0.0;
   double rootMeanSquared = 1000.0;
   for (int j = 0; j < instanceSet.length; j++) {
     outputArray = net.getOutput(instanceSet[j].input);
     for (int i = 0; i < noOutputs; i++) {
       sumSquared +=
           Math.pow((outputArray[i] - instanceSet[j].target[i]), 2);
     }
   }
   rootMeanSquared = Math.sqrt(sumSquared)/(instanceSet.length*noOutputs);

   return rootMeanSquared;
 }
 /**
  * Under development: returns the specified fraction of a randomly arranged array
  * of unique integers. The reduced array has the original value span intact.
  * @param size int
  * @param fractionToUse double
  * @return int[]
  */
 public int[] generateReducedIndexeset(int size, double fractionToUse){
 int[] fullRandomArray = this.getRandomArray(size);
 int reducedArraySize = (int)Math.round(size*fractionToUse);
 int[] reducedRandomArray = new int[reducedArraySize];
 for (int i = 0; i < reducedRandomArray.length; i++){
   reducedRandomArray[i] = fullRandomArray[i];
 }
 return reducedRandomArray;
}
 /**
  * Method that creates an array of random integers spanning from a given start value
 * to a given size without duplicates. This list contains all integers in this integer span
 * but in a random order
  * @param size int size of the integer span filling the randomized array
  * @return int[] array of length <i>size</i> which contains unique, randomly arranged indexes
  */
 public int[] getRandomArray(int size){
  //generate a valid index array
  int[] sourceArray = new int[size];
  for (int i = 0; i < sourceArray.length; i++) {
    sourceArray[i] = i;
  }
  generateRandomArray(sourceArray, 0, size - 1);
  int[] dummy = this.listToArray();
  return dummy;
}

/**
```



```
    int[] remaining;
    Integer newInt;
    for (int i = lowerIndex; i <= upperIndex; i++) {
      index = randomUniform(lowerIndex, upperIndex);
      randomIndex = source[index];
      set.add(new Integer(randomIndex));
      randomSet.add(new Integer(randomIndex));
    }
    /**to avoid repeating a new search with the same search space as initially with n elements,
     * search space is now limited to n - indexed already picked:
     */
    remaining = new int[source.length - set.size()];
    for (int i = lowerIndex; i <= upperIndex; i++) {
      newInt = new Integer(source[i]);
      if (!set.contains(newInt)) {
        remaining[counter] = source[i];
        counter++;
      }
    }
    if (remaining.length != 0)
      generateRandomArray(remaining, lowerIndex, remaining.length - 1);
  }
  /**
   * creates an m*n integer matrix wherem
   * <ol>
   *  <li> all integers are unique</li>
   *  <li> all numbers are randomly arranged within the entire matrix</li>
   * </ol>
   *  m = number of indexes per batch<br>
   *  n = number of batches
   * @param batchSize int number of integers in one batch
   * @param numberOfSamples int total number of integers
   * @return int[][] randomly arranged matrix of unique integers of dimention<br>
   *  batchsize * number of batches <= total number of integers
   */
  public int[][] getBatchSet(int batchSize, int numberOfSamples) {
    //generate a valid index array
    int[] sourceArray= getRandomArray(numberOfSamples);
    //matrix of randomly picked (without replacement) indexes of domension m*n >= numberOfSamples
    int[][] batchMatrix;
    //number of batches needed to cover the sample set
    int numberOfBatches = numberOfSamples / batchSize;
    int counter = 0;
    /**
     * if the instance set is not covered by instPerBatch * numberOfBatches,
     * these instances will not be used for training, as only an integer number of
     * batches might be run during a training cycle
     */
     if (numberOfSamples % batchSize > 0)
      System.out.println("loss during batch training: " +
                         numberOfSamples % batchSize + " instances, or " +
                         (100*(numberOfSamples%batchSize)/numberOfSamples)+ "%");
    batchMatrix = new int[numberOfBatches][batchSize];
    for (int batchNo = 0; batchNo < numberOfBatches; batchNo++) {
      for (int j = 0; j < batchSize; j++) {
        batchMatrix[batchNo][j] = sourceArray[counter];
        counter++;
      }
    }
    return batchMatrix;
   }

  private int [] listToArray(){
    int [] randomList = new int[randomSet.size()];
    //all indexes has been randomly picked and placed in the linked hashset
```

```java
package backprop;
/**
 * @author Inge Valaas
 * @Url http://www.idi.ntnu.no/~ingeasmu
 * @version 1.0.4
 */

import java.io.*;

/**
 * <p>Title: File Input/Output Operations Toolbox</p>
 * <p>Description: Various methods to either collect information from file or write data to file:<br>
 * Reading operations include (but is not limited to):
 * <ul>
 *   <li>necessary methods to loading the topology and weights of a previously trained and saved net</li>
 *   <li>methods to load and generate entire instance sets from an input file</li>
 * </ul>
 * Writing operations include:
 * <ul>
 *   <li>writing the net topology and weights to a file</li>
 *   <li>writing datasets; calculated targets (predicted outputs), modified input data and modified output data</li>
 * </ul>
 * </p>
 */
public class FileOps {
  private String IOFile;
  private FileWriter outStream;
  private BufferedInputStream inStream;
  private String[] column;
  int[] IOquantities;



  /**
   * Constructor used for writing to a file
   * @param file String
   */
 public FileOps(String file) {
    this.IOFile = file;
  try {
    File outputFile = new File(IOFile);
    outStream = new FileWriter(outputFile);
  }
  catch (IOException e) {}
 }
 /**
  * Constructor for reading a file
  * @param path String
  */
 public FileOps (String file, String fileOp) {
   try {
     this.IOFile = file;
     this.IOquantities = new int[2];
     FileInputStream inputFile = new FileInputStream(IOFile);
     inStream = new BufferedInputStream(inputFile);

   }
   catch (IOException e) {}
 }
 /**
  *Traverses the input file and performs the following:
  * <ol>
  * <li><p>
  *  <ul>
  *    <li>traverse the first row to find the number of attributes,
  *        i.e. the number of inputs and targets</li>
  *    <li>traverses the file and find the number of numeric datarows;
  *        i.e. the number of instances</li>
  *  </ul>
  * </li></p>
  *  <li><p>
  *  <ul>
  *    <li>reset the file</li>
  *    <li>traverses the file and place the data into the instance set</li>
  *  </ul>
  * </li></p>
  * </ol>
  *
  * @param inputNodes int
  * @param outputNodes int
  * @return Instance[]
  */
public Instance [] getInstances(int inputNodes){
  Instance[] instances;
  int IOnodes = 0;
  int outputNodes = 0;
  int counter = 0;
  boolean IOset = false;
  double temp_double;
  //variables for progress reporting purposes
  boolean reportProgress = false;
  //number of instances regarded as a limit when reporting progress is necessary
  int reportProgressLimit = 10000;
  int reportProgressAt = 0;
  int reportProgressCount = 1;
  double reportProgressNext = 1.0;
  /**
   * first traverse the file to count the number of attributes (variable names) to
   * deceide the total of input and output nodes.
   */
  while (nextRowExist()) {
    try {
      //if not already done: read the first line and find the number of attributes
      if (!IOset) {
        IOnodes = column.length;
        outputNodes = IOnodes - inputNodes;
        setIOquantities(inputNodes, outputNodes);
        IOset = true;
      }
      for (int i = 0; i < (IOnodes); i++) {
        temp_double = Double.parseDouble(column[i]);
      }
      counter++;
      //User progress reporting. Might be dropped!
      if (counter > reportProgressLimit/2 && (double)counter / (double)(reportProgressLimit * 0.2 * reportProgressCount) == 1.0){
        System.out.println(counter + " instances read");
        reportProgressCount++;
      }
    }
    catch (NumberFormatException e) {}
  }
  instances = new Instance[counter];
  if (counter > reportProgressLimit) {
    reportProgress = true;
    reportProgressAt = counter / 10;
  }
  //reset the file:
  try {
    inStream.close();
  }
  catch (IOException e) {}
  // then traverse the input file again
  try {
    FileInputStream inputFile = new FileInputStream(IOFile);
    inStream = new BufferedInputStream(inputFile);
  }
  catch (IOException e) {}
  //fill the Instance- arrays with data:
  double[] new_inputArray = new double[inputNodes];
  double[] new_targettArray = new double[outputNodes];
  counter = 0;
  while (nextRowExist()) {
    try { // count numeric data only
      for (int i = 0; i < inputNodes; i++) {
        new_inputArray[i] = Double.parseDouble(column[i]);
      }
      for (int i = inputNodes; i < inputNodes + outputNodes; i++) {
        new_targettArray[i - inputNodes] = Double.parseDouble(column[i]);
      }
      instances[counter++] = new Instance(new_inputArray, new_targettArray);
    }
    catch (NumberFormatException e) {}
    /*if another 10'th of the instances has been read and reporting progress
     is applicable, report progress: */
    if (reportProgress){
      reportProgressNext = (double) counter / (double) (reportProgressAt * reportProgressCount);
      if (reportProgressNext == 1.0){
```

```
          reportProgressCount++;
        }
      }
    }
    try {
      inStream.close();
    }
    catch (IOException e) {}
    return instances;
  }

  /**
   * Checks it this request is legal, then returns the number of inputs
   * read from file
   * Only used for predicative purposes, i.e. when the net has been trained and
   * is merely used for prediction.
   * @return int number of inputs which corresponds to the number of input nodes
   */
  public int getNoofInputs() {
    int inputs = 0;
    if (inStream != null) {
      inputs = this.IOquantities[0];
    }
    else {
      throw new IllegalArgumentException();
    }
    return inputs;
  }
  /**
   * Checks it this request is legal, then returns the number of targets
   * read from file.
   * Only used for predicative purposes, i.e. when the net has been trained and
   * is merely used for prediction.
   * @return int number of targets which corresponds to the number of target nodes
   */
  public int getNoofTargets(){
    int targets = 0;
    if (inStream != null){
      targets = this.IOquantities[1];
    }
    else {
      throw new IllegalArgumentException();
    }
    return targets;
  }

  /**
   * Writes (a modified) dataset to file in the comma- separated format used by
   * @param heading String[]
   * @param inputmatrix double[][]
   * @param targetmatrix double[][]
   */
  public void writeDataset(String[] heading, double[][] inputmatrix, double[][] targetmatrix){
    int matrixLength = 0;
    if (inputmatrix != null)
      matrixLength = inputmatrix.length;
    else
      matrixLength = targetmatrix.length;
   //if at least one datamatrix has been transferred:
    if (matrixLength != 0) {
      try {
        for (int i = 0; i < heading.length; i++) {
          outStream.write(heading[i] + "; ");
        }
        outStream.write("\n");
        for (int i = 0; i < matrixLength; i++) {
          if (inputmatrix != null){
            for (int j = 0; j < inputmatrix[0].length; j++) {
              outStream.write(inputmatrix[i][j] + "; ");
            }
          }
          if (targetmatrix != null){
            for (int j = 0; j < targetmatrix[0].length; j++) {
              outStream.write(targetmatrix[i][j] + "; ");
            }
          }
          outStream.write("\n");
```

```
        outStream.close();
      }
      catch (IOException e) {}
    }
    else {
      //if no matrices containing data has been transferred:
      throw new IllegalArgumentException();
    }
  }

  /**
   * Acessor method to retrieve the string found in the string-array in position index: <br>
   * string returned = stringArray[index]
   * @param index int the position in the string array
   * @return String the string in the String- array at position index
   */
  public String getColumn(int index){
   return this.column[index];
 }
  /**
   *Method to save weights and thresholds from a trained net to file for re- use
   * @param links Link[] link to which the weight belongs
   * @param nodes Node[] node to which the threshold is set
   * @throws IOException
   */
  public void SaveWeights1 (Link [] links, Node [] nodes)  {

    // first write weight of each link
    try{
      for (int i = 0; i < links.length; i++) {
        outStream.write("w; " + i + "; " + links[i].getWeight() + "\n");
      }
      outStream.write("\n");
      // then threshold of each node
      for (int i = 0; i < nodes.length; i++) {
        outStream.write("t; " + i + "; " + nodes[i].getThreshold() + "\n");
      }
      outStream.close();
    } catch (IOException e) {}
  }
  /**
   * Saves the topology of a trained net to file for re- use included momentum, learning rate
   * and activation function
   * @param links Link[] all links in the net including to -and from nodes
   * @param nodes Node[] all nodes in the net including pre -and successors
   * @throws IOException
   */

   public void configToFile(Link[] links, Node[] nodes, Layer[] layers) {
     try {
       Node source_node = null;
       Node target_node = null;
       Layer layer_to_check = null;
       outStream.write("// Input units:\n");
       //read and place data into the layers
       outStream.write("#layers;" + layers.length + "\n");
       outStream.write(
           "// type;ID;actFuncMod;actFunc;momentum;learningrate;noOfNodes\n");
       for (int i = 0; i < layers.length; i++) {
         outStream.write("l;" + layers[i].getID() + ";" +
                         layers[i].getActivationFuncModifier() + ";" +
                         layers[i].getActivationFuncType() + ";" +
                         layers[i].getLayerMomentum() + ";" +
                         layers[i].getLayerLearningrate() + ";" +
                         layers[i].getNoOfNodes() + "\n");
       }

       //read and place data into the nodes
       outStream.write("#nodes;" + nodes.length + "\n");
       outStream.write(
           "// type;ID;layer\n");
       for (int i = 0; i < nodes.length; i++) {
         layer_to_check = nodes[i].getLayer();
         if (layer_to_check.getID() == 0 || layer_to_check == null) {
           outStream.write("i;" + i + ";0\n");
         }
         else {
           layer_to_check = nodes[i].getLayer();
```

```
          outStream.write("n;" + i + ";" + layer_to_check.getID() + "\n");
        }
      }
      //read and place data into the links
      outStream.write("#links;" + links.length + "\n");
      outStream.write("// type; ID; sourceunit; targetunit\n");
      for (int i = 0; i < links.length; i++) {
        source_node = links[i].getSourceunit();
        target_node = links[i].getTargetunit();
        outStream.write("s;" + i + ";" + source_node.getID() + ";" +
                        target_node.getID() + "\n");
      }
      outStream.close();
    }
    catch (IOException e) {
      System.out.println("Could not save net configuration");
    }
  }
  /**
   *Writes predicted values to the designated output file.<br>
   * @param predictedValues double[][]
   * @throws IOException
   */
  /**
  *Writes predicted values to the designated output file.<br>
  * @param predictedValues double[][]
  * @throws IOException
  */
 public void saveOutputs(double[][] predictedValues) {
   try {outStream.write("// Predicted output:\n");
     for (int i_no = 0; i_no < predictedValues.length; i_no++) {
       for (int p_val = 0; p_val < predictedValues[0].length; p_val++) {
         outStream.write(predictedValues[i_no][p_val] + ";");
       }
       outStream.write("\n");
     }
     outStream.close();
   }
    catch (IOException e) {System.out.println("Could not write to "
                      + IOFile + ", cause: " + e);}
 }


   /**
    * updates the global string array in which the data from the successfully
    * read row are stored
    * @return boolean true if an entire row of data were placed into the array, false otherwise
    */
   public boolean nextRowExist () {
     column = null;
     column = readNewLine().split(";");
     if (column[0] != "#EOF#") {
       for (int i = 0; i < column.length; i++) {
         column[i] = column[i].trim();
       }
       return true;
     }
     else {
       return false;
     }
   }
   /**
    * Reads the line at which the post marker stands and return the line as a string
    * @return String the line just read
    */
   public String readNewLine() {
     int i;
     char[] temp_array = new char[50000];
     char[] temp_array2;
     boolean last_line;
     int counter;
     String temp_line = "";

     do {
       temp_array2 = null;
       counter = 0;
       last_line = true;
       // read a line
       try {
         while ( (i = inStream.read()) != -1) {
           last_line = false;
           if (i == 13 || i == 10) {
             break;
           }
           else if (i != 10 && i != 13) {
             temp_array[counter++] = (char) i;
           }
         }
       }
       catch (IOException e) {
         System.out.println("Problems during readNewLine");
       }
       // put the array into a string
       if (last_line) {
         temp_line = "#EOF#";
       }
       else if (counter != 0) {
         temp_array2 = new char[counter];
         boolean all_spaces = true;
         for (int j = 0; j < counter; j++) {
           if (temp_array[j] != ' ') {
             all_spaces = false;
           }
           temp_array2[j] = temp_array[j];
         }
         if (all_spaces) {
           temp_line = "";
         }
         else {
           temp_line = new String(temp_array2);
         }
         if (temp_line.length() >= 2 && temp_line.charAt(0) == '/' &&
             temp_line.charAt(1) == '/') {
           temp_line = "";
         }
       }
       else {
         temp_line = "";
       }

     }
     while (temp_line == "");
     return temp_line.trim();
   }

 /**
  * A generic method to close the IO- file in use; first an attempt is made to close
  * a read file, if it is not an input file, an attempt is made to close an output file
  */
 public void close() {
   try {
     if (inStream != null) { //i.e. it is an inStream
       inStream.close();
     }
     else { //it is an outstream
       outStream.close();
     }
   }
   catch (IOException e) {
     System.out.println("Could not close " + IOFile);
   }
 }


private void setIOquantities(int inputs, int targets) throws IllegalArgumentException{
  if (inputs + targets !=0){
    this.IOquantities[0] = inputs;
    this.IOquantities[1] = targets;
  } else{
    //net consist of no nodes
    throw new IllegalArgumentException();
  }
 }



}
```

```
package backprop;
import java.util.*;

/**
 *
 * <p>Title: Toolbox for data preparation</p>
 * <p>Description: This class enables the preparation or post-processing of data
 * used or produced by an Artificial Neural Net
 * </p>
 * @author Inge Valaas
 * @version 1.0
 */
public class DataTools {
  private Instance[] instances;
  /**
   * the inputmatrix is generated by either
   * <ol type="a">
   *      <li>parameter transfer if called by an object already posessing the source data</li>
   *      <li>reading the data from file</li>
   *</ol>

   */
  private double[][] inputmatrix;
  /**
 * the targetmatrix is generated by either
 * <ol type="a">
 *      <li>parameter transfer if called by an object already posessing the source data</li>
 *      <li>reading the data from file</li>
 *</ol>

 */
private double[][] targetmatrix;

  /**
   * path to the file containing data (if required)
   */
  private String infile;
  /**
   * used to write attribute names to the modified data file
   */
  private String[] heading;
  /**
   * array of modificator attributes being set for each modification step being<br>
   * performed on the data, e.g.:
   * <ul>
   *   <li>attr[0]='n' => data set normalized</li>
   *   <li>attr[1]='o' => data set has the specified fraction of min/max outliers removed</li>
   * </ul>
   * The "changesPerformed" attribute array is initialized
   * to give attr[0] = attr[1] = ...attr[n] = 'z'
   */
  private char[] changesPerformed;

  /**
   * constructs a DataTool object that reads the specified data file into a real valued
   * n*m matrix which enable any method supplied by this class to perform the specified
   * modifications to the dataset.<br>
   * There are 3 types of input data:<br>
   * <ol>
   *  <li>If the data source is an output file; no input data exists hence only the
   *      target matrix is instantiated and given values from the instance set.<br>
   *      Furthermore: Only the outlier removal is applicable for output data.</li>
   *  <li>If the data source is a parameter transferred instance set, or</li>
   *  <li>read from an instance collection file, both input and target values are readily<br>
   *      available</li>
   * </ol>
   * For the cases 1 and 3 all data needed is collected from the specified file.
   * @param infile String path to the file containing the needed data
   */
  public DataTools(String infile){
   this.infile = infile;
   constructObject();
  }

  /** constructs a DataTool object that fills a
   * n*m matrix which enable any method supplied by this class to perform the specified
   * modifications to the dataset.<br>
   * Presupposes data already read from file by another object.
   * @param instances Instance[] the instance set used to generate the needed data matrix
   * @param infile String path to the file containing the needed data
   */
  public DataTools(Instance[] instances, String infile){
   this.instances = instances;
   this.infile = infile;
   constructObject();
  }

 private void constructObject(){
   FileOps reader = new FileOps(infile, "read");
  this.heading = reader.readNewLine().split(";");
  reader.close();
  //first, check if instances has been parameter transferred:
  if (this.instances == null){
    reader = new FileOps(infile, "read");
    /* to generate the instance set, the total number of input and output nodes is needed,
     * their relative quantities is unimportant      */
    int numberofInOuts = heading.length;
  //instantiate the membership variable instances
    instances = reader.getInstances(numberofInOuts - 1);
  }
  this.infile = infile;
  this.changesPerformed = new char[3];
  this.inputmatrix = null;
  this.targetmatrix = null;
  int noInstances = -1;
  int noTargets = -1;
  int noInputs;
  //reset all datatool changesPerformed to zero:
  for (int i = 0; i < changesPerformed.length; i++){
    changesPerformed[i] = 'z';
  }
  //number of input and output nodes needed to generate the instance- matrix
  noInputs = instances[0].input.length;
  noTargets = instances[0].target.length;
  noInstances = instances.length;
  if(noInputs > 0)
    inputmatrix = new double[noInstances][noInputs];
  if(noTargets > 0)
    targetmatrix = new double[noInstances][noTargets];
  /**
   * Set values to the input/output matrixes as needed:
   */
  for (int i = 0; i < noInstances; i++) {
    if (inputmatrix != null){
      for (int j = 0; j < noInputs; j++) {
        inputmatrix[i][j] = instances[i].input[j];
      }
    }
    if (targetmatrix != null){
      for (int j = 0; j < noTargets; j++) {
        targetmatrix[i][j] = instances[i].target[j];
      }
    }
  }

 }
 /**
  * normalize an m*n matrix of continuous, double values. <br>
  * All values returned are in the range of [0,1] <br>
  * Presupposes a dataset containing reliable minima and maxima as
  * the normalization conducted is depending on these values.<br>
  * if minimum value, Vmin < 0;
  * <ol>
  *    <li>V_new = V_old + |Vmin| </li>
  *    <li>->Vmax=Vmax + |Vmin|</li>
  *    <li>V_new = V_new/Vmax</li>
  * </ol>
  * if minimum value, Vmin > 0;
  *   <ol>
  *    <li>V_new = V_old - Vmin </li>
  *    <li>->Vmax=Vmax - Vmin</li>
  *    <li>V_new = V_new/Vmax</li>
  * </ol>
  * <p>
  * Attributes with a "narrow" value span, e.g. V_attrib element_of [0.01, 0.012]<br>
  * or more precisely:  0 >max_value >1 there is a need to "stretch" the attribute or expand the value span.<br>
  * A rough description of the procedure follows:
```

```java
 * <ul>
 *   <li>V_max = V_max + increment (if needed)</li>
 *   <li>V_new = 1/Vmax</li>
 * </ul>
 *</p>
 * The new data are stored in a comma- separated file specified by calling this method.
 *
 * @param instances Instance[] the value matrix to be normalized
 * @param path String The target file to which the new data are to be written
 */
public void normalizeDataset(){
  int noInputs = instances[0].input.length;
  int noTargets = instances[0].target.length;
  int noAttribsToTraverse = 0;
  //the number of attributes the outlier removal process is to traverse through the instance set
  double[][] matrixToTraverse = null;
  /*check to see if the dataset to modify is inputs or targets.
       if it is an input matrix; initialize the necessary variables: */
  if (noInputs != 0) {
    matrixToTraverse = inputmatrix;
    noAttribsToTraverse = noInputs;
  }
  /*if the dataset to modify is an output matrix; initialize the
   needed variables: */
  else if (noTargets != 0) {
    matrixToTraverse = targetmatrix;
    noAttribsToTraverse = noTargets;
  }
 //new minimum/maximum values representative for the attribute are initialized:
 double maximum = -100000.0;
 double minimum = 100000.0;
 for (int attribute_no = 0; attribute_no < noAttribsToTraverse; attribute_no++){
    //increment to ensure values start from zero:
   minimum = minValue(attribute_no, matrixToTraverse);
   maximum = maxValue(attribute_no, matrixToTraverse);
   //check if there is a need for value span reduction
   if (maximum > 1 || maximum < 1|| minimum < 0 ){
     zeroToOne(minimum, maximum, attribute_no, matrixToTraverse);
   }
 }
 //set modificator attribute to "normalized":
 changesPerformed[0] = 'n';
}

/**
 * Outliers are defined as values either too large or small to be representative for the dataset.<br>
 * The technique implemented in this method selects a percentage of the data range, e.g. input to
 * node 1 and replaces the specified percentage of
 * <ul>
 *    <li>n% of the highest values by replacing the existing values by the remaining maximum value, e.g.<br>
 *        using 5% tolerance in a dataset of 100 instances, the new maximum value will be that of the <br>
 *        (1-0.05)*100 = 95'th highest attribute value. All values higher than this new maxima of the <br>
 *        remaining 5 instances with higher attribute values will be replaced by this new maxima </li>
 *    <li>n% of the lowest values by replacing the existing values by the remaining minimum value </li>
 * </ul>

 *
 * the new data are stored in a comma- separated file specified by calling this method
 * @param tolerance int the percentage of extremal values replaced by the remaining extremal values
 * @param path String The target file to which the new data are to be written

 */
public void removeOutliers(double tolerance) {
  //test to check if tolerance has been given in % or a fraction:
  if (tolerance >= 1.0)
    tolerance /= 100;
  int noInstances = instances.length;
  int noInputs = instances[0].input.length;
  int noTargets = instances[0].target.length;
  //the number of attributes the outlier removal process is to traverse through the instance set
  int noAttribsToTraverse = 0;
  double[][] matrixToTraverse = null;
  //highest valid value in e.g. a 5% tolerance dataset
  int highestValidIndex = (int) Math.round( (1.0 - tolerance) * noInstances);
  //lowest valid value in e.g. a 5% tolerance dataset
  int lowestValidIndex = (int) Math.round( (tolerance) * noInstances);
  //new minimum/maximum values representative for the (n-tolerance) attribute are initialized:
```

```java
  double min_within_tolerance = 100000.0;
  double[] testArray;

  /*check to see if the dataset to modify is inputs or targets.
       if it is an input matrix; initialize the necessary variables: */
  if (noInputs != 0) {
    matrixToTraverse = inputmatrix;
    noAttribsToTraverse = noInputs;
  }
  /*if the dataset to modify is an output matrix; initialize the
   needed variables: */
  else if (noTargets != 0) {
    matrixToTraverse = targetmatrix;
    noAttribsToTraverse = noTargets;
    this.heading[0] = "output, outliers removed";
  }
  //if data to modify were found; remove the specified fraction of extremal values:
  if (matrixToTraverse != null) {
    /* traverse each input attribute column in the inputmatrix: */
    for (int attrib_no = 0; attrib_no < noAttribsToTraverse; attrib_no++) {
      testArray = sortedArray(attrib_no, matrixToTraverse);
       /*check if there is a need for outlyer removal (always true if the method
       is called for output noise reduction) */
      if (noInputs == 0 || (testArray[testArray.length - 1] > 1) || (testArray[0] < 0)) {
        min_within_tolerance = testArray[lowestValidIndex];
        max_within_tolerance = testArray[highestValidIndex];
       /*now traverse the attribute column and replace the outliers with the new
             maximum/minimum values */
      for (int i = 0; i < matrixToTraverse.length; i++) {
        //if the current attribute value is above the new max, replace by new max
        if (matrixToTraverse[i][attrib_no] > max_within_tolerance)
          matrixToTraverse[i][attrib_no] = max_within_tolerance;
          //if the current attribute value is below the new min, replace by new min
        if (matrixToTraverse[i][attrib_no] < min_within_tolerance)
          matrixToTraverse[i][attrib_no] = min_within_tolerance;
      }
      testArray = null;
    }
  }
  changesPerformed[1] = 'o';
 }

 else {
   //there was no input given to the method; report an error. Should throw customized exception?
   //System.out.println("No argument given in removeOutliers " + this.getClass());
   throw new IllegalArgumentException();
 }
}

/**
  * writes data that has at least one of the following changes from the original:
  * <ul type="circle">
  *     <li>normalized data, i.e. with a value span of [0,1]</li>
  *    <li>extreme noise reduction a.k.a "outliers"</li>
  *   <li>both</li>
  *</ul>
  */
  public void saveCompleteMatrix(){
   String filename = getFilename();
   FileOps writer = new FileOps(filename);
   writer.writeDataset(heading, inputmatrix, targetmatrix);
   System.out.println("DataTool modified file saved as " + filename);
 }
 /**
  * Does exactly the same as it's polymorph peer but saves the data to
  * a target file with a different<br> filename base as well as attributes.
  * @param filename String filename different than the data source file which<br>
  * constitutes the basis for the complete target file name including change attributes
  */
 public void saveCompleteMatrix(String filename){
   this.infile = filename;
   saveCompleteMatrix();
 }
 /**
  * Builds up a filename
  * @return String
  */
```

```
    String[] filevar = filename(infile);
    String filename = filevar[0];
    String extention = filevar[1];
    //then add the relevant attributes (if any) to the filename:
    if (changesPerformed[0] != 'z')
      filename += "-Normalized";
    if (changesPerformed[1] != 'z')
      filename += "-OutliersRemoved";
    filename += "." + extention;
    return filename;
  }

  /**
   * method that takes a complete filename (e.g. test.file") as an argument and
   * returns a string array of
   * <ol>
   *   <li>the filename (here: "test")</li>
   *   <li>and the extention (here: "file")</li>
   * </ol>
   * @param name String the filename to split
   * @return String[] the array containing file pre and post fix
   */
  public String[] filename(String name){
    int pos = name.indexOf('.');
    int length = name.length();
    String extention = name.substring(pos+1, length);
    String file = name.substring(0,pos);
    String[] filename = new String[2];
    filename[0] =file;
    filename[1] = extention;
    return filename;
  }
public void cleanup(){
  this.instances = null;
  this.heading = null;
  this.infile = null;
  this.inputmatrix = null;
  this.instances = null;
  this.targetmatrix = null;
  this.changesPerformed = null;
}
  /**
   * Private method to force a value in the value span of V_attrib element_of [0,1]
   * @param minimum double minimum value for the current attribute
   * @param maximum double maximum value for the current attribute
   * @param attribute_no int the attribute currently undertaking normalization
   * @param in_matrix double[][] input or target matrix
   */
```

```
   double minimum = -100000.0;
   //increment to ensure values start from zero:
   double[] sorted_array = sortedArray(attribute_no, in_matrix);
   //the thestArray is now sorted in ascending order
   minimum = sorted_array[0];
   return minimum;
}

private double maxValue(int attribute_no, double[][] in_matrix){
  //new maximum value representative for the attribute are initialized:
   double maximum = 100000.0;
   //increment to ensure values start from zero:
   double[] sorted_array = sortedArray(attribute_no, in_matrix);
   maximum = sorted_array[in_matrix.length-1];
   return maximum;
}

private double[] sortedArray(int attribute_no, double[][] in_matrix){
 double[] tmp_array = new double[in_matrix.length];
 for (int i = 0; i < in_matrix.length; i++) {
   tmp_array[i] = in_matrix[i][attribute_no];
 }
 Arrays.sort(tmp_array);
 return tmp_array;
}


}
```