**NTNU – Trondheim**
Norwegian University of
Science and Technology

# Evolutionary Feature Selection

Sigve Dreyer

Master of Science in Informatics
Submission date: November 2013
Supervisor: Anders Kofod-Petersen, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

# Abstract

This thesis contains research on feature selection, in particular feature selection using evolutionary algorithms. Feature selection is motivated by increasing data-dimensionality and the need to construct simple induction models.

A literature review of evolutionary feature selection is conducted. After that a abstract feature selection algorithm, capable of using many different wrappers, is constructed. The algorithm is configured using a low-dimensional dataset. Finally it is tested on a wide range of datasets, revealing both it's abilities and problems.

The main contribution is the revelation that classifier accuracy is not a sufficient metric for feature selection on high-dimensional data.

# Preface

This thesis is the intellectual product of Sigve Dreyer, at the artificial intelligence group at the department of computer and information science at NTNU.

I would like to thank Anders Kofod-Petersen for his supervision and guidance. I also give credit to Bjoern Magnus Mathisen for providing a high performance testing environment.

Sigve Dreyer
Trondheim, November 30, 2013

# Contents

# Chapter 1

# Introduction

Big data is a term thrown around a lot these days. It has many meanings. But there seems to be a loose agreement about what makes data ”big”. The data is ”big” when its size makes traditional approaches to handling it unable to perform satisfactory. Be it an inability to effectively search through the data or just not knowing what to do with it.

In this thesis I will take a closer look at one of these problems: How to decide what is important data and what is not, from the point of view of machine learning?

In machine learning we do not need terrabytes of data for there to be a problem. The traditional algorithms can handle datasets with many instances, in most cases this is a good thing, but when the dimensionality of these sets increases we will quickly get problems. An obvious way to solve this issue is to find out which dimensions are important and which can be ignored.

As a student of information technology I am motivated by the need to make the traditional techniques of machine-learning function in a world were data is bigger. Reducing dimensionality is both time consuming and prone to human errors. Artificial intelligence should be used to handle this chore.

One approach is to find the subset of features that gives highest accuracy on the training data. This is a complex task which turns out to be NP-hard. So solutions can not be found within a reasonable time frame using traditional algorithms. Evolutionary algorithms have been successful at coming up with good solutions for complex problems, when there is a way to measure quality of solutions. And that is the case here, we want a simple model with high accuracy.

I investigate how the problem high-dimensional data poses for machine learning can be solved using evolutionary algorithms. It starts out with an extensive review of the already existing literature, there are a lot of biology-inspired approaches to the feature selection problem in the literature. Then a new algorithm, based on the simple genetic algorithm is proposed and its parameters are tuned for the task. A test protocol is created

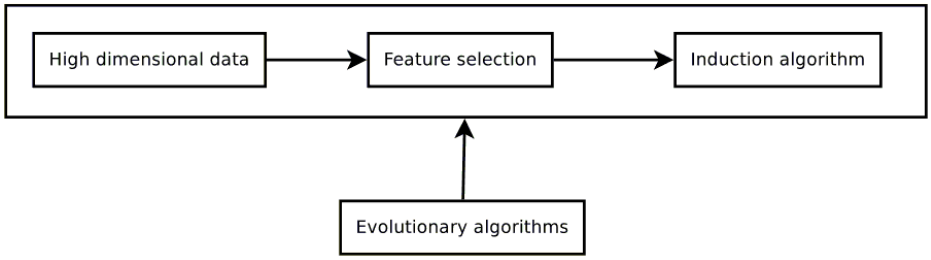


Figure 1.1: The general approach to feature selection and the domain of this research.

containing a range of datasets. It contains very simple problems, very complex problems and multiple interesting performance-metrics. Finally the results of the testing of the algorithm are shown, revealing both the usefulness and the problems of the approach.

# Chapter 2

# Literature review protocol

To gather knowledge about evolutionary algorithms and feature selection a structured literature review [KP] had to be done. This chapter lists the different criteria used when gathering this knowledge.

## 2.1 Approach

Searches are done using Engineering Village [Els]. This is a multi source search platform that includes results from compendex, inspec and others. The articles are first controlled using the inclusion criteria. Those that are deemed relevant are then set up against the quality control statements and included if they have enough quality and relevance.

## 2.2 Research questions

1. Can evolutionary methods be used as a tool for feature selection?
2. What is the goal of feature selection?
3. How are evolutionary methods used as a tool for feature selection?
4. How are evolutionary methods for feature selection evaluated?

## 2.3 Search

The searches are done using two terms.

### 2.3.1 Search terms

1. Evolutionary algorithm, genetic algorithm.
2. Feature selection.

To answer the different research questions a search of this form is used: ST1 AND ST2.

## 2.4 Inclusion criteria

Each inclusion criteria is used to eliminate studies from the search result, starting with the first. This way the other criteria don't need to be checked against studies that would be eliminated by higher ranking criteria.

1. The title of the study suggests both of the search term groups are discussed.
2. The study is from a credible and relevant publication.
3. The main focus of the abstract is on the search terms.
4. The main focus of the conclusion is on the search terms.

## 2.5 Quality control

For a high quality document the following statements should be true:

1. The study considers other relevant research.
2. The study is clear on what is done and what knowledge it unearths.
3. The conclusions reached by the study are extracted from quantitative data, in a logical manner.
4. The study describes how the system was implemented.
5. The study contributes to answering one or more research questions.
6. The approach of the study is reproducible.

# Chapter 3

# Literature review

The literature contains many examples that confirm research question 1: Can evolutionary algorithms be used as a tool for feature selection? The answer is yes.

Actually the initial search resulted in 908 hits. Because of this large number of results the inclusion criteria where applied very strictly. After applying criterion 1 the number was a much more manageable 96 articles. They are listed in appendix A. The high number of irrelevant articles was mainly because the search term ”feature selection” gives many hits in other parts of the biology-inspired artificial-intelligence literature. Also the meta search used gave some double hits.

The rest of this chapter will consider each of the research questions from the protocol, except the first.

## 3.1 What is the goal of feature selection?

Based on the literature review there are multiple reasons for doing feature selection. In this section I will summarise the most common.

In some cases the amount of features can make construction of an induction model hard, either because the model can not fit in memory or construction would take too long. Creating a limited subset can help construct a model within these constraints.

One of the reasons for building classifiers, and in particular decision trees, is to create an understandable decision model. Now an unpruned decision tree made from instances with 100 features is most likely going to be very messy. If the number of features is reduced in a manner that does not significantly reduce accuracy we would get a smaller model. This would make the model more readable and it could also be less over-fitted.

Removing redundant and irrelevant features can help improve the performance of classifiers. This is done by reducing the potential for over-fitting and, in the case of redundant features, selecting those that work best.

### 3.1.1 What is the goal of evolutionary feature selection?

The literature review revealed that evolutionary feature selection has been applied to solving many problems.

#### General classifier improvements

Many of the research paper found in the review take a general approach to feature selection without focusing on a specific domain [Yan+98; Che+12; AlS+10; Zen+09; Wan+12; RC+06; Mun+06; Dro+10; Din+09; Bae+10; MG+07; Pra+10]. There are multiple goals this research tries to reach. The most common task is to improve classifier accuracy and dimensionality reduction on simple machine learning datasets, like those available at the uci machine-learning repository [AA07]. Some of them [Mun+06] also makes the tasks harder by adding irrelevant variables to the data. Others create artificial datasets [Wan+12]. Some of these articles also use their algorithm on a more specific domain.

#### Medical features

The medical field has many areas where machine learning can be useful and feature selection can help find good models for diagnosis and outcome prediction.

[Win+11] uses feature selection to create small models for predicting if tumor-markers are present in blood samples using multiple strategies and wrappers.

Another examples of medical features can be found in [Dur+09] where features from a portable health recording system is used to detect if the user is asleep or awake.

#### Gene expression data

One of the hardest problems in machine learning comes from analysing micro arrays of gene expressions. These are very high dimensional datasets. They contain gene expressions from tissue samples with different properties (classes). This is usually a diagnosis or treatment outcome. The task of analysing them is made harder by the fact that most of these datasets contain very few samples. The goal is to create a high accuracy classifier, by removing the redundant and irrelevant gene expressions, but also to suggest small sets that can be interesting for closer analysis. Thus there are multiple goals. [AlS+10; Wan+12; Mun+06; Ban+07; Can+10; Hua+07; BH+10] all show promising result, but comparison is difficult because many use different datasets. A couple of the studies [Hua+07; BH+10] do compare results with other proposed solutions. Unfortunately, it is unclear what the reported accuracies mean, in [BH+10] they are presented as the best accuracies obtained. Whether these are cross-validation results of the feature selection, results on a hold out set, or just best fitness, is unclear.

### Visual features

The image analysis literature has many examples of evolutionary feature selection. This stems from the high number of features image based machine learning has to choose from not only in the original domain but also different frequency domains. The goals are often to learn different bio-metrics: facial features[Fun+97; Har+05; Vig+12; Liu+08; Nes+09], gesture/activity recognition [Cha+13] and iris recognition [Roy+08]. There are also studies where the goal is generic object recognition or detection of some sort [Che+12; DS+08; Tre+04; Dat+11].

### Text features

Some of the studies focus on extracting features for different induction tasks related to text analysis, this domain can have high dimensional feature-spaces with many useless n-grams.

In [Yah+11] a genetic algorithm is used to select features for dialogue act recognition models.

In [Zhu+10] a parallel and collaborative genetic algorithm is used to learn a pattern for deciding what category a document belongs to.

### Other

There are many other domains where evolutionary algorithms are applied for solving feature selection problems:

- [Kru+12] has success in finding patterns based on surveys, their algorithm both boosts model accuracy and helps clarify the underlying patterns in a manner that gives a better understanding of the data.
- [Viv+03] uses a simple genetic algorithm to create induction models for understanding what makes source-code maintainable by analysing software metrics. The algorithm is improved by the feature selection but little in the form of comparison is presented. The study also analysis the results to suggest how source code can be made more maintainable.
- [Zha+05] uses a genetic algorithm with modified mutation function for selecting data from satellite readings for temperature estimation at different pressure levels.
- In [Sik+05] genetic algorithms are used on a set of industrial plant variables to come up with effective rules for minimizing the use of an expensive chemical.
- [Alf+12] uses feature selection to improve an algorithm that creates summary variables for relational attributes in databases.

- [RC+06] considers feature selection combined with instance selection. The goal here is to avoid instances that make it harder to generalize.

It is clear that evolutionary feature selection can be used in many different domains.

## 3.2 How are evolutionary methods used as a tool for feature selection?

According to [Joh+94] there are primarily two categories of feature selection algorithms. The wrapper and the filter approach. The first subsection explains these two approaches. The rest of the subsections explains the evolutionary approaches in the literature.

### 3.2.1 The wrapper and filter approach

**Wrapper method**

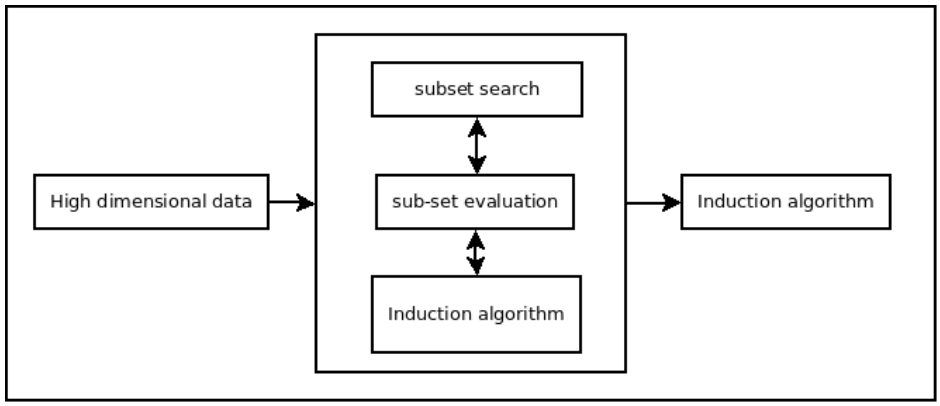


Figure 3.1: The wrapper approach.

Wrapper methods ”wraps around” the induction algorithm that is used for the final classifier. It consists of a heuristic search using the induction algorithm as the heuristic. This search can be something as simple as the backward or forward search presented in [Joh+94].

Naturally the wrapper method can be very expensive. It usually consists of K-fold cross-validation on the training data. But it also has advantages. It evaluates the set of variables selected and not just one at a time, so redundant variables can be removed, and variables that are only useful given other variables can be found. It will also find variables that might be particularly useful with the proposed induction algorithm, so if a

support vector machine (SVM) and a decision tree algorithm get best performance with different variables the wrapper method could take advantage of this.

#### Filter method

The filter method selects features based on statistics only. It can be as simple as selecting the 5 variables with the highest information gain, but the final induction algorithm can not be used. They can be split into two categories; those that evaluates all the features in one pass and those that evaluates multiple proposed feature-subsets combined with a heuristic search.

#### Hybrid method

Any feature selection algorithm that both utilizes wrappers and filters can be considered hybrid feature selection methods. However, if a method only uses a filter as an additional heuristic variable, without effecting the number of wrapper evaluations done, it is not very hybrid. The algorithms usually considered hybrid methods improve run time and/or performance by using filter methods and also use the wrapper method in parts of the algorithm to retain some of the advantages that it provides.

### 3.2.2 Evolutionary approaches

Numerous sources describe different evolutionary approaches to the problem. These are used because the problem is NP-hard and therefore in many cases unsolvable. Random search techniques could give reasonable sub optimal solutions. The most popular approach is the genetic algorithm. In addition ant colony optimisation, particle swarm optimization and genetic programming are also used. The following categorizes these approaches.

#### Ant colony optimization

Some of the literature uses ant colony optimization to perform feature selection [Che+12; Pra+10]. This is done by converting the feature selection problem into a path-finding problem. Then artificial ants are sent to find the optimal path, they do this by laying down pheromones along the edges of the graph. These traces get colder when the paths go unused, thus the ants converge towards the (hopefully) best solution.

#### Genetic algorithms and Evolutionary strategies

Most of these approaches apply a standard genetic algorithm[Yan+98; Wan+12; Sik+05; Har+05; Vig+12; Win+11; Din+09; DS+08; Dur+09; Cha+13; Liu+08; Roy+08; Pra+10;

Kru+12; Viv+03; BH+10]. Some use slightly modified genetic algorithms or evolutionary strategy to optimize the feature-set [AlS+10; Zen+09; Fun+97; Zha+05; RC+06; Win+11; Yah+11; MG+07; Dat+11; Hua+07]. Different genetic encoding may be used, but binary or real numbers are most common. The standard components of a genetic algorithm; genomes, selection, crossover and mutation are combined with a fitness function that somehow estimates the quality of the selected features.

[Tre+04] uses an evolutionary search to do feature extraction, creating new features from the data, and the population is then used as input for a meta classifier.

[Zhu+10] uses multiple evolutions to do feature selection. Many parallel evolutions are done and the algorithm combines individuals from each of them to construct the complete solution.

One of the modified versions uses multi-objective genetic algorithms [Ban+07], it creates sets of solutions that satisfy more than one fitness function, in this case both accuracy and complexity. The most common approach is to combine them and search for an optimal solution.

#### Genetic programming

Genetic programming (GP) is a specialized genetic algorithm. In GP logic structures or programs are constructed by the evolutionary process.

[Nes+09] uses genetic programming to create subsets and then evaluates them with an induction algorithm.

In [Mun+06] each individual has a tree for each class and a weighting scheme is used if they do not agree. Feature selection is done by limiting each individual to a subset of the available features.

[Dro+10] uses a hybrid of GP and genetic algorithms to both shrink the original feature-set and construct new features from combinations of the selected.

#### Particle swarm optimization

Particle swarm optimization is a random search strategy. In this approach many particles are moving through a multi-dimensional space towards the most attractive positions. The movements of these particles is influenced by their previous experiences but also by the other particles, emulating social behaviour. This approach is applied towards feature selection by [Din+09; Bae+10; Pra+10].

### 3.2.3 Evolutionary feature selection using the wrapper method

This seems to be the most popular approach, probably because the classifier accuracy is very tempting as a fitness function. Most of the studies focus on one specific wrapper algorithm and build the evolution around this. The following sections categorize them based on which algorithm is used.

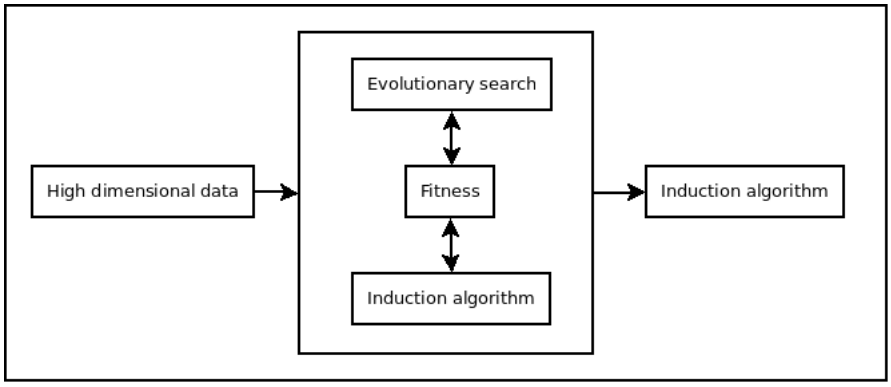


Figure 3.2: Evolutionary wrapper approach.

### Artificial neural-networks

Artificial neural networks (ANN) are popular as the wrapper. This is because these algorithms are capable of creating very accurate models, but also because they are biology-inspired algorithms that researches familiar with evolutionary algorithms also are likely know.

This wrapper is applied in many studies that can be separated into two groups; the ones that use it as a pure wrapper[Yan+98; Zha+05; Dro+10] and therefore only modify input nodes, and those that include parameters or structures of the network into the optimization[Win+11; Dur+09].

### Support vector machines

Support vector machines are applied by many of the studies. Some optimize the resulting parameters in addition to the feature-sets through the evolution [Che+12; Win+11; Din+09; MG+07; Hua+07] and some only use a simple pre-specified SVM [Can+10; Pra+10].

[Roy+08] uses a SVM for its internal fitness function, however it is not used as the final classifier so it is not a text-book wrapper approach. Also the fitness is made up of a weighing including false accept rate and false reject rate, not just recognition rate.

### Bayesian methods

[Kru+12; Nes+09] use the accuracy of the naive bayes algorithm to guide the search. This is a simple algorithm capable of handling continuous and nominal data.

#### Tree-based algorithms

Tree based algorithms are not that common as the wrapper function among the mentioned articles. [Dro+10] performs some tests using the C4.5 algorithm as the wrapper.

#### k Nearest neighbour

The k nearest neighbour (kNN) algorithm is very common in the literature [AlS+10; Fun+97; Wan+12; RC+06; Win+11; Liu+08; Dat+11]. It is a suitable fitness-function because the performance of the algorithm is very dependant on a good feature-subset.

Some of the algorithms use a set number of neighbours to vote among [AlS+10; Fun+97; RC+06; Liu+08]. Others let the number be decided by the algorithm [Win+11]. [Wan+12; RC+06] decide the weight used for each feature with the evolutionary algorithm. [Wan+12] also uses the evolution to decide the distance function (Manhattan-distance, euclidean-distance, etc.).

#### Other induction-algorithms

Some of the algorithms [Sik+05; Mun+06] create rules instead of using an existing induction algorithm classifier.

In addition to SVM, kNN and ANN [Win+11] also uses linear regression in testing.

The K-mean algorithm is used as the wrapper in [Cha+13], the accuracy of this approach is non deterministic, so if a better fitness is found, existing fitness values are updated.

[Tre+04] uses a weak classifier type that will be used in the resulting adaboost classifier as fitness function for individual features.

[Viv+03; BH+10] use linear discriminant analysis as the fitness function.

### 3.2.4 Evolutionary feature selection using the filter method

Filters can either evaluate individual features, giving a ranking, or they can evaluate individual subsets of features. The latter can be used as a statistical test replacing the wrapper method as the fitness function.

[DS+08] uses a separability index for calculating how separate the classes are using the proposed feature-sets, combined with a penalty for large feature-sets. When the feature selection is done, the resulting feature-set is used to train a ANN.

[Ban+07] optimizes both set size and a rough-set theory based measure for discernibility. After selection kNN is used to make classifications.

In [Yah+11] the informativeness of a feature-subset is combined with a punishment for feature-subset size.

### 3.2.5 Other evolutionary solutions for the feature selection problem

There are not that many pure filter approaches to the problem but there are a large selection of studies where hybrid approaches are taken. Both wrappers and filters are utilized to reach good solutions in a reasonable time.

#### Filter prior to evolution

The most popular hybrid solution is to apply filtering prior to using an evolutionary approach to select the final feature-subset. This is done to reduce dimensionality and complexity of the heuristic search, saving time. In [Can+10; Roy+08; BH+10] a feature-pool is created using filters and the evolutionary feature selection is carried out using only this pool.

#### Other hybrid techniques

Most of the algorithms discussed in 3.2.3 have one filter technique included for penalising large feature-subsets. This is done with a separate fitness function in multi objective approaches [Ban+07], by including this penalty directly in the fitness function [Yan+98; Che+12; RC+06; Mun+06; Dro+10; Din+09; Roy+08; Hua+07; Kru+12; BH+10] or by favouring the shortest individual when the fitness is equal[Cha+13; Pra+10].

In [Liu+08] a filter is used to punish individuals with high entropy by giving them a close to zero fitness score.

In [BH+10] the discriminant coefficients of the linear discriminant analysis (LDA) wrapper, is used to also inform the mutation and crossover functions.

## 3.3 How are evolutionary methods for feature selection evaluated?

The literature contains many different evaluation methods, further complicated by many different test-datasets. This section will describe the most popular methods.

### 3.3.1 Classifier accuracy

One indicator of the performance of the feature selection is how well classifiers perform with the selected subset. This is used in almost all of the literature but there are some different approaches. The accuracy is either measured on a single hold out set or cross-validated by running the algorithm multiple times with different training and validation-sets. Sometimes only the accuracy achieved in the fitness evaluation is reported. In many of the papers it is unclear what exactly is measured. This is a problem as one would expect higher accuracy on data used in the feature selection process than on unseen data.

### 3.3.2 Classifier complexity

Another indicator of the efficiency of feature selection is the complexity of the final induction model. Most of the papers report on this by using the number of selected features [Yan+98; Wan+12; Mun+06; Dro+10; Vig+12; Win+11; Din+09; DS+08; Ban+07; Dur+09; Cha+13; Can+10; Tre+04; Alf+12; Hua+07; Pra+10; Kru+12; Nes+09; BH+10]. Some include metrics from the final induction model [Yan+98; Sik+05; Dur+09] like number of nodes. Many of the algorithms return specific sized subsets [AlS+10; Liu+08; Roy+08; Dat+11; Hua+07], these report the accuracy with different pre-specified numbers of features.

### 3.3.3 Comparison to other solutions

Comparison to other solutions is important, unfortunately in many of the studies non or few other similar algorithms are compared. In many cases the comparison is made using a classifiers accuracy with all the features [Yan+98; Wan+12; Zha+05; Sik+05; RC+06; Mun+06; Din+09; DS+08; Dur+09; Liu+08; Pra+10; Nes+09]. Some compare their algorithm with runs of other feature selection algorithms [Yan+98; AlS+10; Zen+09; DS+08; Ban+07; Yah+11; Bae+10; MG+07; MG+07; Tre+04; Roy+08; Dat+11; Hua+07; Pra+10; Kru+12; Zhu+10], while others include reported results of other algorithms [Mun+06; Vig+12; Cha+13; Hua+07; Pra+10; BH+10] or just refer to other results [Win+11; Hua+07]. Usually the comparisons are on number of selected features and accuracy of the resulting models, but some of the studies also include time-to-complete on problems with known optimal solutions [Bae+10; AlS+10].

The comparisons are complicated by the use of different datasets. For example [BH+10] contains a good overview of reported results from micro-array feature selection but each study only contains results from a small subset of the datasets.

## 3.4 Summary

The literature has many proposed solutions to the feature selection problem that use different evolutionary approaches, like genetic algorithms. The problem of feature selection is general, but many of the studies focus on a specific domain. The algorithms either use an induction algorithm's accuracy or a statistical measure to estimate the value of solutions, most of them also penalise complexity. The algorithms are evaluated by looking at the accuracy and size of the final feature-subsets, but there are many different metrics used.

# Chapter 4

# An evolutionary feature selection algorithm

The literature review reveals that there are many different wrapper based algorithms using many different induction algorithms. I want to create an abstract algorithm capable of using any induction algorithm as the wrapper.

This chapter will describe the construction of a feature selection algorithm using evolution as the heuristic search. The first section describes genomes, phenos, fitness and the other evolutionary functions. The the second section describes details of the proposed algorithm. The last section summarizes the implementation.

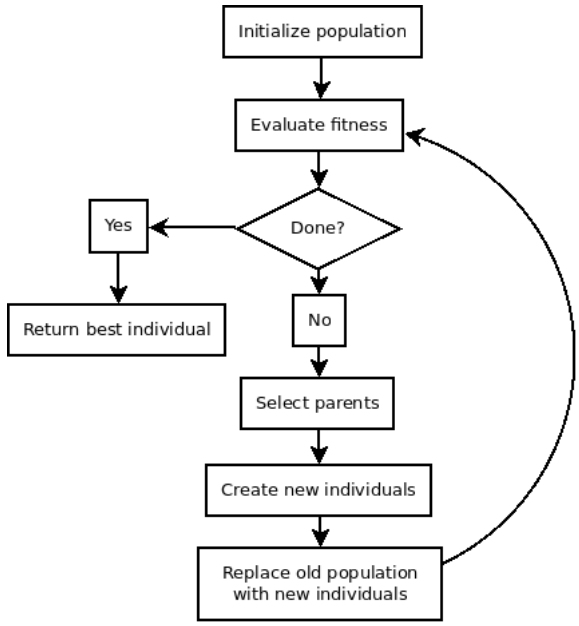


Figure 4.1: A general evolutionary algorithm

## 4.1 Genome and phenotype

To create a genome representing a feature selection we first need an understanding of what we expect as the result of a feature selection algorithm. [Kal+07] summarizes three types of feature selection outputs:

**Weight-scoring**
: Each feature is given an importance score.

**Ranking**
: The features are sorted based on their importance.

**Feature-subset**
: A set of attributes the algorithm proclaims to be best suited to create the final model.

Some induction algorithms can use the weights or the ranking as input, but most of the algorithms expect a finite set of features. This can be created from each of the outputs. When using ranking a fixed set size can be used, so only the N best features are used. Weight-scoring can use the same approach or set a threshold for inclusion.

The subset output can not distinguish between the selected features but it can be used to create the model without any finite threshold used in the conversion.

This algorithm uses the feature-subset as its genome. This way any filtering on how important a feature has to be can be done by the evolution instead of by post-processing the output. This is an advantage as the solutions are general.

The genome will be based on the feature-subset and the pheno is an evaluated feature-subset, including some heuristic on how well it works. This is implemented by having each individual contain a set of selected features. Most of the genetic algorithms discussed in the literature do this with a simple bit-string, each bit signalises if the feature is included or not.

### 4.1.1 Fitness

The fitness is the heuristic used to compare different feature-subsets. It should indicate how close we are to an optimal solution.

#### Filters

As discussed in the literature review, there are two main types of filtering. One that evaluates different features and one that evaluates individual feature-sets. The latter can be used as a heuristic for how well the feature-subsets perform. These filters evaluate the subsets independently of the induction algorithm used, and can therefore not be used to find or exclude features that work particularly well for this. The primary benefit of using filters is low cost when compared to the wrapper function. The proposed algorithm does not use filters as fitness, but there are not many modifications needed to use filters instead of wrappers.

#### Wrappers

The wrapper function is an obvious candidate for measuring fitness. This could be the accuracy from any classification algorithm. The wrapper approach also has the advantage that specific considerations can be made for each algorithm.

One of the good existing libraries with many induction algorithms is weka [Hal+09]. By abstracting the fitness function for wekas classifier super class the algorithm can utilize any of them to measure fitness.

#### Hybrid

The literature has many examples of feature selection algorithms that combine filter functions with wrapper functions. The proposed algorithm will just do a very simple, but popular, combination. By combining the wrapper fitness with a penalty for feature-set size, we can hopefully get both accurate and simple solutions.

The proposed length penalty is just a general cost penalty that assumes every feature is obtained at the the same expense. [Yan+98] discusses how a similar function, that takes each features actual cost into account, can be constructed. Unfortunately most of the time this information is unavailable and in some cases it might not be relevant, for example if the features are created using feature-extraction. The length penalty is a general punishment for complexity. Without it all features that don't worsen the accuracy of the induction algorithm can be considered worthy of inclusion. But if it is to high, valuable features can be filtered out resulting in poorer models.

The literature contains different approaches to penalising long feature-sets. The most common approach is to add a penalty that depends on the number of features used and then dividing it by the number of available features, punishing complexity more when the dataset is low dimensional. I take the other approach and punish with a flat rate independent of the dataset, because the amount of available features should not be used as an indicator on how complex the underlying model is.

### 4.1.2 Selection

In the simple genetic algorithm selection is most commonly done using a roulette-wheel approach, where each individual is given a segment of the wheel based on its fitness. With feature selection, this does not seem to be an appropriate approach because the fitness difference between different subsets can be very slim resulting in very low selection-pressure.

To avoid this issue we use tournament selection instead. This has the advantage that the value of the fitness function is irrelevant. With tournament selection the only thing that matters is the ranking of the different individuals. A small group from the population is randomly selected from the population and the one with highest fitness is selected. The size of this group can be regulated, bigger groups equals more selection pressure. The tournaments can also have random winners, the chance of random selection can be adjusted to change selection pressure.

### 4.1.3 Mutation

This algorithm distinguishes itself from others by using sets as genotypes. The mutation functions have to take this into consideration.

This is done by imitating both the forward and backward search algorithms. Each mutation adds or removes one feature from the genome.

This might seem like an over-simplified approach, but the hope is that the other steps in the evolution will make this sufficient. For example getting out of local minimums can be done by randomized selection instead of changing large parts of the genome.

### 4.1.4 Crossover

Using sets instead of regular strings also makes regular crossover a hard fit. Instead of including both inclusion and exclusion variables in the crossover mix, the proposed approach only focuses on what features to include.

A fixed set of parents are selected using tournament selection and then all the features are laid out on a roulette wheel, giving features that appear in multiple parents more space. Also there is no mirror child that we see in the traditional crossover technique.

### 4.1.5 Elitism

Elitism can also be used to make it easier for the algorithm to converge towards good solutions.

The idea is simple, keep the N best solutions after each iteration of the algorithm. This way the best solution will be preserved until a better solution is found.

## 4.2 The algorithm

The proposed algorithm relies heavily on the weka[Hal+09] classifier library, it contains all the induction algorithms and some of the data handling code. This sections main concern is the algorithm details. An important part of this algorithm is what parameters are used, but this will be discussed in chapter 5.

### 4.2.1 Algorithm description

The algorithm starts with a fresh population of feature subsets. It then goes into a loop consisting of 5 separate steps.

First it evaluates the fitness of each subset(FS) using the wrapper classifier. This is done using cross-validation on the training data, with a pre-set number of folds (NF). The fitness is set to the classification accuracy minus a length penalty, a constant (LP) multiplied by the size of the subset. This will punish complexity in the way discussed in 3.2.5.

$$fitness = crossvalidate(NF) - (LP \times size(FS)) \tag{4.1}$$

Secondly, sets of parents are selected using tournament selection, the size of the sets can be changed to adjust crossover.

The third step is child creation. For each parent-set a new feature-subset is created. The size of the new set is the average size of it's parents and the features are randomly selected from the parents. If a feature is used by multiple parents it's inclusion chance increases.

The fourth step is mutation. This consists of randomly adding and removing random features from the subsets. The mutation rate (MR) determines how likely this is. Each newly created subset can add and/or remove one random feature.

The last thing done is removing the old generation from the population. The algorithm can be configured to use elitism by not removing a few (E) of the best existing subsets.

The loop will run for a fixed set of iterations or until a stopping criterion is met, for example 100% accuracy on a subset with 10 or fewer features.

### 4.2.2 Pseudocode

```
Pseudocode:

while(!done()){
    evaluate(population);
    parentSets=tournamentselection(population);
    for(ps : parentSets)
        population.add(new FeatureSet(ps));
    population.mutateNew();
    population.removeOld();
}
```

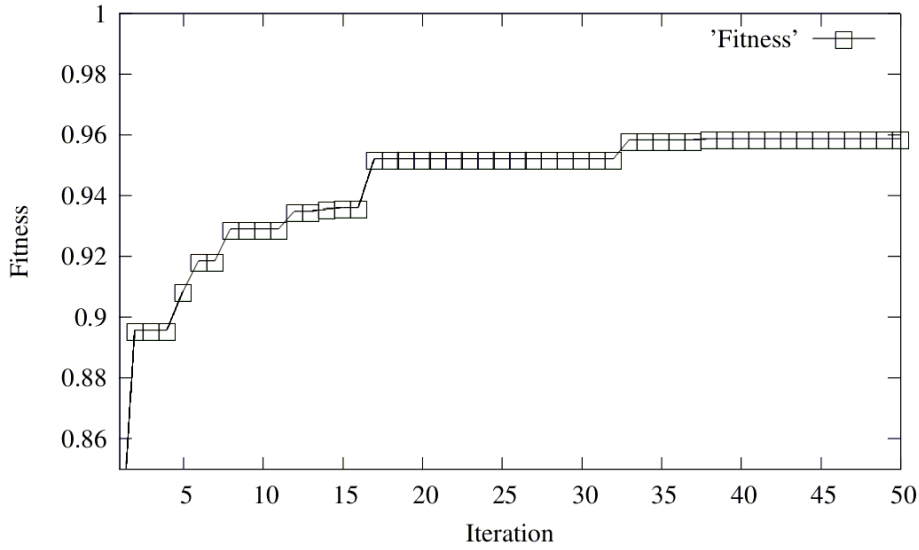


Figure 4.2: An example run of the algorithm

## 4.3 Implementation details

The pseudocode in 4.2.2 describes what the algorithm does but it is far from the actual implementation. Most of the code is already created in the form of classifier algorithms in weka. Hooking this up with a generic evolutionary algorithm required a couple of new classes. The implementation can be seen in figure 4.3. It is done using the Java programming language.

The classes are summarized in figure 4.3. Geno, Pheno, Evolution and the population functions are part of a generic evolutionary library implementing some analysis tools and selection algorithms.

The Classifier is an abstract class from the weka library, all induction algorithms extends this class. Its most important methods are buildClassifier() and classifyInstance(). EVFS is the class used for testing the proposed algorithm and it implements both these methods. EVFS contains a classifier and also inherits the Classifier class, the classifyInstance method uses the classifier created with the algorithm to classify new instances after removing unused indexes. The buildClassifier method runs the algorithm.

The classes important to implementatin of the feature selection-algorithm is from left to right: FeatureSet, ev, FitnessFunction and Datawrapper. The FeatureSet class is the pheno representation of feature-subsets, it also handles the genome functionality by implementing combineWith() for crossover and mutate() for mutation. The ev is just

an Evolution object with the appropriate population functions. The WrapperFitness is the population function responsible for ranking the FeatureSets based on the training data. When the Fitness function is run on the population its setFitness function is called for each FeatureSet in the population. This function evaluates a single featureSet using cross-validation. The lp (length penalty) is used to determine how much the subsets should be penalized for each feature. nf (number of folds) determine how many folds should be used for the cross-validation used to determines their base accuracy. The DataSource is needed by the fitness function as it contains all the training data. It also contains the function getFeatureSubset that can be used to create a new DataSource where only the given columns are included so that specific subsets can be evaluated.

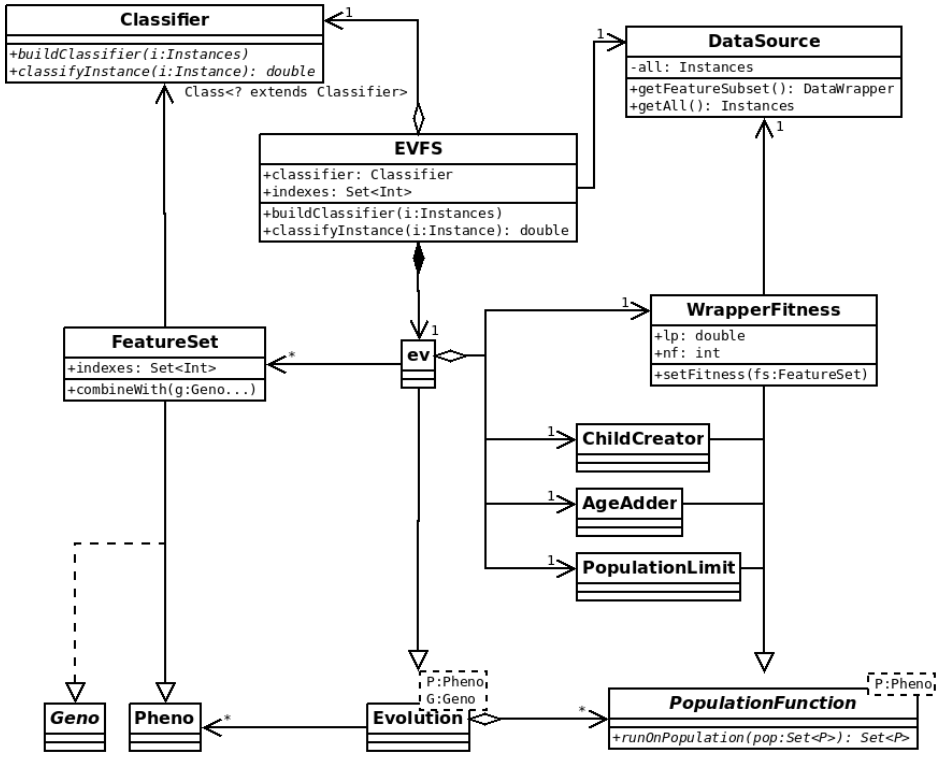


Figure 4.3: Class diagram of the implementation

# Chapter 5

# Parameters

This section describes how the default values of the different parameters where determined. This is done by measuring performance on a well defined feature selection task, the benchmark test.

When defining multiple variables for an algorithm, picking the order they are analysed in is tricky. This starts out with parameters that seem normal in the literature or are intuitive. We start with the analysis of the elitism settings. The settings are summarized at the start of each section.

## 5.1 Benchmark

To find the correct parameters for the evolution a benchmark was developed. It uses the wine dataset, see 8.2. This dataset has 13 features, from these we can create $2^{13}$ or 8192 different subsets of features. This means that brute force can be used to find the highest ranking feature-subset. The goal is then to minimize the time it takes to find this solution. The time is measured in number of classifier evaluations performed. In case there are multiple solutions with the same score any of them will be acceptable. The benchmark is run 300 times with different seeds so we can get an average. To give a more full view of the performance the output also includes best case, worst case and the standard deviation.

The classifier used is the IB5 classifier. It was selected because it is fast and it only has one optimal solution for the test data when using the 10-fold cross-validation on the wine data with 0 as the data randomization seed. It is important that the brute force evaluation is done with the same evaluation method as the fitness function. This benchmark task could be unsolvable if the fitness function was modified (for example by using length penalty) or the training data was slightly different.

The benchmark is very strict in that it requires an optimal solution, something that

is difficult to guarantee when dealing with larger datasets. In these cases the task would be to find a close to optimal solution. For example for the Sonar dataset making sure the feature-set is optimal would involve $2^{60}$ evaluations which of course is unreasonable. This benchmark is a check to see that the evolution goes smooth, avoids local minimums and efficiently makes it's way through the search space.

### 5.1.1 System specifications

To conduct these experiments a high-performance computer was used. This was kindly provided by Sintef. Here are some of the technical specifications for the tests:

**CPUs:** 2x Xeon x5650 @ 2.67 GHz, 2 x 6 Cores with hyper-threading

**Memory:** 44 GB of ram memory

**Java-version:** Java-7-openjdk-amd64

**Operating-system:** Ubuntu, Linux-kernel version 3.11.0-12-generic

## 5.2 Elitism

Benchmark options:

**Initial population:** no features

**population size:** 50

**mutation chance:** 1.0

**crossover rate:** 0.0

**group size:** 2

**random chance:** 0.0

**elitism:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

The average benchmarks for 3 or more elite individuals are not entirely correct, when the benchmark reaches 25000 the algorithm stops even if the ultimate solution is not reached. The results are probably not that far off, the number of runs that didn't complete is in the range of 1 to 7 out of 300.

From table 5.1 we can see one thing clearly: using elitism improves performance on average. The highest mean run time is clearly the run that does not use elitism.

Table 5.1: Benchmark results for different elitism settings

| elitism | mean | std | max | min | unfinished |
|---|---|---|---|---|---|
| 0 | 4124 | 2912 | 15513 | 363 | 0 |
| 1 | 2852 | 2330 | 17063 | 413 | 0 |
| 2 | 2872 | 2492 | 17413 | 463 | 0 |
| 3 | 3111 | 3758 | 25013 | 563 | 2 |
| 4 | 2920 | 3077 | 25013 | 563 | 1 |
| 5 | 2445 | 2737 | 25013 | 563 | 2 |
| 6 | 2936 | 4231 | 25013 | 463 | 5 |
| 7 | 2820 | 3283 | 25013 | 463 | 2 |
| 8 | 2623 | 3551 | 25013 | 563 | 2 |
| 9 | 3204 | 4735 | 25013 | 463 | 7 |

There is also a trend that elitism can speed up the process. Too much and the number of evaluations get higher and some of the tests do not finish. The deviation also gets bigger the more elitism is used.

It is also quite clear that all the benchmarks have big deviations in the time it takes to finish. On average the cases where elitism is used are all better than the brute-force approach, it would use on average $\frac{8129}{2} = 4096$ evaluations to find the best solution. But the algorithm can also use much more time, the worst case with one elite individual took 17063 evaluations to reach the goal, more than the worst case of brute force.

The benchmarks suggest that the best performance is achieved when there is some elitism. Not too much, however, because the more elitism there is the more deviation there is in performance. The best mean, with no unfinished runs and lowest deviation, is 1 elite individual, so we use this from now on, keeping in mind that a little higher elitism could bring the average run-time down.

## 5.3 Population size

Benchmark options:

**Initial population:** no features

**population size:** 10, 20, 30, 40, 50, 60, 70, 80, 100

**mutation chance:** 1.0

**crossover rate:** 0.0

**group size:** 2

**random chance:** 0.0

**elitism:** 1

Table 5.2: Benchmark results for different population sizes

| population-size | mean | std | max | min | unfinished |
|---|---|---|---|---|---|
| 10 | 3062 | 4598 | 30013 | 163 | 3 |
| 20 | 2548 | 3192 | 30013 | 193 | 1 |
| 30 | 2584 | 2218 | 13843 | 253 | 0 |
| 40 | 2499 | 1932 | 12333 | 373 | 0 |
| 50 | 2702 | 1878 | 14013 | 463 | 0 |
| 60 | 3138 | 2342 | 17893 | 733 | 0 |
| 70 | 3404 | 2740 | 29903 | 503 | 0 |
| 80 | 3402 | 2296 | 18733 | 653 | 0 |
| 90 | 3754 | 2548 | 26383 | 733 | 0 |
| 100 | 3919 | 2086 | 13913 | 1013 | 0 |

The results show the initial guess for population size wasn't that far off. However, a little smaller population size saves some evaluations. Also note that if it is too small, there is a drastic performance loss. The population size is set to 40, keeping the performance loss with too small populations in mind for more complex problems.

## 5.4 Initial population

Benchmark options:

**Initial population:** no features, one random feature each, random subsets

**population size:** 40

**mutation chance:** 1.0

**crossover rate:** 0.0

**group size:** 2

**random chance:** 0.0

**elitism:** 1

(a) no features (b) one each (c) random subsets (d) one for each features

Figure 5.1: Different initial populations. Each line is an instance and each colomn is one feature

The benchmark shows that there isn't that big a difference between the initial populations. It is intuitive that starting with empty subsets would take more time than starting with one attribute in each subset, something these numbers contradict. One explanation could be that early crossover between single good attributes and empty subsets helps to focus the search.

The initial population that gives the best performance is random subsets. This makes sense because it gives the algorithm the opportunity to start the search very close to the goal, something we can see in the minimum run time. However, it does not help avoid the worst case.

Table 5.3: Benchmark results for different start populations

| initial population | mean | std | max | min |
|---|---|---|---|---|
| no features | 2634 | 2363 | 17521 | 401 |
| one random feature each | 2955 | 2606 | 19761 | 401 |
| random subsets | 2477 | 2584 | 19761 | 41 |
| one pheno for each | 2783 | 2256 | 17521 | 401 |

## 5.5 Crossover rate

Benchmark options:

**Initial population:** random subsets

**population size:** 40

**mutation chance:** 1.0

**crossover rate:** 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0

**group size:** 2

**random chance:** 0.0

**elitism:** 1

Table 5.4: Benchmark results for different crossover rates

| crossover-rate | mean | std | max | min | unfinished |
|---|---|---|---|---|---|
| 0 | 2549 | 2531 | 15281 | 41 | 0 |
| 0.1 | 2783 | 3300 | 19921 | 41 | 0 |
| 0.2 | 2304 | 2817 | 20001 | 41 | 2 |
| 0.3 | 2426 | 2671 | 20001 | 121 | 1 |
| 0.4 | 2442 | 2609 | 14881 | 41 | 0 |
| 0.5 | 2641 | 3124 | 20001 | 41 | 1 |
| 0.6 | 3270 | 3571 | 20001 | 41 | 4 |
| 0.7 | 3290 | 3733 | 20001 | 41 | 2 |
| 0.8 | 4199 | 4265 | 20001 | 41 | 3 |
| 0.9 | 4867 | 4831 | 20001 | 41 | 6 |
| 1 | 5092 | 5303 | 20001 | 81 | 13 |

The numbers in table 5.4 suggest that some crossover has a positive effect. The increase is not drastic. There is, however, a negative effect from applying crossover too often. In the range of 0.0 to 0.5 there is not much of a difference, but 0.4 is chosen because it has the lowest run time in combination with no unfinished runs.

## 5.6 Mutation rate

Benchmark options:

**Initial population:** random subsets

**population size:** 40

**mutation chance:** 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0

**crossover rate:** 0.4

**group size:** 2

**random chance:** 0.0

**elitism:** 1

Table 5.5: Benchmark results for different mutation-rates

| mutation-rate | mean | std | max | min | unfinished |
|---|---|---|---|---|---|
| 0 | 8191 | 3745 | 10001 | 81 | 243 |
| 0.1 | 7051 | 4295 | 10001 | 41 | 200 |
| 0.2 | 5712 | 4346 | 10001 | 41 | 138 |
| 0.3 | 4239 | 3918 | 10001 | 81 | 73 |
| 0.4 | 2836 | 2831 | 10001 | 121 | 21 |
| 0.5 | 2417 | 2429 | 10001 | 41 | 8 |
| 0.6 | 2069 | 2219 | 10001 | 81 | 10 |
| 0.7 | 2204 | 2117 | 10001 | 41 | 2 |
| 0.8 | 2285 | 2225 | 10001 | 81 | 6 |
| 0.9 | 2486 | 2462 | 10001 | 121 | 8 |
| 1 | 2577 | 2501 | 10001 | 41 | 12 |

The mutation rate used in the previous tests might seem very high, but it makes sense when you consider what the mutation function actually does. With a mutation rate below 1.0 there would be individuals with no change from their parents and the

algorithm would just be wasting time evaluating some of the same subsets. Now with the introduction of crossover, another way of creating new subsets, this parameter should be re-evaluated.

The benchmark results in table 5.5 show that a high mutation rate still is the best approach. However, applying a mutation to every new subset does not give the best performance. It is worth noting that the maximum evaluations allowed was decreased for these runs. A mutation rate of 0.7 gives the second best evaluation time, but also fewest unfinished runs, and is therefore picked.

## 5.7 Group-size

Benchmark options:

**Initial population:** random subsets

**population size:** 40

**mutation chance:** 0.7

**crossover rate:** 0.4

**group size:** 1, 2, 3, 4, 5, 6, 7, 8, 9

**random chance:** 0.0

**elitism:** 1

Table 5.6: Benchmark results for different group-sizes

| group-size | mean | std | max | min | unfinished |
|---|---|---|---|---|---|
| 1 | 8464 | 3136 | 10001 | 41 | 228 |
| 2 | 2544 | 2358 | 10001 | 41 | 5 |
| 3 | 2019 | 2128 | 10001 | 41 | 3 |
| 4 | 4264 | 4271 | 10001 | 41 | 88 |
| 5 | 4846 | 4411 | 10001 | 41 | 113 |
| 6 | 5871 | 4476 | 10001 | 81 | 149 |
| 7 | 6564 | 4363 | 10001 | 41 | 176 |
| 8 | 6875 | 4330 | 10001 | 41 | 189 |
| 9 | 6952 | 4349 | 10001 | 41 | 191 |

These numbers show that once again the best performance is achieved with well balanced parameters. It is however surprising that slightly bigger group-size has such an

impact on performance. Going from group size 3 to 4 doubles the number of evaluations. This could be because of the low population size.

The difference between a group size of 2 and 3 is not very significant. I am reluctant to pick 3 because of the drastic performance drop when using 4 and keep both as potential candidates for now.

## 5.8 Random selection

Benchmark options:

**Initial population:** random subsets

**population size:** 40

**mutation chance:** 0.7

**crossover rate:** 0.4

**group size:** 2, 3

**random chance:** 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7

**elitism:** 1

Table 5.7 shows that selection pressure is important. The lower the group size the lower the pressure, the lower the chance of random selection the higher the pressure. With a group size of 2 the pressure is low enough without any random winners. With 3 in each group a few random winners give better performance. The best performance comes with 0.2 random chance and a group-size of 3.

Table 5.7: Benchmark results for different chances of random tournament winners

| group-size | random chance | avg | std | max | min | unfinished |
|---|---|---|---|---|---|---|
| 1 | 0 | 2264 | 2585 | 19761 | 41 | 0 |
| 1 | 0.1 | 2644 | 2841 | 20001 | 161 | 1 |
| 1 | 0.2 | 2987 | 3190 | 20001 | 41 | 1 |
| 1 | 0.3 | 3720 | 4120 | 20001 | 81 | 5 |
| 1 | 0.4 | 4631 | 4660 | 20001 | 41 | 4 |
| 1 | 0.5 | 6450 | 5917 | 20001 | 41 | 15 |
| 1 | 0.6 | 7606 | 6527 | 20001 | 41 | 32 |
| 1 | 0.7 | 10019 | 7583 | 20001 | 121 | 77 |
| 2 | 0 | 2332 | 3207 | 20001 | 41 | 1 |
| 2 | 0.1 | 1923 | 2211 | 14801 | 81 | 0 |
| 2 | 0.2 | 1721 | 1729 | 12361 | 41 | 0 |
| 2 | 0.3 | 1873 | 2148 | 15321 | 81 | 0 |
| 2 | 0.4 | 1854 | 1965 | 12801 | 41 | 0 |
| 2 | 0.5 | 2304 | 2618 | 20001 | 81 | 1 |
| 2 | 0.6 | 2805 | 3213 | 20001 | 41 | 3 |
| 2 | 0.7 | 5531 | 5472 | 20001 | 81 | 14 |

## 5.9 Final settings

options:

**Initial population:** random subsets

**population size:** 40

**mutation chance:** 0.7

**crossover rate:** 0.4

**group size:** 3

**random chance:** 0.2

**elitism:** 1

If these settings will work for anything other than the benchmark used here remains to be seen, but by looking at all the numbers it is clear that picking the correct settings can have a huge effect on the algorithm's ability to reach the global optimum. It is no surprise that settings that remove any selection pressure will degrade performance. But completely reasonable variables like slightly larger group-size (and slightly higher selection pressure) can also drastically increase the run time.

# Chapter 6

# Dealing with noise

The benchmarks in chapter 5 are created using a dataset with very little noise. Feature selection will be more useful the more noise there is. In this section we will do benchmarks with different levels of noise added to the original dataset to see if the algorithm and it's parameters are still suitable.

When adding noise to the benchmark dataset the old optimal solution is still valid, however new solutions may also become valid, see table 6.2. In the original benchmark the accuracy is very high (0.9944) and there is only one subset that is able to give this rate: $\{0, 1, 4, 6, 7, 9, 10, 12\}$. This is no longer true when we start to add noise.

For the following benchmarks we consider the task completed when an accuracy of $<= 0.9944$ is reached, even if there does exist subsets that show no errors in the cross-validation, since they clearly reach this level of accuracy by over-fitting and including attributes that are unrelated to the concept. What we hope to get from the algorithm is maximum accuracy and minimum complexity. Therefore we need some more metrics for the tests:

**# feats:** The number of features in the solutions.

**avg accurasy:** The average accuracy of the solutions.

The results from the initial noise benchmark can be seen in table 6.1. The more noise that is added the more iterations are required to reach a high enough fitness. 10 noise variables dilutes the training data and makes the search space $2^{10} = 1024$ times bigger so an increase is expected, and finding the best solution is no longer the highest priority. The main problems are too diverse and big solutions, especially with 30 noise variables. Looking at table 6.2 we see very different solutions, but all the solutions have the same fitness ($> 0.994$ accuracy) in the cross-validation. One of the reasons we do feature selection is to create understandable models so clearly fewer variables in the solution is better. Also reaching a maximum with many variables is much harder because there are

Table 6.1: Benchmark results for different levels of noise.

| added noise | avg | std | max | min | fails | # feats | avg accuracy |
|---|---|---|---|---|---|---|---|
| 0 | 1696 | 1882 | 14961 | 41 | 0 | 9.00 | 0.9944 |
| 10 | 2943 | 4019 | 20001 | 161 | 9 | 10.60 | 0.9946 |
| 20 | 3273 | 3472 | 20001 | 321 | 4 | 15.88 | 0.9946 |
| 30 | 9593 | 7283 | 20001 | 601 | 78 | 18.64 | 0.9929 |

Table 6.2: Output examples for different noise levels

| # of noise variables | selected variables |
|---|---|
| 0 | $\{0,1,4,6,7,9,10,12\}$ |
| 10 | $\{0,1,4,5,6,8,9,10,11,12,15\}$ |
| 20 | $\{0,2,3,4,5,6,9,10,11,12,+7\text{ more}\}$ |

so many ways to mutate big subsets. The size of the subsets in the middle of execution could make the search much harder. The resulting accuracy and complexity is not that much worse with 30 noise variables but we might be able to do better by modifying parts of the algorithm.

## 6.1 Length-penalty

If the fitness-function is modified performance could improve. We start by doing the same benchmark with a close to zero value (0.00000000001) for the length penalty. This will give the shortest of two equally accurate subsets an advantage in the tournament selection.

Table 6.3: Benchmark results when using close to zero length penalty.

| noise | avg runs | stdev | max | min | # fails | # features | avg accuracy |
|---|---|---|---|---|---|---|---|
| 0 | 1311 | 1068 | 6841 | 81 | 0 | 9.00 | 0.9944 |
| 10 | 2648 | 2789 | 19721 | 321 | 0 | 9.87 | 0.9947 |
| 20 | 2609 | 2476 | 20001 | 441 | 1 | 13.78 | 0.9946 |
| 30 | 6706 | 5708 | 20001 | 641 | 23 | 14.35 | 0.9942 |

The algorithm's run time is slightly better for all noise-levels, this could be because the penalty makes it easier to travel through the search space. By giving simpler solutions a slight advantage, fewer runs are required to reach the stopping criteria. Also the final

solutions are less complex. There is very little difference on the low noise tests, but with 30 noise variables the solutions are on average 4 variables smaller when using the penalty. Because these results are only positive without restricting us to new optimal solutions, the close to zero length-penalty is used for all the following benchmark tests.

The next test was done to illustrate one of the problems with the length penalty. What happens when the best feature-set no longer gets the highest fitness because it is too long? When the penalty is put at 0.003 the best solution, according to the fitness-function, is a subset with only 0.98 accuracy. Note that the stopping criteria is lowered to this value for these tests.

Table 6.4: Benchmark results when using 0.003 as the length penalty.

| noise | avg runs | stdev | max | min | # fails | # features | avg accuracy |
|---|---|---|---|---|---|---|---|
| 0 | 496 | 439 | 4321 | 41 | 0 | 8.13 | 0.9899 |
| 10 | 1266 | 1061 | 7361 | 241 | 0 | 8.34 | 0.9898 |
| 20 | 1621 | 1758 | 14601 | 161 | 0 | 10.29 | 0.9897 |
| 30 | 2097 | 1734 | 12801 | 361 | 0 | 9.96 | 0.9897 |

From table 6.4 we see the number of runs is drastically reduced. The accuracy is unfortunately also slightly reduced. The number of features is also lower at all the noise levels, which means less complex solutions.

When using the close to zero penalty one of the most accurate solutions will always be picked, if the algorithm can reach it. The close to zero length-penalty also reduces the runtime, even when there is no artificial noise added. The results of the test with 0.003 length penalty shows that sacrificing the ability to reach the best solution can help reduce the time needed to reach a close to optimal solution.

We can safely say that a non-zero length penalty should always be used. How high it is set, however, should be determined by how big the feature-space is, how much training data is available and how complex solutions we want.

## 6.2 Initial population

When the dimensionality increases, there is a risk of running the algorithm for many generation without the most significant features showing up in the gene pool. This can be avoided by tweaking the initial population. The proposed solution is making an initial population with one individual for each variable with only this variable in it's genome. This should not affect the performance by starting further from the global optimum than with random genes, because it is highly unlikely to start close to it. The wrapper method is used for the fitness of each of them. The features that are most useful on their own are

then preserved through normal elitism and the selection process is more likely to pick them for future breeding.

This is also done because the use of completely random subsets, the best performing on the small scale, will start with very big subsets. This will increase the run-time of the fitness evaluations.

# Chapter 7

# Run-time optimization

Initial testing suggests the majority ($> 99\%$) of the run-time is spent evaluating subsets. In this chapter we will discuss two techniques to improve the run-time. They will focus on limiting the time of the fitness evaluation.

## 7.1 Look-up table

It is obvious that the algorithm will have much redundancy. At the end of a run the population will become less diverse and new subsets are more likely to have already been evaluated.

A look-up table linking all evaluated subsets to their fitness could decrease run-time, especially with lower dimensionality datasets and a resource hungry induction algorithm. This is a classic case of memory versus run-time trade-off, storing the fitness for each possible subset becomes impossible if there are many features. On the other hand, if the induction algorithm is slow, we will run out of patience long before memory is full.

Evaluation of this optimization is done on the system described in 5.1.1. We use the same parameters as for previous benchmarks, but also include run-time.

Table 7.1: Benchmark with and without lookup table.

| look-up table | average time (ms) | average # evaluations | worst # evaluations |
|---|---|---|---|
| no | 4139 | 1230 | 5401 |
| yes | 1759 | 521 | 1656 |

Table 7.1 shows that about half of the evaluations can be avoided by using the look-

up table. However, it seems to be even more helpful for the worst cases where the algorithm gets stuck in local minimums for some time. For these runs the number of evaluations is reduced by more than $2/3$.

By using this technique, the solution to the benchmark, that took on average 4096 evaluations to brute force, can be found by doing on average 521 evaluations.

## 7.2 Multithreading

There are many examples of parallelizable induction algorithms. Given the abstraction of the proposed algorithm, the only practical parallelization that can be done for the fitness evaluation, is to split the fitness evaluations up among the available cpu cores.

The implementation is parallelized by creating a list of tasks: one for each individual in the population. A thread-pool is made, with one thread for each available cpu-core. Each thread takes one task and executes it. If there are more available afterwards, it goes on to another task.

For an infinite population we would get linear speed-up, the best possible speed-up. Since the current algorithm merges once for each generation, the ability to speed-up will be limited by the population size. To test how this optimization performs, the benchmark was run on 5.1.1 with different thread-pool sizes from 1 to 12. The results of this, and the curve of linear speed-up, can be seen in figure 7.1.

The algorithm scales pretty well. Multi-threading can significantly reduce the run time by using more cores. In this case going from 1.75s in average run-time when single threaded to 0.37s when 12 threads are in use. But when we compare the run time with linear scaling (1/x), the results are far from perfect.

The main problem is that the algorithm is synchronous and therefore has to end all the threads each generation. This means waiting until the last and most likely slowest individual is evaluated while the other threads idle. This problem can be removed by making the evolution asynchronous. But this would also cause the algorithm to change, for example small subsets could reproduce more quickly because they take less time to evaluate. It's not necessarily a problem, but it would require a rework of the entire algorithm.

One of the ways we can make the algorithm more parallelizable, without changing it, is to increase the population size. But, as can be seen in chapter 5, a big population size can make the algorithm slower. First, tests are made to see if the algorithm is more scalable when we double the population size.

Figure 7.2 shows that the algorithm does scale better with a bigger population, but unfortunately the total cpu time is much higher, now 2.34s compared to 1.75s, when running on one thread. The average run time with 12 threads is 0.34s, a little better. But the increase in population size makes the search worse, eating up most of the gain in scalability. It might be possible to compensate for this by increasing selection pressure.

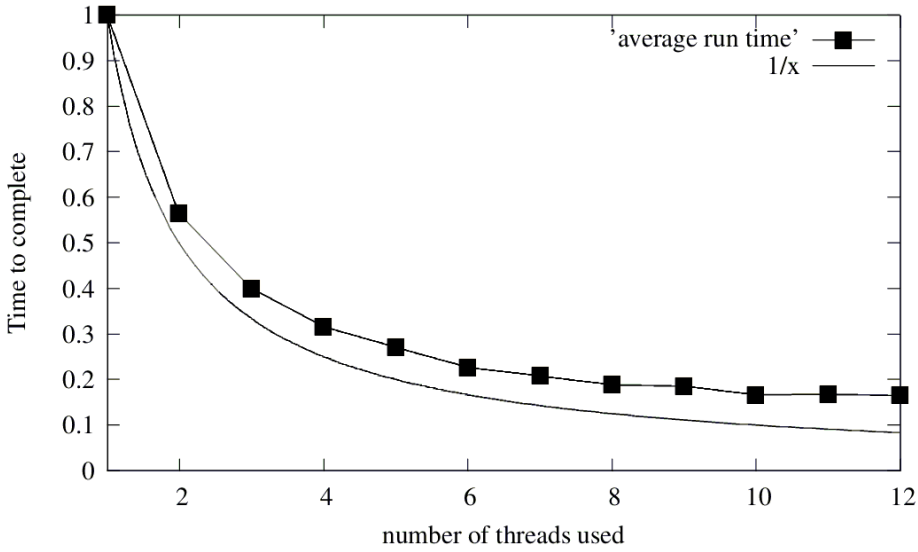


Figure 7.1: How run time changes with multiple threads.

Increasing the number of tournament participants gives us better run-time single-threaded. It now takes 1.83s, which is slightly worse than the original. However, the scalability is similar to the previous run. When we look at figure 7.3 we see that the new configuration outperforms the old one when multiple threads are used. The gain is significant. The old time was 0.37s and with this set-up the optimal solution is found in 0.26s on average. This shows how sacrificing run-time when single threaded can give higher performance when running on many threads.

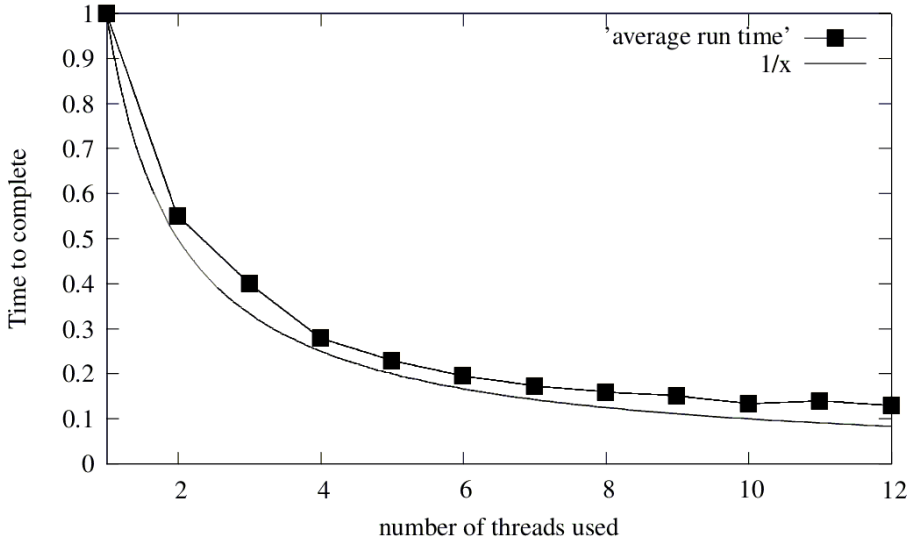


Figure 7.2: How run time changes with multiple threads, with twice as big population.

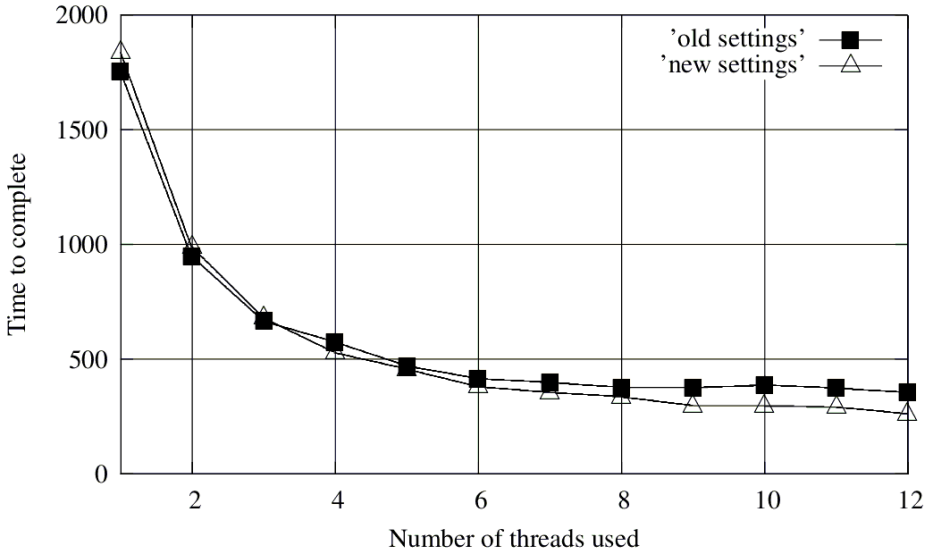


Figure 7.3: How run time (ms) changes with multiple threads, with larger population and higher selection pressure.

# Chapter 8

# Testing protocol

For evaluating the proposed algorithm some extensive testing was done, using multiple metrics on many datasets. These are described in this chapter.

## 8.1 Testing feature selection

These are the tests performed with the algorithm.

### 8.1.1 Cross-validation

To cross-validate the algorithm it is run multiple times with altering training and testing data. So for the proposed algorithm this means doing the feature selection, not the fitness function, with multiple subsets of the available data. From this we can get both the resulting classifiers average performance and the deviation in performance.

For these tests 10-fold cross-validation is done, this is most popular in the literature.

For the feature selection algorithm it is also interesting to know how many features we end up with on average, this can also be obtained from the cross-validations.

### 8.1.2 Tanimoto distance

In addition to set-length it is also interesting to know how different the subsets of each fold are. If two subsets each have 5 different features, then they are completely different and the algorithm is unstable. Therefore we also need to measure this difference with some additional metric other than the number of features. These tests include the average Tanimoto distance [Kal+07] of all the feature-subsets from the cross-validation. Higher distance means more stable, which is desirable.

The Tanimoto distance, measures how much overlap there are between two subsets. Given two subsets a and b it is:

$$S(a,b) = 1.0 - \frac{|a| + |b| - 2|a \bigcap b|}{|a| + |b| - |a \bigcap b|} \tag{8.1}$$

If it is 1.0 the subsets are equal and if it is 0.0 there is no overlap.

One small problem with this metric is that it will be higher for bigger subsets simply because the chance for overlap increases[Kal+07]. Therefore the average Tanimoto distance must be seen in context with the number of possible features and the size of the selected subset.

## 8.2 UCI machine learning datasets

One of the challenges of testing this algorithm was obtaining realistic data that can be used for feature selection. Preferably this data should also be used by other similar algorithms as well.

The UCI machine learning repository [AA07] has many available datasets that can be used for testing the presented algorithm. These sets are also popular in the feature selection literature. Therefore a selection of 8 datasets was used for these tests.

A short summary of the sets and their characteristics follow:

**Wdbc**
: A dataset with 30 continuous attributes computed for 569 images of breast masses. The dataset also contains one ID number and a nominal diagnosis value for each instance. This is one of the most popular datasets in the literature.

**Ionosphere**
: A dataset based of electromagnetic return-signals from the ionosphere. The signals are described by 34 continuous attributes, based on 17 pulse numbers. The 351 instances are divided into good and bad returns based on whether they show evidence of structure in the ionosphere.

**Bupa**
: This dataset consists of 5 blood test values and number of half-pints each day for 345 individuals. The task is to figure out if they suffer from disorders or not.

**Pima**
: This is a dataset with the results of 8 different medical metrics, such as BMI, and a class attribute indicating whether the patient has diabetes. In total the dataset has 768 instances. According to [AA07] this set probably contains missing values (some of the zero values are supposedly impossible) but this was not taken into consideration when the tests where run.

**Wine**

A dataset with 13 attributes from different wine types, the wine types are derived from 3 different cultivars. The task is to predict the type based on a wines attributes. The set contains a total of 178 wines. This set was used for the previous benchmarks.

**Iris**

This dataset contains 4 plant measurements in cm and has 3 classes, each is an iris plant type. There are 150 instance, 50 of each class.

**Sonar**

The dataset is made from reading sonar signals with to different objects as reflectors, a rock and a cylinder (this represents a mine). They make up the two classes of the instances. Each instance has 60 numbers which are the energy levels at different times and frequencies, they are normalized into the range of 0.0 - 1.0. The original set is sorted depending on the angle used, but to avoid any bias it's randomized for the tests. There is only one instance for each angle.

**Vote**

This dataset contains the voting history of 435 representatives from the democratic and the republican party. The task is to determine the party by looking at 16 different votes.

## 8.3 Adding noise to datasets

One of the aspects of a good feature selection algorithm is it's ability to reduce the dimensionality of the feature-space by removing useless features. An easy way to test this would be to take a dataset with a few relevant variables, add noise to it, and see if it can filter this noise. The noise variables are then known to be useless and any variable from the original set is potentially useful. A good feature selection algorithm should therefore only select a subset of the original attributes.

For the noise reduction tests, continuous noise variables are added to the datasets from 8.2. These variables have a range of either -1.0-1.0 or 0.0-1.0. The values are randomly or normally distributed. This noise is created by a random number generator seeded by the number of instances. The generator also decides the range and distribution pattern.

## 8.4 Big datasets

A couple of big gene-expression datasets from http://www.gems-system.org/ where used to validate the algorithm's performance on datasets that are useless without proper fea-

ture selection. These tests are interesting because they contain thousands of features and very few samples, fifty to a few hundred. Note that the abbreviations used are the same as in [AlS+10].

The goal of these datasets is to determine tissue types; different types of cancer or normal tissues. The sets have 2 to 26 different tissue types. The accuracy might be a little miss-leading as a metric, we can not expect that much accuracy when there are 26 different classes. The datasets are summarized in table 8.1.

Table 8.1: High dimensional gene-expression data

| dataset | name | # features | # classes | # samples |
|---|---|---|---|---|
| lc | lung-cancer | 12600 | 5 | 203 |
| pt | prostate tumor | 10509 | 2 | 102 |
| l1 | leukaemia 1 | 5328 | 3 | 72 |
| l2 | leukaemia 2 | 11226 | 3 | 72 |
| bt1 | brain tumor 1 | 5920 | 5 | 90 |
| bt2 | brain tumor 2 | 10368 | 4 | 50 |
| t9 | 9 tumors | 5726 | 9 | 60 |
| t11 | 11 tumors | 12533 | 11 | 174 |
| t14 | 14 tumors | 15009 | 26 | 308 |
| dl | lymphomas | 5469 | 2 | 77 |
| sr | Small, round blue cell tumors | 2309 | 4 | 83 |

## 8.5 Settings

For the tests we use the proposed algorithm with the following settings:

**Initial population:** one of each feature

**population size:** 80

**mutation chance:** 0.7

**crossover rate:** 0.4

**group size:** 4

**random chance:** 0.2

**elitism:** 1

**Length penalty:** 0.005

**Threads:** 24

**Wrappers:** IB1, IB5, j48, NaiveBayes, SMO

**Number of folds:** 10

## 8.6 Result metrics

This section summarizes the result metrics:

**dataset**
: What dataset is used.

**# features**
: How many variables are there in the dataset, this includes the class index.

**original**
: The error rate (and deviation) when the original machine learning algorithm is used without feature selection. Not included for section 9.3.

**error**
: The error rate when the induction algorithm is used with feature selection, standard deviation in parenthesis.

**fitness**
: The fitness score of the highest ranking individual (actually it is 1.0-fitness) at the end of the evolution. This variable includes length penalty, which means the actual error rate it is based on is lower. Standard deviation is shown inside parenthesis.

**# selected**
: The number of features in the proposed solutions, lower number equals less complex solutions. Standard deviation is shown inside parenthesis.

**Tanimoto**
: A metric for how much overlap there is between the different solutions of the cross-validation. A higher number means the algorithm is stable.

# Chapter 9

# Test results

This chapter contains the test results and analysis of them. It is divided into one for the simple machine learning data from [AA07], one for the same datasets with additional noise added, one for the high dimensional datasets from [Gem]. Finally the algorithm is compared to other reported results on three other data-sets.

## 9.1 Normal datasets

Table 9.1-9.5 show the results of the algorithm running on the normal datasets. There is one table for each induction algorithm.

Table 9.1: uci tests with naive bayes as the fitness function

| dataset | # features | original | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|---|
| bupa | 7 | .464(.072) | .403(.096) | .383(.011) | 3.1(.7) | .613 |
| wine | 14 | .022(.037) | .044(.054) | .039(.003) | 4.4(.663) | .587 |
| vote | 17 | .099(.048) | .044(.035) | .049(.004) | 1.0(.0) | 1.0 |
| sonar | 61 | .322(.094) | .269(.07) | .21(.011) | 6.0(1.673) | .246 |
| iris | 5 | .047(.043) | .047(.043) | .049(.004) | 1.5(.5) | .507 |
| pima | 9 | .241(.048) | .247(.057) | .243(.008) | 2.8(.872) | .677 |
| wdbc | 32 | .067(.038) | .039(.022) | .045(.002) | 3.1(.3) | .469 |
| ionosphere | 35 | .183(.091) | .114(.054) | .097(.005) | 5.2(.98) | .36 |

Table 9.2: uci tests with SMO as the fitness function

| dataset | # features | original | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|---|
| bupa | 7 | .417(.055) | .42(.052) | .42(.006) | .0(.0) | 1.0 |
| wine | 14 | .022(.027) | .022(.037) | .036(.002) | 5.9(.831) | .654 |
| vote | 17 | .039(.029) | .044(.035) | .049(.004) | 1.0(.0) | 1.0 |
| sonar | 61 | .201(.084) | .254(.093) | .19(.014) | 7.2(1.249) | .266 |
| iris | 5 | .047(.043) | .047(.043) | .046(.004) | 1.6(.49) | .6 |
| pima | 9 | .23(.056) | .254(.049) | .248(.007) | 3.6(.8) | .612 |
| wdbc | 32 | .023(.019) | .046(.02) | .049(.002) | 3.3(.64) | .366 |
| ionosphere | 35 | .12(.03) | .134(.092) | .123(.009) | 3.5(1.5) | .449 |

Table 9.3: uci tests with IB1 as the fitness function

| dataset | # features | original | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|---|
| bupa | 7 | .382(.061) | .443(.057) | .36(.011) | 4.2(.4) | .641 |
| wine | 14 | .045(.049) | .039(.05) | .038(.003) | 5.9(1.044) | .632 |
| vote | 17 | .085(.047) | .053(.036) | .054(.005) | 2.6(1.02) | .414 |
| sonar | 61 | .149(.101) | .149(.089) | .097(.009) | 9.2(1.327) | .288 |
| iris | 5 | .04(.033) | .04(.033) | .051(.008) | 2.8(.6) | .8 |
| pima | 9 | .301(.052) | .32(.058) | .297(.01) | 3.1(.3) | .625 |
| wdbc | 32 | .047(.019) | .067(.027) | .048(.004) | 3.9(.539) | .38 |
| ionosphere | 35 | .143(.054) | .143(.081) | .076(.008) | 4.9(1.044) | .215 |

Table 9.4: uci tests with IB5 as the fitness function

| dataset | # features | original | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|---|
| bupa | 7 | .382(.061) | .446(.062) | .357(.011) | 3.8(.98) | .526 |
| wine | 14 | .045(.049) | .034(.051) | .038(.003) | 5.9(1.044) | .655 |
| vote | 17 | .078(.045) | .053(.036) | .046(.003) | 2.9(1.136) | .416 |
| sonar | 61 | .149(.101) | .149(.069) | .096(.007) | 9.8(.872) | .31 |
| iris | 5 | .04(.033) | .027(.033) | .038(.004) | 2.0(.0) | 1.0 |
| pima | 9 | .301(.052) | .305(.038) | .293(.008) | 2.0(1.0) | .585 |
| wdbc | 32 | .047(.019) | .067(.027) | .048(.004) | 3.9(.539) | .392 |
| ionosphere | 35 | .143(.054) | .134(.056) | .077(.008) | 5.0(.894) | .154 |

Table 9.5: uci tests with J48 as the fitness function

| dataset | # features | original | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|---|
| bupa | 7 | .342(.068) | .353(.056) | .33(.012) | 3.2(1.249) | .621 |
| wine | 14 | .067(.048) | .061(.052) | .047(.01) | 3.6(.49) | .501 |
| vote | 17 | .046(.032) | .044(.035) | .049(.004) | 1.0(.0) | 1.0 |
| sonar | 61 | .284(.072) | .255(.092) | .143(.018) | 7.1(.943) | .208 |
| iris | 5 | .047(.043) | .06(.047) | .055(.008) | 1.3(.458) | .593 |
| pima | 9 | .265(.047) | .263(.029) | .259(.012) | 2.4(1.02) | .434 |
| wdbc | 32 | .072(.033) | .047(.016) | .051(.003) | 3.3(.458) | .537 |
| ionosphere | 35 | .1(.055) | .08(.054) | .077(.005) | 4.1(.831) | .483 |

### 9.1.1 Analysis

To start with we can see clearly that the algorithm is able to come up with good solutions to the feature-subset selection problem, finding subsets that maximize accuracy on training data. In most tests the reported accuracy on training data is higher than the accuracy of the wrapper-algorithm on the original dataset. The numbers do not make it that obvious, however, for example on the pima dataset with the SMO wrapper the reported fitness is .248, this is equivalent to an average error rate of 0.23 when we take the length penalty into consideration. On the wdbc dataset, on the other hand, the SMO based feature selection does not reach a good optimum. Its error rate (0.0325) is slightly higher than what is achieved with all the features. However, it is still better from the point of view of the algorithm, considering that the original dataset would give a total length penalty of 0.155. This shows that a length penalty higher than slightly below zero can give suboptimal solutions. There is two more example of this behaviour; the vote test with SMO and the iris test with the J48 wrapper.

The results show that the wrapper fitness in many cases does not relate well to good performance on instances not used in the feature selection. The high fitness could be a result of over-fitting. In most cases the difference is a few percent, but a few cases have very big difference. For example the bupa test with the IB1 wrapper has an error rate that is on average 0.104 higher on unseen data.

The lack of improvements could be because many of these datasets already are noise free and there is little to gain from the removal of features. In many of the tests this difference gives the algorithm a worse error rate than the original feature-set.

24 of the 40 tests show a reduction in dimensionality with lower or equal error rate on unseen data. The difference in error rate is rarely big, though there are a couple of exceptions. The bupa dataset seems to get high variations in the error rate from feature selection but depending on the wrapper used feature selection is both positive and negative. The same is true for the ionosphere dataset with a significant gain with

naive bayse wrapper and a small loss of performance with the SMO.

The SMO error rate does not seem to benefit much from feature selection. In none of the tests does feature selection lower it. This is probably because the SMO is able to get close to optimal error rate on all tests (except bupa) without feature selection.

When we look at the number of selected features, the results are clear. The algorithm is capable of significantly reducing the dimensionality of the data. The highest number of average features is 9.8 on the sonar set containing 60 candidate features. In all the tests the dimensionality is reduced significantly. For the vote dataset it is often reduced down to only 1 feature, and in these cases we get an error rate of 0.044.

The smallest set selected was 0, for the bupa dataset with SMO as the wrapper, suggesting that guessing the most common class was the only model, and looking at the error rate when using all 6 features, this is not far from the truth.

In general we see that we do get much smaller feature-sets and induction models with the feature selection algorithm.

The stability varies, and is also dependent on the wrapper function used. The Vote dataset was often completely stable. But, when using the IB1 and IB5 classifiers, more than one vote was selected and the stability is drastically reduced. There are two more examples of no set distance. The iris (lowest dimensionality) with the IB5 classifier was completely consistent through each part of the cross-validation using the same 2 features and the empty feature selection for bupa when using the SMO. The SMO seems to be the most stable, if we rank the wrappers by how stable they are on each set. It is also the wrapper that selects fewest features. This is mainly because of the way it handles the pima dataset. In general the wrappers get similar scores on this ranking. No one is much better than the other because they do well and poorly on different sets. The instance based wrappers (IB1 and IB5) on average select more features than the rest. More features on average should give higher Tanimoto. Unfortunately this does not seem to be the case here. This suggests that there might be a local maximum for stability that these tests overshoot.

## 9.2 With noise

Tables 9.6-9.10 show the results when the same tests are done with 400 additional noise variables added to each dataset.

### 9.2.1 Analysis

When we compare the original induction algorithms with and without noise we can see that in general the accuracy is worse with added noise, especially for the instance based classifiers. For example for the iris dataset using IB5 gets an error rate of .493 compared to .04 without noise. However, there are some cases where the error actually becomes smaller when adding noise.

Table 9.6: uci tests with noise, with naive bayes as the fitness function

| dataset | # features | original | error | fitness | # selected | # noise | Tanimoto |
|---|---|---|---|---|---|---|---|
| bupa | 407 | .409(.09) | .525(.113) | .33(.019) | 6.1(1.136) | 5.9(.943) | .072 |
| wine | 414 | .039(.05) | .039(.043) | .038(.003) | 5.0(.775) | .9(.7) | .519 |
| vote | 417 | .096(.043) | .044(.035) | .049(.004) | 1.0(.0) | .0(.0) | 1.0 |
| sonar | 461 | .299(.112) | .341(.087) | .202(.012) | 6.1(2.071) | 3.2(1.661) | .123 |
| iris | 405 | .093(.074) | .087(.085) | .033(.006) | 3.4(.8) | 2.4(.8) | .183 |
| pima | 409 | .269(.055) | .238(.057) | .236(.008) | 3.8(.6) | 1.7(.458) | .484 |
| wdbc | 432 | .069(.039) | .037(.024) | .045(.002) | 3.5(.5) | .5(.5) | .393 |
| ionosphere | 435 | .171(.095) | .1(.063) | .101(.007) | 4.0(.894) | .3(.458) | .492 |

Table 9.7: uci tests with noise, with SMO as the fitness function

| dataset | # features | original | error | fitness | # selected | # noise | Tanimoto |
|---|---|---|---|---|---|---|---|
| bupa | 407 | .412(.071) | .42(.052) | .42(.006) | .0(.0) | .0(.0) | 1.0 |
| wine | 414 | .169(.071) | .04(.045) | .034(.003) | 6.0(.894) | 1.2(.748) | .62 |
| vote | 417 | .046(.042) | .044(.035) | .049(.004) | 1.0(.0) | .0(.0) | 1.0 |
| sonar | 461 | .278(.064) | .269(.075) | .202(.017) | 6.9(1.814) | 3.4(1.562) | .139 |
| iris | 405 | .22(.085) | .053(.04) | .034(.006) | 3.9(1.044) | 2.3(.9) | .254 |
| pima | 409 | .361(.058) | .245(.064) | .24(.01) | 4.1(.7) | 2.0(.447) | .301 |
| wdbc | 432 | .065(.019) | .051(.02) | .05(.002) | 2.9(.539) | .3(.458) | .353 |
| ionosphere | 435 | .188(.067) | .117(.063) | .124(.007) | 3.3(.781) | .9(.7) | .434 |

Table 9.8: uci tests with noise, with IB1 as the fitness function

| dataset | # features | original | error | fitness | # selected | # noise | Tanimoto |
|---|---|---|---|---|---|---|---|
| bupa | 407 | .46(.079) | .516(.048) | .35(.016) | 2.7(1.1) | 2.6(.917) | .011 |
| wine | 414 | .393(.111) | .073(.05) | .037(.005) | 5.6(.8) | 1.4(.663) | .384 |
| vote | 417 | .092(.051) | .051(.03) | .054(.005) | 2.2(.6) | .3(.458) | .446 |
| sonar | 461 | .41(.143) | .201(.09) | .123(.012) | 7.6(1.428) | .7(.64) | .152 |
| iris | 405 | .493(.068) | .1(.061) | .024(.006) | 3.4(.8) | 1.7(.64) | .283 |
| pima | 409 | .409(.035) | .346(.046) | .277(.007) | 4.2(.872) | 2.2(.748) | .258 |
| wdbc | 432 | .183(.057) | .054(.024) | .048(.004) | 3.9(.7) | .4(.49) | .313 |
| ionosphere | 435 | .236(.078) | .131(.041) | .083(.008) | 4.9(1.136) | .7(.64) | .249 |

One example would be the sonar dataset with NaiveBayes, the classifier improves its error rate by .023. NaiveBayes handles real variables by creating Gaussian distributions,

Table 9.9: uci tests with noise, with IB5 as the fitness function

| dataset | # features | original | error | fitness | # selected | # noise | Tanimoto |
|---|---|---|---|---|---|---|---|
| bupa | 407 | .46(.079) | .44(.087) | .36(.012) | 1.9(.7) | 1.4(1.02) | .256 |
| wine | 414 | .393(.111) | .05(.052) | .037(.005) | 5.6(1.2) | 1.5(.922) | .386 |
| vote | 417 | .092(.051) | .048(.039) | .048(.004) | 2.0(.632) | .0(.0) | .719 |
| sonar | 461 | .41(.143) | .216(.093) | .124(.022) | 6.4(1.685) | .4(.49) | .168 |
| iris | 405 | .493(.068) | .073(.063) | .024(.007) | 3.1(.831) | 1.5(.671) | .266 |
| pima | 409 | .409(.035) | .324(.05) | .284(.008) | 3.0(1.483) | 1.3(1.1) | .305 |
| wdbc | 432 | .183(.057) | .054(.03) | .048(.004) | 3.9(.539) | .5(.5) | .364 |
| ionosphere | 435 | .236(.078) | .143(.059) | .079(.008) | 4.2(.872) | .5(.5) | .254 |

Table 9.10: uci tests with noise, with J48 as the fitness function

| dataset | # features | original | error | fitness | # selected | # noise | Tanimoto |
|---|---|---|---|---|---|---|---|
| bupa | 407 | .42(.078) | .377(.06) | .292(.013) | 6.7(1.1) | 4.5(1.285) | .197 |
| wine | 414 | .067(.048) | .045(.049) | .047(.01) | 3.7(.458) | .4(.49) | .515 |
| vote | 417 | .071(.051) | .048(.031) | .047(.003) | 2.0(.894) | .8(.6) | .444 |
| sonar | 461 | .275(.094) | .322(.095) | .16(.019) | 7.3(1.1) | 3.7(1.187) | .078 |
| iris | 405 | .08(.04) | .087(.052) | .041(.007) | 2.6(.663) | 1.5(.806) | .193 |
| pima | 409 | .324(.069) | .269(.048) | .247(.011) | 4.3(.64) | 2.9(.831) | .171 |
| wdbc | 432 | .098(.03) | .055(.012) | .055(.005) | 3.4(.663) | .8(.6) | .17 |
| ionosphere | 435 | .151(.08) | .077(.036) | .079(.004) | 4.1(.7) | .2(.4) | .548 |

so it is possible that a false pattern is removed when the noise variables are added. The standard deviation for this classifier is large so this could also be a statistical error.

In all these tests the algorithm is able to find pretty good optimisations for the fitness function. Actually, many of them are higher than the fitnesses found during the noise free testing. This suggests that the algorithm is over-fitting the data by using noise variables that proved slightly useful during the fitness cross-validation. Apart from this, the new fitnesses are very similar to the old.

These concerns are verified when we compare the fitness with the error rate on instances not used in the fitness function. The biggest difference is also this time seen in the bupa dataset. The error rate reported, when using the naive bayes classifier for fitness, is 0.3 but the error on unseen examples is 0.525, a decrease in accuracy of 0.225, leaving us with a model that is worse than flipping a coin. This shows that the variables in the bupa dataset do not relate all that well to any concept, but also that the feature selection algorithm performs very poorly if the pattern is weak and there is a lot of noise.

All this aside, the feature selection proved useful in most of the tests. 34 performed

equal or better with fewer features. Three of the times feature selection reduced the accuracy was on bupa (Naive bayes, SMO and IB1), two times on sonar (Naive bayes and J48) and one time on the iris dataset (J48). On all the other tests there was a positive effect, especially for the lazy classifiers. For example IB5 had an error rate of 0.493 and with feature selection the average error was as low as 0.73.

Unlike on the original set the SMO seems to really benefit from feature selection. All tests, except bupa, perform better and the result is actually the same as we got with no artificial noise at all.

The number of selected features is still pretty low, in most cases it is slightly higher on average than with the original datasets. But in some cases, especially with the instance based models, the number of selected features is reduced. This is probably because it is harder for the algorithm to land on the right features in the same number of generations as before when there are much more to choose from. These classifiers are especially vulnerable to noise so using suboptimal variables that compensates is also more difficult.

The size of the selected feature-sets are similar, but these results can also tell us if the selection algorithm can remove the variables that are unrelated to the underlying concepts. If a randomized variable was selected by the algorithm when the punishment for each included is more than zero, it suggests that it is used to over-fit the model to score a higher fitness. When we analyse the amount of noise selected, it puts some of the results in perspective.

We can see why Naive bayes performed so poorly on the bupa set, pretty much every selected variable through the 10 folds was a random variable and it only took 6 of them to get a significant increase in accuracy on the training data. This obviously did not generalize well. In fact, the only algorithm able to select on average 1 or more variables from the real dataset was j48, the algorithm with the highest accuracy without noise.

The IB5 algorithm was the best algorithm for filtering noise. On 4 of the tests it has the lowest noise levels. This is most likely because it is very sensitive to noise, as can be seen in the big drop in performance the added noise gives. This could give it an advantage over other algorithms when the dimensionality increases.

In general, all the algorithms perform pretty well noise vice, given that about 9/10 variable are noise but only about 1/3 of the selected are. The error rate after selection is also not that different from the previous test. However, the stability has taken a turn for the worse. Most of the Tanimoto measures are worse as one would expect when most of the tests contain noise variable over-fitted for each fold. With these Tanimoto measures we do not need to take the set size into account because the number of features makes the increase we can attribute to size insignificant.

Some of the solutions in the tests most likely do not have one feature in common with each other. It is clearly the case with some of the bupa tests. The sonar test with J48 must also contain solutions like this. On the tests with poorest results there seems to be little stability in the solutions and big difference between fitness and error rate.

But on the wdbc and pima tests with J48 we do not see a big loss in accuracy. So poor performance comes with low stability, but not necessarily the other way around.

Overall the SMO still is more stable than the rest the of the algorithms. Naive bayes and IB5 have similar stability as the SMO. NB1 and J48, however, seem to be much more unstable than the rest. This could be because they are more likely to over-fit their training data.

## 9.3 Big datasets

Tables 9.11-9.15 show the results from using the same approaches on a couple of high-dimensional micro arrays.

Some of the SMO-tests, on the problems with many classes, took too long to finish and could not be included.

Table 9.11: tests using high-dimensional datasets with naive bayes as the fitness function

| dataset | # features | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|
| lc | 12601 | .094(.061) | .064(.009) | 6.7(.781) | .071 |
| pt | 10510 | .119(.087) | .048(.008) | 3.7(1.005) | .119 |
| l1 | 5328 | .112(.107) | .028(.006) | 5.4(1.428) | .08 |
| l2 | 11226 | .136(.151) | .029(.007) | 5.6(1.2) | .024 |
| bt1 | 5921 | .3(.165) | .083(.016) | 6.6(1.02) | .023 |
| bt2 | 10368 | .34(.156) | .059(.02) | 6.3(1.345) | .017 |
| t9 | 5727 | .6(.17) | .251(.057) | 9.7(1.792) | .023 |
| t11 | 12534 | .213(.106) | .16(.015) | 13.8(1.99) | .019 |
| t14 | 15010 | .49(.074) | .438(.022) | 14.2(1.6) | .018 |
| dl | 5470 | .155(.126) | .029(.003) | 5.8(.6) | .027 |
| sr | 2309 | .097(.119) | .037(.009) | 7.4(1.8) | .066 |

### 9.3.1 Analysis

In many of the tests the algorithm is still able to find a high fitness value. For the tests on lc, pr, l1, l2, dl and sr the error rate the fitness is based on would be zero or very close in all of the tests. On the bt1 and bt2 datasets the fitness function does not get the error rate that low, however, it is always below .1. For the multi-class datasets t9, t11 and t14 the error rate is considerably higher. t9 gets an error rate between .2025(Naive bayes) and .3115 (J48). t11 does better and the error rate on the training data is between .091 (Naive bayes) and .1885 (J48). On t14, with 26 different classes, the accuracy on the training data was between .367 (Naive bayes) and .4805 (J48). Naive bayes consistently

Table 9.12: tests using high-dimensional datasets with IB1 as the fitness function

| dataset | # features | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|
| lc | 12601 | .098(.052) | .065(.008) | 7.4(1.114) | .013 |
| pt | 10510 | .177(.152) | .039(.007) | 4.5(.922) | .077 |
| l1 | 5328 | .17(.154) | .024(.006) | 4.6(1.2) | .064 |
| l2 | 11226 | .082(.093) | .024(.006) | 4.2(.872) | .189 |
| bt1 | 5921 | .267(.166) | .08(.016) | 6.9(.831) | .018 |
| bt2 | 10368 | .36(.15) | .055(.012) | 5.8(1.4) | .002 |
| t9 | 5727 | .733(.186) | .292(.049) | 8.0(1.949) | .011 |
| t11 | 12534 | .327(.086) | .195(.02) | 10.9(1.513) | .043 |
| t14 | 15010 | .588(.128) | .474(.023) | 13.0(1.265) | .003 |
| dl | 5470 | .154(.134) | .024(.006) | 4.5(1.204) | .036 |
| sr | 2309 | .086(.097) | .036(.004) | 7.0(.775) | .08 |

Table 9.13: tests using high-dimensional datasets with IB5 as the fitness function

| dataset | # features | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|
| lc | 12601 | .138(.061) | .063(.008) | 7.5(1.204) | .015 |
| pt | 10510 | .139(.12) | .037(.011) | 3.8(.6) | .208 |
| l1 | 5328 | .109(.078) | .025(.004) | 4.7(.64) | .056 |
| l2 | 11226 | .109(.136) | .025(.005) | 4.4(.8) | .181 |
| bt1 | 5921 | .211(.092) | .071(.012) | 6.6(.663) | .017 |
| bt2 | 10368 | .38(.189) | .05(.017) | 6.1(1.578) | .013 |
| t9 | 5727 | .6(.249) | .3(.039) | 8.4(1.908) | .016 |
| t11 | 12534 | .273(.136) | .194(.026) | 11.0(1.483) | .034 |
| t14 | 15010 | .601(.091) | .475(.02) | 14.6(1.497) | .01 |
| dl | 5470 | .116(.119) | .022(.005) | 4.2(1.166) | .035 |
| sr | 2309 | .062(.062) | .029(.007) | 5.8(1.327) | .136 |

finds classifiers with higher accuracy on the training data and J48 does not perform very well.

Unfortunately the gap between fitness accuracy and error rate on unseen data is very clear in these tests. It is most obvious for the t9 dataset, this does not get much out of the feature selection at all. The error rate is between .6 and .767, granted there are 9 classes. But this is nowhere near the accuracy on the training data mirrored by the fitness. bt2 and t9 seem to get the biggest gaps. These are also the smallest datasets, so it could be that the cross-validation used does not leave enough of a pattern in the training data and features with outliers in the validation-set get picked.

Table 9.14: tests using high-dimensional datasets with J48 as the fitness function

| dataset | # features | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|
| lc | 12601 | .133(.071) | .069(.01) | 5.4(.663) | .027 |
| pt | 10510 | .148(.121) | .055(.014) | 3.1(.943) | .079 |
| l1 | 5328 | .141(.111) | .048(.012) | 2.3(.458) | .122 |
| l2 | 11226 | .136(.147) | .062(.016) | 2.6(.8) | .06 |
| bt1 | 5921 | .311(.109) | .111(.017) | 4.7(.458) | .006 |
| bt2 | 10368 | .58(.189) | .1(.027) | 3.8(.6) | .004 |
| t9 | 5727 | .767(.111) | .341(.047) | 5.9(.943) | .021 |
| t11 | 12534 | .403(.1) | .235(.022) | 9.3(1.269) | .038 |
| t14 | 15010 | .627(.101) | .532(.023) | 10.3(1.269) | .056 |
| dl | 5470 | .195(.147) | .048(.018) | 2.1(.7) | .19 |
| sr | 2309 | .122(.112) | .073(.018) | 3.7(.781) | .063 |

Table 9.15: tests using high-dimensional datasets with SMO as the fitness function

| dataset | # features | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|
| lc | 12601 | .112(.075) | .083(.008) | 9.9(1.578) | .016 |
| pt | 10510 | .158(.068) | .046(.006) | 6.4(1.744) | .047 |
| l1 | 5328 | .125(.077) | .035(.008) | 5.2(1.833) | .103 |
| l2 | 11226 | .138(.116) | .045(.006) | 8.5(1.91) | .018 |
| bt1 | 5921 | .233(.126) | .119(.009) | 8.6(1.281) | .021 |
| bt2 | 10368 | .44(.233) | .07(.011) | 9.1(1.972) | .008 |
| t9 | 5727 | .617(.198) | .253(.046) | 18.8(2.441) | .004 |
| - | - | - | - | - | - |
| - | - | - | - | - | - |
| dl | 5470 | .102(.094) | .038(.006) | 6.8(1.887) | .035 |
| sr | 2309 | .11(.118) | .047(.007) | 9.5(1.36) | .065 |

When we look at some of the more positive results, it is clear that useful models where created for lc, pt, l1, l2, dl and sr. The algorithm was able to do this with every wrapper.

The number of selected features is still very small. the largest average is 14.6, much bigger than anything seen in the other tests but the t14 also has much more classes than anything else that has been tested. Given that there are 26 different classes, this might not be enough, it is possible that the length penalty is holding back the accuracy. On the other hand; many of the tests come close to no error in the fitness evaluation, and without a length penalty there would not be any improvements to make.

By ranking the different wrappers based on how well they work on each of the sets (1 = best, 5 = worst) we can analyse how well the different wrappers performed. See table 9.16.

Table 9.16: Wrappers ranked by error on unseen data

| wrapper | avg rank |
|---|---|
| Naive Bayes | 2.0 |
| IB5 | 2.1 |
| IB1 | 2.82 |
| SMO | 3.25 |
| J48 | 4.2 |

The results show that NaiveBayes performed best on average, closely followed by IB5. IB5 did well on the noise-test, so this is no surprise. We also see that J48 ranked very poorly. This is no surprise , as it did not do well on the noise test either. The results for the SMO are not reliable as only 8 out of 11 ranks are included in this average.

There is no similar trend if we rank the wrappers by how high their Tanimoto distance is, see table 9.17.

Table 9.17: Wrappers ranked by their Tanimoto distance

| wrapper | avg rank |
|---|---|
| Naive Bayes | 2.45 |
| IB5 | 2.82 |
| IB1 | 3.18 |
| SMO | 3.38 |
| J48 | 2.64 |

In fact, the Tanimoto is so low in most cases that the only thing it indicates is that the feature selection in this domain was very unstable. This is also reflected in very big deviations in the error rates.

So, the conclusion is that with data of this dimensionality, the feature selection algorithm is very capable of finding a feature-subset that fits the training-data. However, models created using this model will probably not generalize well to unseen data, even with a close to zero error rate on the training data. The wrappers that are most suited are IB5 and NaiveBayes whereas J48 is not suited at all.

### 9.3.2 Follow up tests

Based on the analysis new tests were run, only using Naive Bayes and IB5.

- The length penalty was set to near zero.
- Population size was increased to 500
- Number of iterations increased to 2500.

Table 9.18: bigTests-class weka.classifiers.bayes.NaiveBayes.tex

| dataset | # features | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|
| lc | 12601 | .108(.065) | .0(.0) | 17.5(3.905) | .021 |
| pt | 10510 | .139(.08) | .003(.005) | 7.0(2.049) | .089 |
| l1 | 5328 | .086(.095) | .0(.0) | 6.9(1.513) | .038 |
| l2 | 11226 | .098(.128) | .0(.0) | 7.2(1.536) | .019 |
| bt1 | 5921 | .178(.124) | .005(.008) | 10.5(3.557) | .018 |
| bt2 | 10368 | .42(.14) | .0(.0) | 9.3(2.002) | .009 |
| t9 | 5727 | .633(.145) | .047(.018) | 14.4(2.059) | .045 |
| t11 | 12534 | .212(.117) | .007(.008) | 26.9(8.814) | .014 |
| t14 | 15010 | .448(.109) | .215(.016) | 36.4(4.883) | .026 |
| dl | 5470 | .077(.084) | .0(.0) | 7.3(2.238) | .027 |
| sr | 2309 | .178(.106) | .0(.0) | 8.6(1.2) | .03 |

Table 9.19: bigTests-class weka.classifiers.lazy.IBk.tex

| dataset | # features | error | fitness | # selected | Tanimoto |
|---|---|---|---|---|---|
| lc | 12601 | .094(.065) | .0(.0) | 16.3(5.08) | .01 |
| pt | 10510 | .198(.142) | .0(.0) | 9.1(2.663) | .045 |
| l1 | 5328 | .138(.166) | .0(.0) | 5.0(1.732) | .051 |
| l2 | 11226 | .127(.119) | .0(.0) | 6.6(2.538) | .019 |
| bt1 | 5921 | .211(.144) | .0(.0) | 13.3(2.41) | .015 |
| bt2 | 10368 | .36(.215) | .0(.0) | 10.0(2.324) | .012 |
| t9 | 5727 | .683(.138) | .038(.016) | 16.7(4.291) | .03 |
| t11 | 12534 | .213(.128) | .018(.008) | 22.8(4.936) | .025 |
| t14 | 15010 | .468(.076) | .226(.025) | 37.7(8.05) | .022 |
| dl | 5470 | .139(.118) | .0(.0) | 5.5(1.285) | .007 |
| sr | 2309 | .06(.096) | .0(.0) | 7.8(2.04) | .114 |

The results are pretty similar to the previous runs. The main difference is bigger feature-sets on average, but they are still pretty small. Error rate is different, in some cases better and in some cases worse, but this is expected given the high deviations.

The tests do, however, make the problem more obvious. In almost every test the fitness of the solutions used is 0.0. Even the multi-class problems come pretty close to no error rate. This shows that the main issue is not the search algorithm, but the heuristic used.

## 9.4 Comparisons

Here we compare the proposed algorithm to a couple of other reported results found in the literature review. The previous datasets where found through an article discussing how to avoid local maxima, and are not used by the other micro-array articles, so a couple of new tests where done. None of the others report on stability, therefore it is omitted. The datasets where difficult to obtain. Following the links mentioned in the articles did not always work. Because of this there is no guarantee that the tests are done on exactly the same datasets, they do, however, have the same dimensionality and number of examples. They are from:

**leukemia3:** http://www.broadinstitute.org/mpr/publications/projects/Leukemia/

**colon:** http://genomics-pubs.princeton.edu/oncology/affydata/index.html

**lymphoma:** http://llmpp.nih.gov/lymphoma/data/figure1/figure1.cdt

Table 9.20: comparison of other results reported as: error rate (standard deviation), number of features.

| dataset | Naive Bayes | IB5 | [Mun+06] | [Ban+07] | [Can+10] |
|---|---|---|---|---|---|
| leukemia3 | .094(.036), 2.0 | .103(.037), 2.0 | 0.075, 10.45 | .176-.059, 2-3 | 0.0, 3 |
| colon | .361(.087), 3.1 | .277(.059), 2.8 | - | .273-.09, 8-10 | - |
| lymphoma | .106(.049), 2.0 | .1(.054), 2.0 | - | .083-.042, 2-3 | - |

The results for NaiveBayes and IB5 in table 9.20 are the results of 10 runs with different seeds (and the standard deviation of these runs). The reported tests use separate training and test-sets. Using this technique gives poorer performance than cross-validation because fewer examples are available for training, but this is what the other studies use, so comparison is simpler. 10-fold cross-validation runs using these datasets gave approximately half the reported error rate. It is also worth mentioning that the fitness on every run of the proposed algorithm for these datasets was 0.0.

[Mun+06] uses a genetic programming approach with one tree for each class, the fitness function used is not accuracy but each tree's contribution towards the correct class, this means the algorithm will be able to separate between features and sets of trees that give perfect accuracy. The fitness function in [Ban+07] is based on distinction tables, it also uses filters to reduce initial search space. They report many different results for the datasets, using different candidates and induction algorithms (1NN - 7NN). The proposed algorithm gets similar results using IB5 as the wrapper, slightly higher error rate and fewer features. [Can+10] uses a SVM as fitness function but instead of 10-fold cross-validation a leave-one-out-cross-validation is used, meaning for each training example the induction algorithm is tested after construction from the rest. This technique could give more general feature-subsets, but it is also much more expensive.

The statistics of the test results are not very convincing, because they are done on one small training set and one small validation set. Results could be down to a lucky configuration, particularly for [Can+10] that only use one dataset for validation of the algorithm. [Mun+06] shows good results on multiple datasets, but unfortunately only one with very high dimensionality. The results in [Ban+07] are more reliable because they use multiple datasets, but they report results on multiple settings and the best performance is not achieved with the similar settings on each dataset.

The use of training and test-sets also makes it harder to evaluate the stability of the selection. One could of course slightly modify the training set on multiple runs and measure the difference, but the training-sets are so small that there probably will be no stability anyway.

# Chapter 10

# Summary and Conclusion

## 10.1 Summary

In this thesis the literature on evolutionary approaches to the feature-set selection problem is discussed. A new implementation is created with a generalized wrapper approach capable of using any classifier extending the Classifier super-class from weka. It uses new simplified genetic operators. The algorithm's parameters are extensively tested using an example dataset. The algorithm's run-time and scalability was analysed and enhanced. The resulting algorithm was able to solve the benchmark test in a fraction of the time of the brute-force approach. Extensive tests where then carried out to evaluate the abilities of the algorithm on a wide range of datasets.

The proposed algorithm scales pretty well, but to be able to use more advanced induction algorithms as wrappers one must use multi-threading, this is no big issue with a few threads, but the algorithm was not capable of linear scaling, even if significant improvements where found by tweaking it. To achieve linear speed-up, an asynchronous algorithm must be created. Given that this is a type of algorithm that takes some time to run, reports on scalability and run time is important, unfortunately this is not the norm in the literature, an exception being [Zhu+10].

The results show that genetic algorithms can be used for fast and useful wrapper based feature selection. But they also show a problem with wrapper-based feature selection in general. The test results are plagued by big differences in test-error and holdout-error. This is not due to over-fitting of the wrapper algorithm, but rather an over-fitting of the feature-set with regard to the cross-validation used as in the fitness evaluation.

In most cases performance is improved anyway, but for some of the tests, subsets evaluated to be significantly improved are outperformed by their supersets on unseen instances. This shows that feature selection is more than just finding the optimal subset for training-data accuracy. Optimal accuracy on training-data does not always equal good

generalization. Higher accuracy on test data is one indicator, but it is not optimal because there are so many similar subsets that some are bound to test better by coincidence. This leads to our main conclusion.

## 10.2 Conclusions

The proposed evolutionary approach works well for traversing the subset search-space. This is demonstrated by finding a solution with high fitness on the test-data in every test. The main limitations are problems with the search heuristic.

Both the low dimensional tests and the high-dimensional tests demonstrate that there is a gap between the accuracy used as fitness and the accuracy the final algorithm gets on unseen data, this gap often means the feature selection does not help, especially for data with little noise.

The tests demonstrate a limitation of the wrapper-approach, with accuracy as heuristic, when it comes to dealing with high dimensional datasets. The proposed algorithm is not able to compete with other similar algorithms, not because of some limitations of the search algorithm, but because the wrapper-accuracy is incapable of finding the best subsets when there are many that give no, or similar, error rate. This leads to a big gap between accuracy on training and unseen instances. This problem is especially evident when looking at the results of the tests with high-dimensional datasets with few training examples.

This problem causes the proposed algorithm to perform worse than other examples from the literature, especially on the tests that use separate training and testing-sets, giving fewer training-examples than cross-validation. For wrapper-based feature selection with high dimensionality and few examples an approach similar to what is done in [Mun+06] should be taken, this uses not just accuracy but the certainty of the induction algorithm allowing search among different solutions that classify every example correctly.

The main main conclusion is: Wrapper accuracy is not a sufficient measure for subset fitness with high dimensional datasets, the fitness should also include some metric for differentiation of subsets that show complete accuracy on training data.

Consequently, an abstract approach is difficult to implement. This type of metric would differ between the wrapper algorithms and would in some cases not be available. This is exemplified here by the classifier super-class in weka that does not include any such methods.

# Bibliography


[AA07] D.J. Newman A. Asuncion. *UCI Machine Learning Repository: http://www.ics.uci.edu/∼mlearn/MLRepository.html*. 2007.

[Alf+12] Rayner Alfred, Irwansah Amran, Leau Yu Beng, and Tan Soo Fun. "Unsupervised learning of mutagenesis molecules structure based on an evolutionary-based features selection in DARA". In: vol. 7691 LNAI. 2012, pp. 291 – 299.

[AlS+10] A. AlSukker, R.N. Khushaba, and A. Al-Ani. "Enhancing the diversity of genetic algorithm for improved feature selection". In: 2010//, pp. 1325 –31.

[Bae+10] Changseok Bae, Wei-Chang Yeh, Yuk Ying Chung, and Sin-Long Liu. "Feature Selection with Intelligent Dynamic Swarm and Rough Set". In: *Expert Systems with Applications* 37.10 (2010/10/), pp. 7026 –32.

[Ban+07] M. Banerjee, S. Mitra, and H. Banka. "Evolutionary rough feature selection in gene expression data". In: *IEEE Transactions on Systems, Man, and Cybernetics–Part C (Applications and Reviews)* 37.4 (2007/07/), pp. 622 –32.

[BH+10] Edmundo Bonilla Huerta, J. Crispin Hernandez Hernandez, and L. Alberto Hernandez Montiel. "A new combined filter-wrapper framework for gene subset selection with specialized genetic operators". In: vol. 6256 LNCS. 2010, pp. 250 –259.

[Can+10] Laura Maria Cannas, Nicoletta Dessi, and Barbara Pes. "A filter-based evolutionary approach for selecting features in high-dimensional micro-array data". In: vol. 340 AICT. 2010, pp. 297 –307.

[Cha+13] Alexandros Andre Chaaraoui and Francisco Florez-Revuelta. "Human action recognition optimization based on evolutionary feature subset selection". In: 2013, pp. 1229 –1236.

[Che+12] Bolun Chen, Ling Chen, and Yixin Chen. "Efficient ant colony optimization for image feature selection". In: *Signal Processing* (2012).

[Dat+11] A. Datta, S. Ghosh, and A. Ghosh. “Wrapper based feature selection in hyperspectral image data using self-adaptive differential evolution”. In: 2011//, 6 pp. –.

[Din+09] Sheng Ding and Xiaoming Liu. “Evolutionary computing optimization for parameter determination and feature selection of support vector machines”. In: 2009//, 5 pp. –.

[Dro+10] K. Drozdz and H. Kwasnicka. “Feature Set Reduction by Evolutionary Selection and Construction”. In: vol. pt.2. 2010//, pp. 140 –9.

[DS+08] C. De Stefano, F. Fontanella, and C. Marrocco. “A GA-based feature selection algorithm for remote sensing images”. In: vol. 4974 LNCS. 2008, pp. 285 –294.

[Dur+09] P. Durr, W. Karlen, J. Guignard, C. Mattiussi, and D. Floreano. “Evolutionary selection of features for neural sleep/wake discrimination”. In: *Journal of Artificial Evolution & Applications* (2009//), 179680 (9 pp.) –.

[Els] Elsevier. *Engineering Village: http://www.engineeringvillage.com/.*

[Fun+97] George S.K. Fung, James N.K. Liu, K.H. Chan, and Rynson W.H. Lau. “Fuzzy genetic algorithm approach to feature selection problem”. In: vol. 1. 1997, pp. 441 –446.

[Gem] *http://www.gems-system.org/.*

[Hal+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. “The WEKA Data Mining Software: An Update”. In: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009), pp. 10–18.

[Har+05] M.T. Harandi, M.N. Ahmadabadi, B.N. Araabi, and C. Lucas. “Feature selection using genetic algorithm and it’s application to face recognition”. In: vol. vol.2. 2005//, 6 pp. vol.2 –.

[Hua+07] Hui-Ling Huang and Fang-Lin Chang. “ESVM: Evolutionary support vector machine for automatic feature selection and classification of microarray data”. In: *BioSystems* 90.2 (2007), pp. 516 –528.

[Joh+94] George H John, Ron Kohavi, Karl Pfleger, et al. “Irrelevant Features and the Subset Selection Problem.” In: *ICML*. Vol. 94. 1994, pp. 121–129.

[Kal+07] Alexandros Kalousis, Julien Prados, and Melanie Hilario. “Stability of feature selection algorithms: a study on high-dimensional spaces”. In: *Knowledge and information systems* 12.1 (2007), pp. 95–116.

[KP] Anders Kofod-Petersen. *How to do a Structured Literature Review in computer science, http://research.idi.ntnu.no/aimasters/files/SLR_HowTo.pdf.*

[Kru+12] Joseph S. Krupa, Somdeb Chatterjee, Ethan Eldridge, Donna M. Rizzo, and Margaret J. Eppstein. “Evolutionary feature selection for classification: A plug-in hybrid vehicle adoption application”. In: 2012, pp. 1111 –1118.

[Liu+08] Nan Liu and Han Wang. “Improving predictive accuracy by evolving feature selection for face recognition”. In: *IEICE Electronics Express* 5.24 (2008), pp. 1061 –1066.

[MG+07] Ivan Mejia-Guevara and Angel Kuri-Morales. “Evolutionary feature and parameter selection in support vector regression”. In: vol. 4827 LNAI. 2007, pp. 399 –408.

[Mun+06] D.P. Muni, N.R. Pal, and J. Das. “Genetic programming for simultaneous feature selection and classifier design”. In: *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)* 36.1 (2006/02/), pp. 106 –17.

[Nes+09] Kourosh Neshatian and Mengjie Zhang. “Dimensionality reduction in face detection: A genetic programming approach”. In: 2009, pp. 391 –396.

[Pra+10] Y. Prasad, K.K. Biswas, and C.K. Jain. “SVM Classifier Based Feature Selection Using GA, ACO and PSO for siRNA Design”. In: vol. pt.2. 2010//, pp. 307 –14.

[RC+06] Jose-Federico Ramirez-Cruz, Olac Fuentes, Vicente Alarcon-Aquino, and Luciano Garcia-Banuelos. “Instance selection and feature weighting using evolutionary algorithms”. In: 2006, pp. 73 –79.

[Roy+08] Kaushik Roy and Prabir Bhattacharya. “Improving features subset selection using genetic algorithms for iris recognition”. In: vol. 5064 LNAI. 2008, pp. 292 –304.

[Sik+05] R. Sikora and S. Piramuthu. “Efficient genetic algorithm based data mining using feature selection with Hausdorff distance”. In: *Information Technology & Management* 6.4 (2005/10/), pp. 315 –31.

[Tre+04] A. Treptow and A. Zell. “Combining Adaboost learning and evolutionary search to select features for real-time object detection”. In: vol. Vol.2. 2004//, pp. 2107 –13.

[Vig+12] L. Vignolo, D. Milone, C. Behaine, and J. Scharcanski. “An evolutionary wrapper for feature selection in face recognition applications”. In: 2012//, pp. 1286 –90.

[Viv+03] R.A. Vivanco and N.J. Pizzi. “Identifying effective software metrics using genetic algorithms”. In: vol. vol.2. 2003//, pp. 1305 –8.

[Wan+12] Shangfei Wang, Shan He, and Hua Zhu. “Similarity Measurement and Feature Selection Using Genetic Algorithm”. In: vol. vol.2. 2012//, pp. 20 –9.

[Win+11] S.M. Winkler, M. Affenzeller, G. Kronberger, M. Kommenda, S. Wagner, W. Jacak, and H. Stekel. “Analysis of Selected Evolutionary Algorithms in Feature Selection and Parameter Optimization for Data Based Tumor Marker Modeling”. In: vol. pt.1. 2011//, pp. 335 –42.

[Yah+11] A.A. Yahya, A. Osman, A.R. Ramli, and A. Balola. “Feature selection for high dimensional data: an evolutionary filter approach”. In: *Journal of Computer Sciences* 7.5 (2011//), pp. 800 –20.

[Yan+98] Jihoon Yang and Vasant Honavar. “Feature subset selection using a genetic algorithm”. In: *Feature extraction, construction and selection*. Springer, 1998, pp. 117–136.

[Zen+09] Xiao-Ping Zeng, Yong-Ming Li, and Jian Qin. “A dynamic chain-like agent genetic algorithm for global numerical optimization and feature selection”. In: *Neurocomputing* 72.4-6 (2009/01/), pp. 1214 –28.

[Zha+05] C.K. Zhang and Hong Hu. “An effective feature selection scheme via genetic algorithm using mutual information”. In: 2005//, pp. 73 –80.

[Zhu+10] Hao-Dong Zhu, Hong-Chan Li, Xiang-Hui Zhao, and Yong Zhong. “Feature Selection Method by Applying Parallel Collaborative Evolutionary Genetic Algorithm”. In: *Journal of Electronic Science and Technology* 8.2 (2010/06/), pp. 108 –13.

# Appendix A

# Literature search

This chapter contains a table of all the documents that passed the first search criteria described in 2.4. For those excluded there is a description of why (IC: Inclusion criteria. QC: Quality control). See chapter 2 for more details.

| document name | year | excluded because of: | included as: |
|---|---|---|---|
| Feature Selection Method by Applying Parallel Collaborative Evolutionary Genetic Algorithm | 2010 | - | [Zhu+10] |
| Enhancing the diversity of genetic algorithm for improved feature selection | 2010 | - | [AlS+10] |
| A multi-objective evolutionary algorithm-based ensemble optimizer for feature selection and classification with neural network models | 2013 | IC3, IC4 | - |
| A dynamic chain-like agent genetic algorithm for global numerical optimization and feature selection | 2009 | - | [Zen+09] |
| Fuzzy genetic algorithm approach to feature selection problem | 1997 | - | [Fun+97] |
| Similarity Measurement and Feature Selection Using Genetic Algorithm | 2012 | - | [Wan+12] |
| An effective feature selection scheme via genetic algorithm using mutual information | 2005 | - | [Zha+05] |
| Efficient genetic algorithm based data mining using feature selection with Hausdorff distance | 2005 | - | [Sik+05] |
| Feature selection in SVM based on the hybrid of enhanced genetic algorithm and mutual information | 2006 | QC2, QC3 | - |
| Feature selection using genetic algorithm and it's application to face recognition | 2005 | - | [Har+05] |
| Feature selection using hybrid Taguchi genetic algorithm and Fuzzy Support Vector Machine | 2010 | QC3 | - |
| Enhancing evolutionary instance selection algorithms by means of fuzzy rough set based feature selection | 2012 | IC3 | - |
| Feature subset selection by Bayesian networks: a comparison with genetic and sequential algorithms | 2001 | IC3 | - |
| Knowledge discovery in medical and biological datasets using a hybrid Bayes classifier/evolutionary algorithm | 2003 | IC3 | - |
| Instance selection and feature weighting using evolutionary algorithms | 2006 | - | [RC+06] |
| Genetic programming for simultaneous feature selection and classifier design | 2006 | - | [Mun+06] |
| Feature selection based on immune clonal selection algorithm | 2004 | IC2 | - |
| GEFeS: Genetic & evolutionary feature selection for periocular biometric recognition | 2011 | corrupt | - |
| Evolutionary feature selection and electrode reduction for EEG classification | 2012 | corrupt | - |
| Parametric and nonparametric evolutionary computing with a content-based feature selection approach for parallel categorization | 2009 | QC2, QC3 | - |
| Genetic & Evolutionary Biometrics: Hybrid feature selection and weighting for a multi-modal biometric system | 2012 | QC3 | - |
| Feature Set Reduction by Evolutionary Selection and Construction | 2010 | - | [Dro+10] |
| Evolutionary learning of linear trees with embedded feature selection | 2006 | IC3, IC4 | - |
| Bio-inspired algorithms for optimal feature subset selection | 2012 | IC3, IC4 | - |
| Evolutionary feature selection via structure retention | 2012 | QC3, corrupt | - |
| Hybridization of Evolutionary Mechanisms for Feature Subset Selection in Unsupervised Learning | 2009 | QC2, QC3 | - |
| Feature selection for neural networks through functional links found by evolutionary computation | 1997 | IC4 | - |
| An evolutionary attribute clustering and selection method based on feature similarity | 2010 | IC3, IC4 | - |
| Hybrid neural network and genetic algorithm based machining feature recognition | 2004 | IC3, IC4 | - |
| Analyzing the cross-generalization ability of a hybrid Genetic & Evolutionary application for multibiometric feature Weighting and Selection | 2012 | IC3, IC4 | - |

| document name | year | excluded because of: | included as: |
|---|---|---|---|
| A minimum risk wrapper algorithm for genetically selecting imprecisely observed features, applied to the early diagnosis of dyslexia | 2008 | QC3 | - |
| Understanding the evolutionary process of grammatical evolution neural networks for feature selection in genetic epidemiology | 2006 | IC3, IC4 | - |
| The feature selection method based on the evolutionary approach with a fixation of a search space | 2007 | QC3 | - |
| An evolutionary wrapper for feature selection in face recognition applications | 2012 | - | [Vig+12] |
| Binary particle swarm optimization based algorithm for feature subset selection | 2009 | IC3, IC4 | - |
| Genetic-fuzzy rule mining approach and evaluation of feature selection techniques for anomaly intrusion detection | 2006 | IC3, IC4 | - |
| Feature selection and classification by using grid computing based evolutionary approach for the microarray data | 2010 | not complete? | - |
| Evolutionary-rough feature selection for face recognition | 2010 | IC3, IC4 | - |
| Analysis of Selected Evolutionary Algorithms in Feature Selection and Parameter Optimization for Data Based Tumor Marker Modeling | 2011 | - | [Win+11] |
| Combining evolutionary and sequential search strategies for unsupervised feature selection | 2010 | QC3 | - |
| Evolutionary computing optimization for parameter determination and feature selection of support vector machines | 2009 | - | [Din+09] |
| Genetic & Evolutionary Type II Feature Extraction for Periocular-based Biometric Recognition” | 2010 | IC3, IC4 | - |
| A GA-based feature selection algorithm for remote sensing images | 2008 | - | [DS+08] |
| Evolutionary feature selection in boosting | 2004 | IC3, IC4 | - |
| Features selection approaches for intrusion detection systems based on evolutionary algorithms | 2009 | IC3, IC4 | - |
| Feature selection for a fast speaker detection system with neural networks and genetic algorithms | 2006 | IC3, IC4 | - |
| Co-evolutionary genetic Multilayer Perceptron for feature selection and model design | 2011 | IC2 | - |
| Evolutionary rough feature selection in gene expression data | 2007 | - | [Ban+07] |
| Feature selection for high dimensional data: an evolutionary filter approach | 2011 | - | [Yah+11] |
| Evolutionary feature selections for face detection system | 2008 | IC3, IC4 | - |
| Evolutionary selection of features for neural sleep/wake discrimination | 2009 | - | [Dur+09] |
| An incremental approach to genetic-algorithms-based classification | 2005 | IC3, IC4 | - |
| Evolutionary Feature Construction for Ultrasound Image Processing and its Application to Automatic Liver Disease Diagnosis | 2011 | IC3, IC4 | - |
| Evolutionary multi-objective optimization of trace transform for invariant feature extraction | 2012 | IC3, IC4 | - |
| A New Dimensionality Reduction Algorithm for Hyperspectral Image Using Evolutionary Strategy | 2012 | IC3, IC4 | - |
| Hyperspectral feature selection and classification with a RBF-based novel double parallel feedforward neural network and evolution algorithms | 2009 | IC3, IC4 | - |
| Investigation of evolutionary feature subset selection in multi-temporal datasets for harmful algal bloom detection | 2011 | QC1, QC3 | - |
| Feature Selection with Intelligent Dynamic Swarm and Rough Set | 2010 | - | [Bae+10] |
| Hybrid optimization of feature selection and SVM training model | 2004 | IC2 | - |
| Evolutionary feature and parameter selection in support vector regression | 2007 | - | [MG+07] |
| A filter-based evolutionary approach for selecting features in high-dimensional micro-array data | 2010 | - | [Can+10] |
| Feature subset selection based on co-evolution for pedestrian detection | 2011 | IC2 | - |
| Combining Adaboost learning and evolutionary search to select features for real-time object detection | 2004 | - | [Tre+04] |

| document name | year | excluded because of: | included as: |
|---|---|---|---|
| Improving predictive accuracy by evolving feature selection for face recognition | 2008 | - | [Liu+08] |
| Unsupervised learning of mutagenesis molecules structure based on an evolutionary-based features selection in DARA | 2012 | - | [Alf+12] |
| Memetic Feature Selection: benchmarking Hybridization Schemata | 2010 | QC3 | - |
| Feature extraction using evolutionary weighted principal component analysis | 2005 | IC3, IC4 | - |
| Improving features subset selection using genetic algorithms for iris recognition | 2008 | - | [Roy+08] |
| Knowledge discovery in gene expression data via evolutionary algorithms | 2011 | IC3, IC4 | - |
| Feature selection based on rough sets and particle swarm optimization | 2007 | IC3, IC4 | - |
| An evolutionary approach for protein classification using feature extraction by artificial neural network | 2010 | IC3, IC4 | - |
| Wrapper based feature selection in hyperspectral image data using self-adaptive differential evolution | 2011 | - | [Dat+11] |
| Automated Selection of Damage Detection Features by Genetic Programming | 2013 | IC3, IC4 | - |
| Evolutionary search of thresholds for robust feature set selection: application to the analysis of microarray data | 2004 | IC4 | - |
| ESVM: Evolutionary support vector machine for automatic feature selection and classification of microarray data | 2007 | - | [Hua+07] |
| Evolutionary feature selection applied to artificial neural networks for wood-veneer classification | 2008 | IC2 | - |
| SVM Classifier Based Feature Selection Using GA, ACO and PSO for siRNA Design | 2010 | - | [Pra+10] |
| Evolutionary feature selection for classification: A plug-in hybrid vehicle adoption application | 2012 | - | [Kru+12] |
| Optimization of an image set by genetic feature selection for real-time photomosaics | 2010 | QC1, QC3 | - |
| Feature selection and classification in noisy epistatic problems using a hybrid evolutionary approach | 2007 | QC1, QC2, QC3 | - |
| Identifying effective software metrics using genetic algorithms | 2003 | - | [Viv+03] |
| Supervised genetic search for parameter selection in painterly rendering | 2006 | IC3, IC4 | - |
| Improving statistical measures of feature subsets by conventional and evolutionary approaches | 2000 | QC3 | - |
| Dimensionality reduction in face detection: A genetic programming approach | 2009 | - | [Nes+09] |
| On objective feature selection for affective sounds discrimination | 2012 | IC2 | - |
| Selection of relevant features for EEG signal classification of schizophrenic patients | 2007 | IC3, IC4 | - |
| Enhanced prediction of misalignment conditions from spectral data using feature selection and filtering | 2007 | QC3 | - |
| A new combined filter-wrapper framework for gene subset selection with specialized genetic operators | 2010 | - | [BH+10] |
| Genetic programming for kernel-based learning with co-evolving subsets selection | 2006 | IC3, IC4 | - |
| Using feature selection approaches to find the dependent features | 2010 | QC3 | - |
| Feature subset selection by particle swarm optimization with fuzzy fitness function | 2008 | QC3 | - |
| Evolutionary search of optimal features | 2006 | IC3, IC4 | - |
| A new sampling technique and SVM classification for feature selection in high-dimensional imbalanced dataset | 2011 | IC3, IC4 | - |
| Selection and fusion of facial features for face recognition | 2009 | IC3, IC4 | - |
| Genetic programming for mining DNA chip data from cancer patients | 2004 | QC3 | - |
| Data mining based on gene expression programming and clonal selection | 2006 | IC3, IC4 | - |