NTNU

Norwegian University of
Science and Technology

A Machine Learning Approach for Determining Reference Wells in the Norwegian Continental Shelf

Minh Hoan Bui Pham

Master of Science in Industrial Cybernetics
Submission date: July 2018
Supervisor: Ole Morten Aamo, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

Before drilling a new well, engineers carefully select relevant reference wells that serve as a source of information for efficiently planning the new well. The selection process entails browsing through a large database and is at present a time-consuming, highly manual task. The objective of this master thesis work is to develop an expert system that automatically extracts the relevant reference wells from the database, given the new well's geological profile. The candidate will look into methods for unsupervised machine learning and clustering analysis to achieve this. In particular, the following tasks will be addressed

- Summarize relevant methods from the machine learning literature.
- Extract well data from the database available at the Norwegian Petroleum Directorate (NPD), as well as from the collaborative firm. .
- Select data fields that are relevant for the problem and that will be used in the clustering analysis. Motivate the choice.
- Specify the algorithm to use, and define in detail its input data (can be all of the data selected in point 3, or a justified subset of it).
- Implement the system using appropriate tools.
- Write report.

Supervisor: Professor Ole Morten Aamo

Preface

This thesis will be submitted to the Department of Engineering Cybernetics at NTNU, in collaboration with an oil firm in the industry, to fulfill the requirements of a Master of Science in Industrial Cybernetics. What motivates me, is the thought of encouraging the development of in-house solutions for small scale problems that can increase the efficiency in an oil firm. The case that is presented in this thesis came to attention during my summer internship at the collaborative firm. While working, I got caught up in how engineers still has a lot of manual work when it comes to accessing information. I found this interesting and presented it as one of four cases in a feasibility study that was submitted fall of 2017. Luckily, both me and the collaborative firm wished to pursue the case presented in this thesis.

The thesis purposes a machine learning solution to the problem of selecting good and relevant reference wells for when a new well is to be drilled. An expert system that automatically extracts suitable wells is scripted with tools that are regularly available for the petroleum engineer. To be applicable to a wide audience, an official database at the Norwegian Petroleum Directorate has been used as a main source for well data, while confidential information extracted from Final Drilling programs of the collaborative firm are made anonymous. Python is used as the main scripting language, together with Scikit-learn, a free software library for machine learning, and PyCharm as the IDE, both compatible with Python.

This semester has been quite interesting and challenging, but at the same time rewarding, as I have been able to work with Python over a longer period. I have learned how important it is to gain knowledge and experience to better understand the task one is solving. This, combined with familiarization of the selected scripting language will help creating good scripts. I therefore look forward to gain more knowledge and experience as an engineer to better solve such tasks.

I would like to thank the collaborative firm and my team there for taking on the thesis on such short notice, specially Kjell Eivind Stormo and Terje Myklebust, your feedback has been valuable. Big thanks to my supervisor Professor Ole Morten Aamo and my fellow peers at the NTNU office for being vital discussion partners throughout the whole process.

July 2018, Trondheim

Minh Hoan Bui Pham

Abstract

An unsupervised learning solution for selecting relevant reference wells for a new well that is to be drilled in the Norwegian Continental Shelf (NCS), shall be made with readily available software tools such as Python and the machine learning library Scikit-Learn. The developed expert system shall select the most relevant wells based on a simple feature of the well; its geological profile. Clustering analysis, such as the k-means algorithm and Voronoi partition, will serve to increase the convergence towards the solution, proposed by the system, by pinpointing which clusters of wells are most relevant in the total database of wells. The final solution by the system is then achieved by studying these selected clusters with simple statistics and by exploiting the greedy property of the nearest neighbor rule.

The foundation of the expert system is based on a small test set consisting of 16 wells from two different fields. Eight from the Ivar Aasen field and eight from the Valhall field. Simple and clear assumptions are made on this test set in order to program a scalable prototype of the expert system. The prototype is tested on a large available database of well data taken from the Norwegian Petroleum Directorate (NPD), together with a well, *Well X*, from a collaborative firm where the selected reference wells are already known. The results gained from both cases were good. When running the test set, the system was able to distinguish between the Ivar Aasen field and the Valhall field. When running *Well X* through the system, it was able to select the eight original wells that were actually picked in the real operation.

Though further work is necessary to achieve an optimal system, the proposed final solution in this thesis leaves a good first impression, and should motivate the average engineer in the industry to develop small scale solutions to optimize daily work processes.

Sammendrag

En ikke-ledet læring løsning for å velge relevante referansebrønner for en ny brønn som skal bores på norsk kontinentalsokkel (NCS), skal utvikles med Python og maskinlæringsbiblioteket Scikit-Learn, to lett tilgjengelige programvarer.
Det utviklede ekspertsystemet velger ut de mest relevante referansebrønnene basert på én enkel egenskap av brønnen; dens geologiske profil. Klyngeanalyse, som k-means og Voronoi-partisjon, vil bidra til rask konvergering mot løsningen som ekspertsystemet foreslår, ved å påpeke hvilke klynger som er mest hensiktsmessig å analysere videre på av den totale databasen av brønner. Den endelige løsningen som foreslås av systemet oppnås ved å studere disse klyngene med enkle statistiske metoder, og ved å utnytte den grådige egenskapen til nærmeste nabo-regelen.

Grunnlaget for ekspertsystemet er basert på et lite testsett bestående av 16 brønner fra to forskjellige felt. Åtte brønner fra Ivar Aasen-feltet og åtte fra Valhall-feltet. Enkle og klare antagelser er testet på dette testsettet i den hensikt å kunne programmere en skalerbar prototype av ekspertsystemet. Det endelige ekspertsystemet testes på en stor tilgjengelig database som består av brønndata hentet fra Oljedirektoratets nettsider og en brønn, Brønn X, hentet fra samarbeidsselskapet hvor de valgte referansebrønnene for Brønn X allerede er kjent. Begge tilfellene ga gode resultater. Når ekspertsystemet kjørte testsettet klarte systemet å skille mellom Ivar Aasen-feltet og Valhall-feltet. Når Brønn X ble kjørt, var systemet i stand til å velge de åtte brønnene som faktisk ble plukket i den virkelige operasjonen.

Selv om ytterligere arbeid er nødvendig for å oppnå et optimalt system, gir den foreslåtte løsningen i denne oppgaven et godt førsteinntrykk. Dette bør motivere ingeniøren i bransjen til å utvikle små og effektive løsninger som kan bidra til å optimalisere hverdagslige arbeidsprosesser.

Contents

List of Figures

List of Abbreviations

AI Artificial Intelligence

FM Formation

GP Group

HCA Hierarchical Cluster Analysis

IDE Integrated Development Environment

MDS Multidimensional Scaling

ML Machine Learning

NCS Norwegian Continental Shelf

NPDID Norwegian Petroleum Directorate Identity Document

NPD Norwegian Petroleum Directorate

PCA Principal Component Analysis

TVD True vertical depth

Chapter 1

Introduction

1.1 Motivation

Machine learning (ML) methods have been present in the petroleum industry for a long time but only in a different form. As ML bases heavily on the field statistics and pattern recognition, different technique involving forecasting, modeling, optimization, or estimation are fundamental for machine learning solutions. Recently, the industry of petroleum here on the Norwegian Continental Shelf (NCS) have shown a significantly increased interest in the field of machine learning. The larger oil firms are stating that digitization is correct path to go, painting a bigger picture that seeks to incorporate ML, artificial intelligence (AI), and automation to develop unmanned facilities and automated drilling solutions, but also, that small and efficient solutions must make their way into the daily work processes of an oil firm to ensure optimal results [5] [4].

An example of such a process is the selection of relevant wells when a new well is to be drilled by a firm. When planning to drill a new well, a hand full of reference wells must be selected to eliminate as much uncertainty as possible. The reference wells serves as a source of information for efficient planning of the different phases throughout the operation of the new well, this includes everything from casing and completion design to choosing the correct mud weight for the whole drilling process. A drilling engineer can use anything from a few hours to a day, or a couple of days at worst, to select good reference wells for a new well that is to be drilled. The selection process entails browsing through a large database and is at present a time-consuming, complex, and highly manual task. Instead of having an engineer starting arbitrary somewhere in his or hers search for reference wells, if we can implement the best practice of selecting reference wells into a system and pinpoint where the engineer should start, then surely, we should have a software that could serve as a strong helping tool for an engineer.

The main motivation for this thesis is to encourage in-house development of small scale problems with tools that are readily available for any engineer in the industry. If in-house solutions can be made free, reliable, and usable, there is

no telling how much value that would add, both economically and in increased expertise in employees, in the long run. The thesis will use Python in order to implement a machine learning solution for determining reference wells in the NCS.

1.2 Previous Work

Well planning in general involves an abundant of softwares that serves to aid an engineer in finding good referential information. However, a bulk of these softwares are mostly used further down the process chain while the goal of determining which wells one should start extracting information from, would be one of the primary tasks in well planning.

The AGR software, iQx, is an award-winning cloud-based data platform that can be used for well planning [11]. The software combines well data from over 80,000 wells from several available databases worldwide and enables the user to search for information and knowledge regarding any well that the software can reach. iQx is convenient as it has done a great amount of work for us; it has gathered all the scattered well data into a single database. But how good is it if it does not include the well we wish to drill?

iQx features four different search types; *adjacent wells*, *within radius*, *well name*, and *field*. What is usually known in the early phase of planning a well is where in the world we have to drill. By knowing so, one can plug in the GPS coordinates and search for adjacent wells, or wells within a certain radius, from that location and work from there. This is often how iQx is used today to find potential reference wells [9].

1.3 Problem Formulation

Does this mean that a close lying well is always a relevant? What if the new well that is to be drilled has complicated layers, where lessons learned from wells drilled in Brasil could serve as a relevant source to drill these layers? This thesis will look into the possibility of developing an expert system that can select relevant reference wells from a database of wells, based solely on the geological profile of a new well that is to be drilled. The solution proposed by the expert system is a listing of the most relevant wells for the new well. The final expert system could serve as a module for softwares such as the iQx, meaning that if the system were to be implemented into iQx we would gain a fifth search type: searching by geological profile. We state two hypotheses that the expert system should put to test:

Hypothesis 1: *The expert system should be able to identify that different fields exists on the NCS.*

Hypothesis 2: *The expert system should be able to select relevant reference wells.*

1.3.1 Approach and limitations

The thesis will look into methods for unsupervised machine learning and clustering analysis to achieve its purpose. In particular, the following tasks will be addressed

- Summarize relevant methods from the machine learning literature.
- Extract well data from the database available at the Norwegian Petroleum Directorate (NPD), as well as from the collaborative firm.
- Select data fields that are relevant for the problem and that will be used in the clustering analysis. Motivate the choice.
- Specify the algorithm to use, and define in detail its input data (can be all of the data selected in point 3, or a justified subset of it).
- Implement the system using appropriate tools.
- Write report.

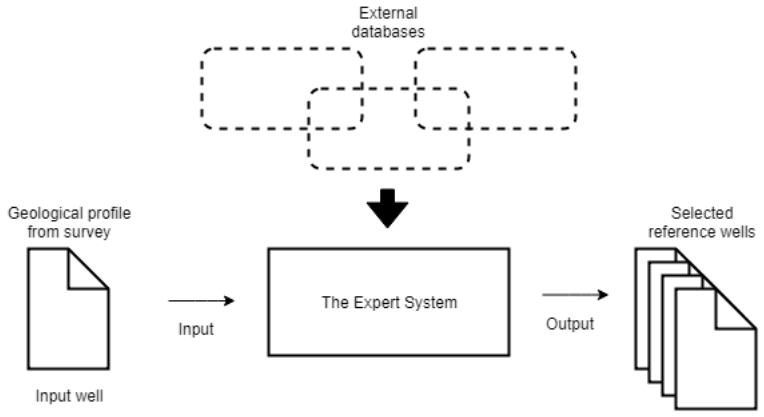

Figure 1.1: Rough sketch of the expert system.

To save a lot of manual work, an official database at the Norwegian Petroleum Directorate has been chosen as main source for well data over the database that is available at the collaborative firm. A large portion of the thesis's work is done in Python, therefore, manually extracting well data from rapports available at the collaborative firm, would have not resulted in the expert system that is presented.

The expert system is mainly developed with a smaller set of wells, referred to as the test set. The goal here, is to achieve a prototype of the system that is scalable, leaving us only to retrieve the external database at NPD and implement it into our prototype. Developing the expert system first with a smaller test set will make error detection easier and also computational time shorter [1]. This means, that the assumptions and methods that were chosen for the test set will lay the foundation for the whole expert system. Also, to begin with there is a naive assumption in our approach; we select relevant reference wells based solely on a well's geological profile. This is not always the case in actual selection. Other factors such as the availability of data or important incidents and lessons learned from other wells would also play a role in the selection of a well. Though, it is important to remember that the solution of our expert system is not carved in stone, rather, it should serve as guidance or indication on where the engineer can start looking for information.

To ensure that the prototype of the system is relevant for an engineer in his or hers daily work, the approach will include user feedback such that best practice is implemented. In any product development, involving the user of the product will increase its utility. This two-way interaction is though subject to the availability in having such meetings between the candidate and the collaborative firm.

1.4 Available Software

The thesis will be using free and available softwares that are readily available for any engineer.

1.4.1 Python

Python is a free, open-source, high-level programming language [2] that is used in thousands of real-world business applications around the world, including anything from large and critical systems to small scale problem solvers. Its syntax focuses on readability and is on of many features that increases its popularity. Google and YouTube are for instance users of Python [25].

1.4.2 Scikit-Learn

By being open-sourced, Python supports the use of modules and packages developed by other users. Scitkit-learn is also open-sourced, and it has a strong and widely used libraries of machine learning modules that are compatible with Python. Scikit-learn will in this thesis be the main source of imported ML methods [27].

[1] In this thesis we refer to computational time to be the actual time we need to wait for an algorithm or function to complete. In data science, the mathematical analysis of computational time of algorithms are important but it is beyond the scope of this thesis.

[2] High-level programming language is more intuitive than low-level language.

1.4.3 PyCharm

Instead of programming directly in the *cmd* window of an operating system, or in any other text editor [3], an *integrated development environment (IDE)* will create an environment of the language one is using in order to increase efficiency in writing scripts. By enabling features such as marking errors, debugging, compilation, and easy access to modules and packages, PyCharm makes it easier to write good scripts efficiently [22] [4].

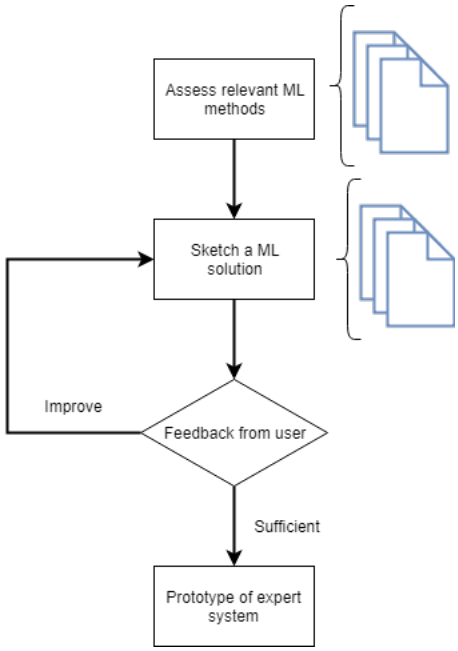

Figure 1.2: Approach in Python with user involvement.

[3] Notepad is a text editor.
[4] A PyCharm license has to be purchased to unlock the full version.

1.5 Notation

Some notation is seemed necessary to define before continuing. Due to scripting in Python, we will be using that a vector is a row vector by default. A vector, or a matrix, can have the dimensions of m rows $\times$ n columns which defines the configuration in $\mathbb{R}^n$ vector space. Vectors and matrices are written as **bold** lower and upper case letters, respectively. A scalar can either be a lower or upper case letter.

In machine learning, and often pattern recognition as well, we often use the terms *samples* and *features*. While in object oriented programming we often refer samples as *objects* and features as *attributes*. A sample could have different features describing it; like height, eye color, and hair color can be features (attributes) that describes a human being (the object or sample). Features can be put into a feature vector $\boldsymbol{x}$ of dimensions $1 \times n$ where n is the number of features describing the object. We will use the term attribute when referring explicitly to some code, while the term features is used when discussing theory. Objects and samples are used interchangeably. Definitions are stated by using a colon and an equality sign; a:=b, which means that a, by definition, equals b.

Example 1.5.1

$$\mathbf{X} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x}_i\}^T \tag{1.1}$$

where $\mathbf{X} :=$ *the set of* i *objects, while* $\boldsymbol{x_i} :=$ *the set of features describing the object, called the feature vector.*

When presenting the approach used to develop the expert system in Chapter 3, we will often refer to the source code given in Appendix A. Though the thesis is written independent of Appendix A, as excerpts of the original source code will be introduced for the most essential sections, some sections will be referenced to the appendix in case they are of interest to study. It should be mentioned that excerpts in this form will have different line numbering than the source code given in Appendix A:

Excerpt 1

```
1 Something from the source code
2 is written to better understand
3 the selected approach
```

Anything that is directly related to the source code (i.e commands, variables, functions etc.) will appear in the thesis as a covered gray word or sentence with the line reference next to it. The latter is optional and based on importancy. Scripts, which are files, will have have the same appearance only with **bold** letters.

Example 1.5.2 *The variable* `example` *in* **`example.py`** *, line 1. Is defined as the ndarray example variable.*

1.6 Structure of Rapport

We have just covered the motivation for approaching the problem presented in this thesis, as well as the necessary tools and notation needed to proceed.

Chapter 2 will cover the necessary background material to understand the different methods that are implemented with Python. We will be focusing on what the algorithms solves and how they are trained, and not on the statistical analysis that considers distributions and probabilities. The background material will define and introduce the different directions within machine learning, further topics includes the nearest neighbor rule, clustering analysis, and a dimensionality reduction technique called *multidimensional scaling* (MDS).

The approach used to develop the expert system is explained in detail in Chapter 3. We will work stepwise through the different scripts that make up the system; from initialization of Python to the final function that produces the output of the system. We will also assess how our input data is processed throughout the whole system, and justify why certain choices and approach has been used.

Chapter 4 presents the results obtained when running the expert system. First, we will look at what the test set produces. The test set, has played an important part of the thesis as a large portion of the final expert system is based on the results gained from this set. The final part of the chapter, will include a whole database of 1662 wells drilled in the NCS. We will also be using a well drilled by the collaborative firm to see if the expert system can retrieve relevant wells that actually were selected in the real operation.

Towards the end, in Chapter 5, we will be discussing our approach of solving our problem, i.e. if certain parts and results were as we assumed.

Finally, the thesis comes to a conclusion regarding the final expert system and future work in Chapter 6.

Chapter 2

Background Material

2.1 Introduction

This chapter elaborates on the background material that is fundamental to understanding the approach of solving our problem with emphasis on different machine learning methods that are applied. The main algorithms used in this thesis comes from *unsupervised learning* and *manifold learning* methods.

2.2 What is Machine Learning?

Machine learning has its origin from the field of statistical analysis and pattern recognition. Although it is often used interchangeably with the field of AI there exists a clear line where we differentiate between the two of them. Machine learning focuses on the theory that computers can learn without being explicitly programmed to perform and complete specific tasks [12]. AI on the other hand, is the broader concept of machines being able to fulfill tasks in a way we humans consider as *smart* [38]. In machine learning, we give the computer access to data and construct algorithms in such way that it gives the computer the ability to independently adapt when exposed to new data, i.e. the ability to *learn* from these data.

Machine learning can be divided into four main categories:

- Supervised learning,
- unsupervised learning,
- reinforcement learning,
- and semisupervised learning

which all differs in how the they work with the data that is given.

2.2.1 Supervised learning

Supervised learning bases its learning from data where we already know the outcome of. The data is provided by some knowledgeable external supervisor (us humans) where the relationship between the input and the output is known. When this is true, we say that the output is *labelled*. The main goal in supervised learning models is to accurately predict an output measurement for a given input that is *unlabeled*, i.e. the relationship between input and output is unknown, leaving us only to predict what the output *could* be. An example would be a house (the input) of a certain *size*, *location*, *number of rooms* (the features) that is sold on the market for a certain prize (the output). Accurate prediction is achievable by building a prediction model, also called a *learner*. The learner is trained by using a *training set* of data where both inputs and outputs are known, typically with a rule of thumb of 80% of the complete data set. The remaining 20% is used to test the accuracy of the learner. A good learner is one that can predict the correct output for an unlabeled sample with great accuracy [33]. Supervised learning is often divided into two main prediction methods; *classificiation* and *regression*. Classification is the task of qualitative labeling of unlabeled input data, like determining if a house is a villa, studio, or three-room apartment. While regression focuses quantitative labeling, i.e. predicting a numerical value for the input like prize, in the house example given above.

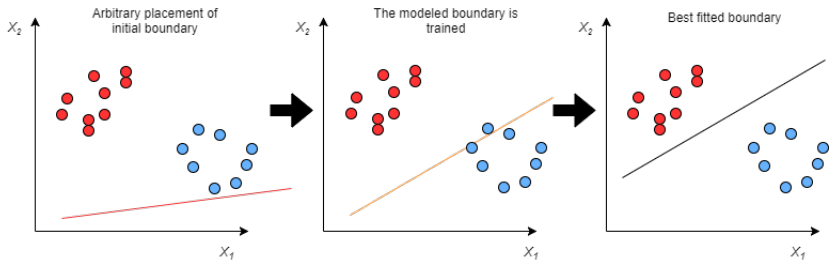

Figure 2.1: A learner is trained on a set of labeled objects. The final model can be used for classification.

2.2.2 Unsupervised learning

With unsupervised learning, we approach a problem with no information about what the output should look like for a given input. We say that our input is given to us unlabeled. Because of this, there is no way of evaluating the accuracy of the produced output like in supervised learning. One is simply left to address a large set of unknown input data and try to identify patterns and structures in it. An important method within unsupervised learning is *clustering* analysis. In clustering analysis we try to identify groups (clusters) consisting of data samples with similar features, while samples in different clusters have features that are dissimilar [2]. Unsupervised learning can be a goal in itself by discovering these

patterns and structures, or it can be used as a means towards an end where again a knowledgeable external supervisor can label the discovered structures, this is called *semisupervised learning* [32]. Note how the training set in unsupervised learning is the unlabeled data set itself.

Figure 2.1 can easily relate to an unsupervised learning method. Imagine that all objects have the same color, i.e. they are unlabeled. Regardless of this, clustering will be able to separate them into two clusters that defines a region of which objects belong to that cluster. This will be elaborated on thoroughly in Section 2.6.

2.2.3 Semisupervised learning

Semisupervised learning is working with data that is both labeled and unlabeled; a good mix of unsupervised and supervised methods. The most common scenario though, is having a majority of unlabeled data and some that are labeled. An example from daily use is Google Photos where unsupervised learning (clustering) is initially performed to identify that certain people exists in certain pictures. Then, the only thing remaining is labeling these people with a name (classification), which is the supervised part of the solution [32].

2.2.4 Reinforcement learning

Reinforcement learning is a method that follows a learning-by-doing paradigm. Rather than relying on known knowledge, experience, or specific programmed instructions, the computer, also called an *agent*, must freely explore the data that is in the current environment without any knowledge about it. It uses a reward and punishment system for solving a task, where the main objective is to gather as much reward as possible [31]. With the combination of both exploring new data, and the exploitation of already explored data, reinforcement learning can eventually converge to an optimized solution or decision for a given task. The learning-by-doing paradigm, together with its ability to connect rewards to certain actions, enables reinforcement learning to discover solutions one did not think even existed. Expert systems that surpasses chess or Go players are based on this type of learning [26].

2.3 The Feature Space

In ML literature, we often work with data in the *feature space*. The feature space is an Euclidean space that uses Cartesian coordinates. It is defined as a n-dimensional configuration $\mathbb{R}^n$ that holds a finite number of objects $\mathbf{x}_i$ called the feature vectors. Since the feature vectors defines the configuration $\mathbb{R}^n$, the vectors themselves are of dimensions $1 \times n$. We define the feature space to be a set of n directed axes that are perpendicular to each other and intersect in an origin O. An object $\mathbf{x}_i$ lying in $\mathbb{R}^n$ can be expressed uniquely by the n-tuple of points $(x_{i1}, x_{i2}, ..., x_{in})$, where x_{ia} is the point i's projection onto dimension a. This expression is called Cartesian

coordinates. The Euclidean space with Cartesian coordinates enables us to to do calculations in it [34] [33].

2.3.1 Metric for distance

The selected metric for distance mentioned in this thesis will be the Euclidean distance. The distance between two objects $\mathbf{q}$ and $\mathbf{p}$ of dimension $1 \times n$ is expressed as:

$$d(\mathbf{q}, \mathbf{p}) = ||q - p|| = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{2.1}$$

The Euclidean distance between objects is, in statistics, often called a *dissimilarity* measure [1]. The general term *proximities* is often used to denote both similarity and dissimilarity measurements [33]. In this thesis, we will think of proximities as dissimilarity measures unless it is explicitly stated otherwise.

2.4 The Nearest Neighbor rule

A simple, yet quite important decision rule that is widely used in machine learning is the *nearest neighbor (NN) rule*. The NN rule has the greedy property [2] of selecting whichever object that is closest to the object of interest, and is fundamental to algorithms such as the k-Nearest-Neighbor (k-NN) in supervised learning, or the k-means clustering algorithm in unsupervised learning. We will now take a look at how the k-NN algorithm classifies an object to better understand the nearest neighbor rule.

Algorithm 1 Pseudocode for the k-NN algorithm

1: Map the training set $\mathbf{X}$ of i objects $\mathbf{x}_i$, with $\mathbf{y}$ class labels, and the unknown object $\mathbf{x}$ to the feature space.
2: Select the number of neighbors, k.
3: Compute distances from $\mathbf{x}$ to all objects in training set $\mathbf{X}$, $d(\mathbf{x_i}, \mathbf{x})$.
4: Find the k shortest distances.
5: Assign $\mathbf{x}$ to the class that is of majority of the k nearest neighbors.
6: **end**

[1] Dissimilarity measures are measurements that have small values for similar objects, greater values for dissimilar objects. Similarity measures, is the opposite; they have large values for similar objects and zero or negative values for dissimilar objects

[2] Greedy algorithms are often associated with computer sicence and for solving optimization problems. They have the property of always selecting the choice that looks best at the moment [37].

We begin by retrieving a labeled training set $\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x}_i\}^T$ of i objects as input. The possible labeling is that objects are either of class A or of class B, meaning that $\mathbf{y} = \{A, B\}$. For simplicity, let us define the feature vector $\mathbf{x}_i$ to be of dimensions 1×2, i.e. there are two features describing the object. This means that the feature space is a configuration in $\mathbb{R}^2$. We select the number of neighbors to search for to be $k = 1$.

The learning part of the k-NN algorithm is not complex. It is simply storing the training set that consists of labeled feature vectors in the $\mathbb{R}^2$ configured feature space, as illustrated in Figure 2.2. The trained learner $\hat{Y}$ for the k-NN algorithm can be expressed as:

$$\hat{Y}(\mathbf{x}) = \frac{1}{k} \sum_{\mathbf{x}_i \in N_k(\mathbf{x})} y_i, \tag{2.2}$$

$$\tag{2.3}$$

where $N_k(\mathbf{x})$ is defined as the neighborhood of $\mathbf{x}$ that have the k closest objects $\mathbf{x}_i$ that exists in the training set. $\hat{Y}(\mathbf{x})$ produces an estimate that assigns $\mathbf{x}$ to either class A or class B. The estimate is based on the majority vote of classes, y_i[3], of the k closest objects $\mathbf{x}_i$, i.e. the k nearest neighbors.

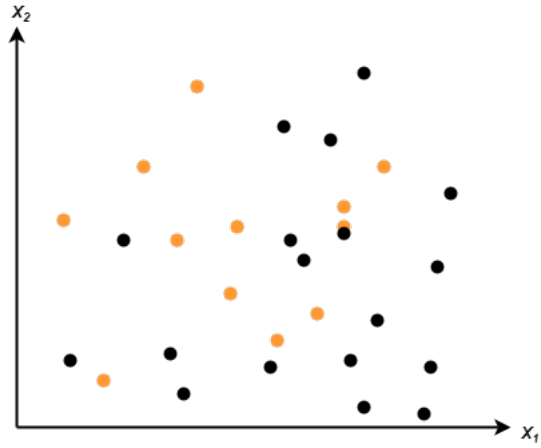

Figure 2.2: Mapping of the training set to feature space. Orange objects are labeled as class A. Black objects are labeled as class B.

2.5 Voronoi Partition

The $\hat{Y}(\mathbf{x})$ learner solves the k-NN algorithm by separating the feature space up in *regions of classification*. Since $k = 1$ the estimate is reduced to simply assigning

[3]Here, $i = 1, 2$ as it is either class A or B.

$\mathbf{x}$ to the class y_i of the closest object $\mathbf{x}_i$ in the training set. This simplifies the computation of the classification regions and correspond to a method called *Voronoi partition* [33]. In a Voronoi partition of the training set, each object $\mathbf{x}_i$ has its own *Voronoi cell* surrounding it, defining an area for which an unlabeled object $\mathbf{x}$ is classified as the class of that object $\mathbf{x}_i$, if it places within this cell, see Figure 2.3.

The Voronoi partition makes classification of an unlabeled object simple. For $k = 1$, the boundaries of the Voronoi cells, in feature space, must lie in the half-way between any two objects of the training set since it only requires one neighbor to classify $\mathbf{x}$, and that neighbor must be closest one. Meaning that for any unlabeled object $\mathbf{x}$ that lands on the boundary of two different Voronoi cells, we can immediately say that it has two equally nearest neighbors, and the classification of $\mathbf{x}$ becomes arbitrary. The same goes for, an unlabeled object that lands on a vertex combining k Voronoi cells; it has k equally nearest neighbors, and the classification is again arbitrary. So for solving the k-NN for increasing values of k, the partitioning of the feature space is done by increasing a training object's Voronoi cell till it includes k neighbors. This is done for all objects x_i in the training set. The final result is a Voronoi partition that defines the regions of classification [4] [16].

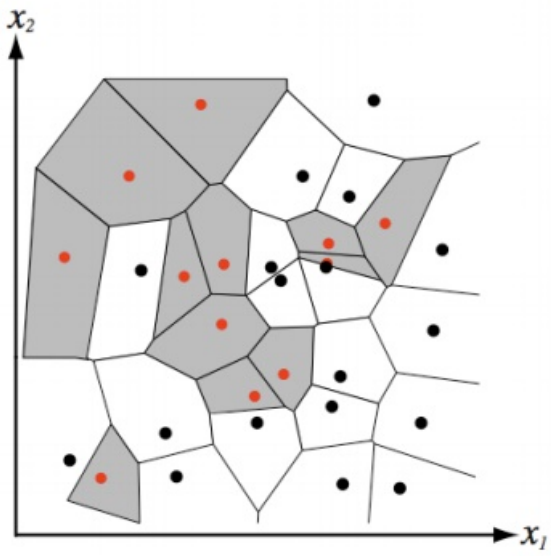

Figure 2.3: Voronoi partition of a two class problem with $k = 1$. Unlabeled objects that falls within the grey region is classified as class A. Class B in white regions. Note how the boundaries lie in the half-way between any points that share a Voronoi cell boundary.

We will later in the thesis see how we do not explicitly use the k-NN algorithm as an classification or regression method but rather, we exploit the nearest neighbor rule to select the most relevant reference wells based on Euclidean distance.

[4] [10] is a good graphical example of Voronoi partitioning.

2.6 Unsupervised Learning methods

Two important methods within unsupervised learning are clustering analysis such as the k-Mean and Hierarchical Cluster Analysis (HCA) algorithms, and visualization and dimensionality reduction methods such as Principal Components Analysis (PCA) and Multidimensional Scaling (MDS). The goal of unsupervised learning is to gather any information about the unlabeled input data that can help in understanding the data that is analyzed. This is done by discovering patterns and structures that lie in the given data [32]. The k-Means and MDS will be two important methods that is used in this thesis.

2.6.1 Clustering

Cluster analysis focuses on data segmentation and discovering patterns. Given a set of unlabeled data $\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x}_i\}$ as input. We wish to *cluster* the data set into groups such that:

- objects within the same cluster are similar to each other
- objects from different clusters are dissimilar to each other [2]

The k-Means clustering algorithm is one of the most popular clustering algorithms due to its simplicity and fast computational time. The algorithm takes the data set (also, training set) $\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x}_i\}$ as input in addition to the parameter k that decides how many clusters we should group the data into. We recall, that the feature vector $\mathbf{x}_i$ can have n features describing the object but again, let us keep using $n = 2$ as it is easy to understand how the algorithm works by visualizing how an object $\boldsymbol{x} = [x_1, x_2]$ is placed in $\mathbb{R}^2$.

Algorithm 2 Pseudocode for the k-Means algorithm

1: Map the training set $\mathbf{X}$ into feature space
2: Randomly initialize k cluster centroids $\boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_k$
3: **Cluster assignment: for** each object $\mathbf{x}_i$, find the nearest cluster centroid $\boldsymbol{\mu}_k$ and assign the index of this cluster to $c^{(i)}$
4: **Move centroid: for** each cluster, update the cluster centroid position $\boldsymbol{\mu}_k$ to be the mean of all objects that were assigned to cluster k in the previous step
5: Repeat line 3-4 until convergence
6: **end**

k-Means in feature space

The initialization of the k-Means algorithm is similar to the k-NN; the training set $\mathbf{X}$ is mapped into the feature space. In order to produce an output that is clustered into groups, the algorithm initializes k cluster *centroids* $\boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_k$ arbitrary in the feature space. The cluster centroids have same dimensions as the feature vectors. The first step of the k-Means algorithm starts by assigning every object $\mathbf{x}_i$ to a cluster k by selecting the closest cluster centroid $\boldsymbol{\mu}_k$, this can be expressed by

$$c^{(i)} = \arg\min_k ||\mathbf{x}_i - \boldsymbol{\mu}_k|| \tag{2.4}$$

where $c^{(i)} :=$ index of cluster centroid k that is closest to $\boldsymbol{x}_i$. This is called *the cluster assignment step*. When all i objects are assigned to a cluster, the algorithm goes forth by updating the position of the k cluster centroids $\boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_k$. This is done by calculating the mean of all observations of n objects lying in cluster k:

$$\boldsymbol{\mu}_k = \frac{1}{n} \sum_{x_i \in c_k}^{n} \mathbf{x}_i, \tag{2.5}$$

where n is the number of objects that were assigned to cluster k in the previous step. Equation (2.5) is often called the *moving centroid step*. The cluster assignment and moving centroid step is run till convergence, that is, until the k cluster centroids does no longer move [2]. This is recognized as a minimization of the *Within-Cluster Sum of Squares* (WCSS), which can be expressed as

$$W(c) = \sum_{j=1}^{k} n_j \sum_{c^{(i)}=k} ||\mathbf{x}_i - \boldsymbol{\mu}_k|| \tag{2.6}$$

It is the sum of squared distances to the closest centroid for all observations in the training set. Minimizing the WCSS ensures that by convergence, within each cluster k, the average distance from all observations n to the cluster centroid $\boldsymbol{\mu}_k$, is minimized [33] [29].

By now, it should be noticeable that the k-Means uses the nearest neighbor rule in the assignment step; greedily, assigns an object to the nearest cluster centroid. Also, when the $\boldsymbol{\mu}_k$ positions of all cluster centroids have converged we have a Voronoi partitioning of the feature space, where each cluster is its own Voronoi cells where the cluster boundaries lies in the half-way between any cluster centroids that shares a Voronoi cell boundary. The k-means imported from Scikit-learn is solved with Lloyd's algorithm, which seeks a Voronoi Partition of the feature space.

2.7 Multidimensional Scaling

The main source for this section is from [33]. Multidimensional Scaling (MDS) is a dimensionality reduction method within *manifold learning* [5] that attempts to preserve the proximities between objects as best possible. For a large data set with many features, reducing dimensionality will not only a save computational time but if also the dimensions can be reduced to an intuitive space like $\mathbb{R}^2$ or $\mathbb{R}^3$, it can serve as a strong visualization technique to analyze the hidden structures in the data set [32]. MDS takes a symmetric matrix $\mathbf{D}$ of proximities among pairs of objects as input,

$$\mathbf{D} = \begin{bmatrix} 0 & p_{12} & p_{13} & \dots & p_{1j} \\ p_{21} & 0 & p_{23} & \dots & p_{2j} \\ p_{31} & p_{32} & 0 & \dots & p_{3j} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{i1} & p_{i2} & p_{i3} & \dots & 0 \end{bmatrix}, \tag{2.7}$$

where p_{ij} is distance between the paired objects (i, j) in the data set. The output produced is a representation of these measurements as distances between points in a low-dimensional space, called the *MDS space*. Note that the input matrix $\mathbf{D}$ is symmetric in the sense that $p_{ij} = p_{ji}$ and $p_{ij} = 0$ for $i = j$, this type of matrix is often called a *distance matrix* [34].

2.7.1 The MDS Space

The output in MDS space is a configuration $\mathbf{X}$ that maps how the proximities are related to each other. In the case where proximities are dissimilarity measurements, the output mapping will have objects that are more similar to each other, lying more closely. While objects that are not similar, will be further away from each other. This means that it is possible to determine clusters and patterns intuitively if the output were to be in $\mathbb{R}^2$ or $\mathbb{R}^3$ space.

The mapping onto the MDS space can be expressed by

$$f : p_{ij} \to d_{ij}(\mathbf{X}), \tag{2.8}$$

meaning, there exists a *representation function* f that transforms p_{ij} onto the MDS space $\mathbf{X}$. We say that f will specify the MDS *model*, meaning, that the MDS model is simply a proposition that given proximities, after some transformation $f(p_{ij})$, are equal to the distances among points of $\mathbf{X}$

$$f(p_{ij}) = d_{ij}(\mathbf{X}), \tag{2.9}$$

where $d_{ij}(\mathbf{X})$ denotes the Euclidean distance between the pair of objects (i, j) in the MDS space X. We are free to select any representation function we seem fit.

[5] Manifold learning is an unsupervised learning method and is often associated with dimensionality reduction. More known methods are for instance the Principal Component Analysis (PCA) which reduces dimensionality by finding which feature describes most of the input data, and selects these to train or explain the data [32].

In practice, one should not try to strictly satisfy f, instead, one can restrict f to some class of functions[6].

We will not go deeper into the theory of representation functions but to have an understanding of equality (2.9) we can imaging selecting the simplest choice of f; $f(p_{ij}) = p_{ij}$. This is often referred to as *absolute* MDS. With absolute MDS we can write (2.9) as

$$p_{ij} = d_{ij}(\mathbf{X}) \tag{2.10}$$

Equality (2.10) catches the essence of MDS; representing high-dimensional input data *as it is* in low-dimensional space, thereby retaining all information in the original data.

In practice, data for a given problem can often be quite complex and of high dimensionality. If that is the case, choosing absolute MDS as the model could be a very poor choice as a lot of information may be lost with the dimensionality reduction. In fact, there is no guarantee that there actually exists a representation function f that fulfills equality (2.9) but instead, only one that approximates it.

$$f(p_{ij}) \approx d_{ij}(\mathbf{X}) \tag{2.11}$$

This approximation implies that there must exist some sort of *error*, a value that determines the *goodness of fit* of the representation function f to the distances in $\mathbf{X}$, $d_{ij}(\mathbf{X})$. In the literature of MDS this value is known as the *Stress* value. Figure 2.4 shows an example of MDS.

[6] A function f could be linear, quadratic, exponential or some other class.

Table 2.1: Example: An input matrix of distances between ten cities in Europe.

	1	2	3	4	5	6	7	8	9	10
1	0									
2	569	0								
3	667	1212	0							
4	530	1043	201	0						
5	141	617	596	431	0					
6	140	446	768	608	177	0				
7	357	325	923	740	340	218	0			
8	396	423	882	690	337	272	114	0		
9	569	787	714	516	436	519	472	364	0	
10	190	648	714	622	320	302	514	573	755	0

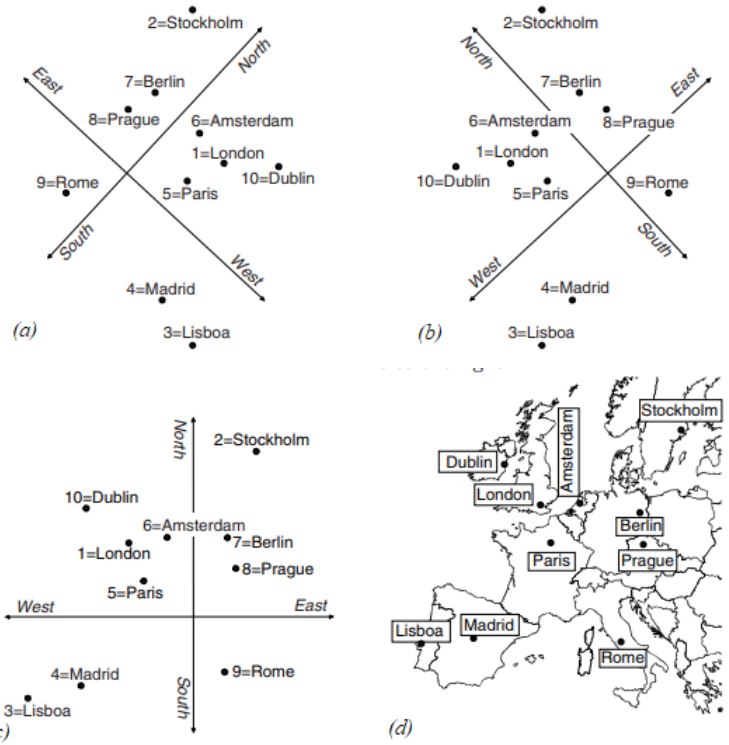

Figure 2.4: Example: The proximities are mapped into MDS space as distances, trying its best to preserve the original given data. Further, one can see how reflection,(a)-(b), rotation, (b)-(c), or translation does not affect the fact that the distances are still intact.

2.7.2 The Stress function

Whether the output produced by the MDS is a reliable representation of the original data or not, is determined by the *Stress* value. The value of Stress expresses the degree of correspondence between the distances among points produced in the output, $d_{ij}(\mathbf{X})$, and the proximities given by the symmetric input matrix $\mathbf{D}$, after some transformation $f(p_{ij})$. The optimal result for MDS is to have a Stress value of zero, i.e. the proximities in the original data is unchanged through the reduction from high to low dimensionality. This also means that Stress measures a *badness of fit* rather than a goodness of fit, where higher values of Stress indicates bad fit and poor results.

A squared error of representation is written as

$$e_{ij}^2 = \big[f\big(p_{ij}\big) - d_{ij}(\mathbf{X})\big]^2 \tag{2.12}$$

where $d_{ij}(\mathbf{X})$ is the Euclidean distance between any two points i and j in MDS space. $f(p_{ij})$ is the transformation of the dissimilarity between object i and j in the input distance matrix $\mathbf{D}$. By summing 2.12 over i and j we get the total error of an MDS representation called *Raw Stress*,

$$\sigma_r = \sigma_r(\mathbf{X}) = \sum_{(i=1)}^{n} \sum_{(j=i+1)}^{n} \big[f(p_{ij}) - d_{ij}(\mathbf{X})\big]^2 \tag{2.13}$$

$$\sigma_r = \sigma_r(\mathbf{X}) = \sum_{(i<j)} \big[f(p_{ij}) - d_{ij}(\mathbf{X})\big]^2 \tag{2.14}$$

$$\sigma_r(\mathbf{X}) = \sum_{(i<j)} w_{ij}\big[f(p_{ij}) - d_{ij}(\mathbf{X})\big]^2 \tag{2.15}$$

The notation $i < j$ in (2.14) simply says that it is sufficient to sum over only the half of the data as the original given input $\mathbf{D}$ is symmetric. w_{ij} is a fixed weighting parameter that is 1 if a proximity p_{ij} exists, 0 otherwise (data is missing). Other values for w_{ij} is also possible as long as $w_{ij} \geq 0$.

Raw Stress can also be written as a function

$$\sigma_r(\mathbf{X}) = \sum_{(i<j)} w_{ij}\big[f(p_{ij}) - d_{ij}(\mathbf{X})\big]^2 \tag{2.16a}$$

$$= \sum_{(i<j)} w_{ij} f(p_{ij})^2 + \sum_{(i<j)} w_{ij} d_{ij}(\mathbf{X})^2 - 2\sum_{(i<j)} w_{ij} f(p_{ij}) d_{ij}(\mathbf{X}) \tag{2.16b}$$

$$= \eta_\delta^2 + \eta^2(\mathbf{X}) - 2p(\mathbf{X}) \tag{2.16c}$$

where η_δ^2 is a constant as its only dependant on the fixed weights w_{ij} and the proximities p_{ij}. $\eta(\mathbf{X})^2$ is the weighted sum of squared distances, while the final part $-2p(\mathbf{X})$, is the weighted sum of the distances. The objective of MDS is to minimize function (2.16).

Kruskal's Stress

Raw Stress is itself not that informative; a low value does not necessarily indicate a bad fit. An example would be if your input matrix consisted of distances that described lengths between objects. Lets say that these objects are different cities located somewhere in the world. There are many different units of measurements that can describe the lengths between different cities. If we were to change, lets say from kilometers to meters, the MDS would give the same results but the Raw Stress value would not be the same. It would have been 1000 times as large. To avoid these type of scale dependencies, a normalized version of the Stress value can be expressed as

$$\sigma_1^2 = \sigma_1(\mathbf{X})^2 = \frac{\sigma_r(\mathbf{X})}{\sum d_{ij}^2(\mathbf{X})} = \frac{\sum \left[f(p_{ij}) - d_{ij}(\mathbf{X})\right]^2}{\sum d_{ij}^2(\mathbf{X})}, \tag{2.17}$$

since σ_1^2 is often very small, we use σ_1 to better discriminate between values. We then write

$$Stress - 1 = \sigma_1(\mathbf{X}) = \sqrt{\frac{\sum \left[f(p_{ij}) - d_{ij}(\mathbf{X})\right]^2}{\sum d_{ij}^2(\mathbf{X})}} \tag{2.18}$$

Note that we use σ_1 and $\sigma_1(\mathbf{X})$ interchangeably. The notation $\sigma_1(\mathbf{X})$ is to show that the Stress value is for the given MDS configuration $\mathbf{X}$. This also means, that an optimal configuration $\mathbf{X}$ in $\mathbb{R}^n$, must exist to be able to minimize (2.18).

2.7.3 Minimizing Stress

What solves the MDS is an iterative approach that minimizes the Stress function. *Scaling by Majorizing a Complicated Function*, or better known as the SMACOF algorithm is used to minimize Stress. It usually starts with some initial configuration $\mathbf{X}$ in a predefined $\mathbb{R}^n$ space of the input proximities, either at random or by selection. This configuration is then improved by moving around its points in small steps, iteratively, to minimize Stress and therefore converge to equality (2.10). The specific iterative optimization technique is called *majorization*. We will not be going into detail of the minimization of Stress but rather, we shall remember that Stress can be expressed as a function and look at the main principles that ensures minimization of a given function f.

2.7.4 Principles of Majorization

The main idea of majorization is that we, iteratively, replace the original complicated function f by an *auxiliary* function $g(x, z)$, where the z in $g(x, z)$ is some fixed reference value. To say that $g(x, z)$ is a majorizing function of f, the following must hold

- The auxiliary function $g(x, z)$ has to be simpler to minimize than f.
- The original function f is always smaller than, or at most, equal to the auxiliary function $g(x, z)$. That is, $f(x) \leq g(x, z)$.
- The auxiliary function touches the surface at a so-called *supporting point* z, which is at $x = z$: $f(z) = g(z, z)$.

Let the minimum of an auxiliary function $g(x, z)$ be at the point $x = x^*$. For the last two requirements above, we have the inequalities

$$f(x^*) \leq g(x^*, z) \leq g(z, z) = f(z) \tag{2.19}$$

also known as the *sandwich* inequality [34]. Inequality (2.19) has an attractive aspect of that it will ensure a monotone[7] convergence towards the solution, meaning that for each iteration the algorithm will be one step closer to minimizing the objective function $f(x)$. This feature, together with the ability to work with high-dimensional data, makes it more powerful than traditional techniques such as gradient descent [30]. Figure 2.5 shows two iteration of majorizing a function.

Algorithm 3 Pseudocode for iterative majorization (IM)

1: Set $z = z_0$, where z_0 is the starting value.
2: For iteration u, find update x^u for which $g(x^u, z) \leq g(z, z)$
3: If $f(z) - f(x^u) < \epsilon$, then stop. Here ϵ is a small positive constant.
4: Set $z = x^u$ and go to point 2.

[7] In calculus, a monotonic function is a function that is either entirely non-increasing, or entirely non-decreasing [15].

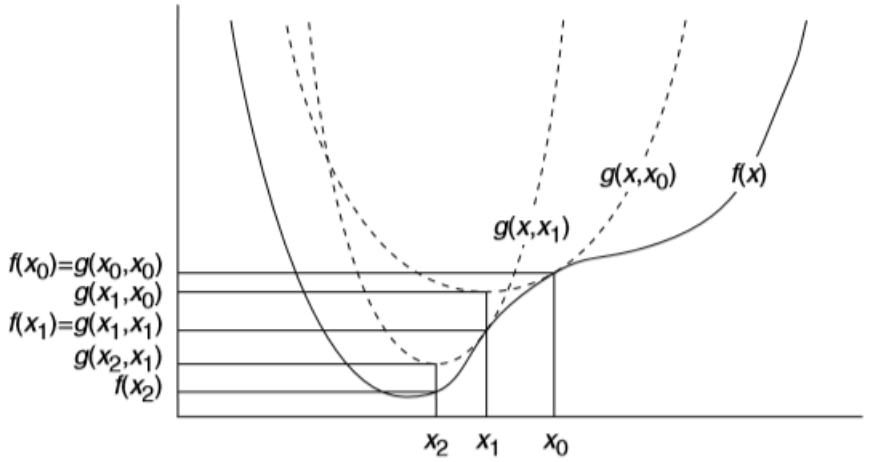

Figure 2.5: Two iterations of the iterative majorization method. We see that the first iteration finds an auxiliary function $g(x, x_0)$, that lies over the original function $f(x)$ and touches it at the supporting point $z = x_0$. The minimum of $g(x, x_0)$ is found at $x^* = x_1$, where $f(x_1)$ can never be larger than $g(x_1, x_0)$, fulfilling the sandwich inequality 2.19. The second iteration has the same procedure as the first, and the algortihm will run till convergence.

2.8 Standardization

Standardization, also known as normalization, allows features that have different scales have one and the same scale; the *Z-score*. The Z-score of a sample x of a population N can be obtained by

$$z = \frac{x - \bar{x}}{\sigma} \tag{2.20}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N - 1}} \tag{2.21}$$

where $z :=$ the Z-score of sample x. $\bar{x}$ is the mean value of the population and σ is the standard deviation of the population of N samples. The common scale makes the distribution of the N samples have zero mean and unit variance. This scaling method is often used in pre-processing of data to achieve good trade off between small subtle values and dominating large values with high variance [35].

Chapter 3

The Expert System

3.1 Introduction

An expert system that extracts the most suitable reference wells for a new well that is to be drilled was scripted in Python. The system is made up by a total of eight scripts; a main script where the core of the system is written, together with seven functions that is necessary to provide the solution. A hierarchy of the scripted expert system is illustrated in Figure 3.3. This chapter will in detail go through the methodology that is used in order to achieve the expert system that proposes a solution. Step by step, we will work ourselves through the different scripts that make up the system. The complete source code can be found in Appendix A but the chapter can be read independently.

3.2 Main.py

`main.py` describes the core of the expert system without going into such detail on the different algorithms and functions that are used. From here, well data is retrieved and pre-processed to the forms that are required of the different functions and algorithms that are used. The complete source code can be found in in Appendix A.1.

3.2.1 Setting up Python

The complete system is mainly based on unsupervised learning methods which focuses on clustering analysis, such as the k-means algorithm but theory within the nearest neighbor method has also contributed to solve the problem. Scitkit-learn is a strong and widely used library for machine learning packages for Python, and it is in this thesis the main source of imported ML methods [27]. When working with arrays, matrices, and numerical procedures in Python the recommended packages are NumPy, SciPy, and Pandas to easier deal with mathematical calculations [18] [28] [20]. We have mainly used NumPy and Pandas.

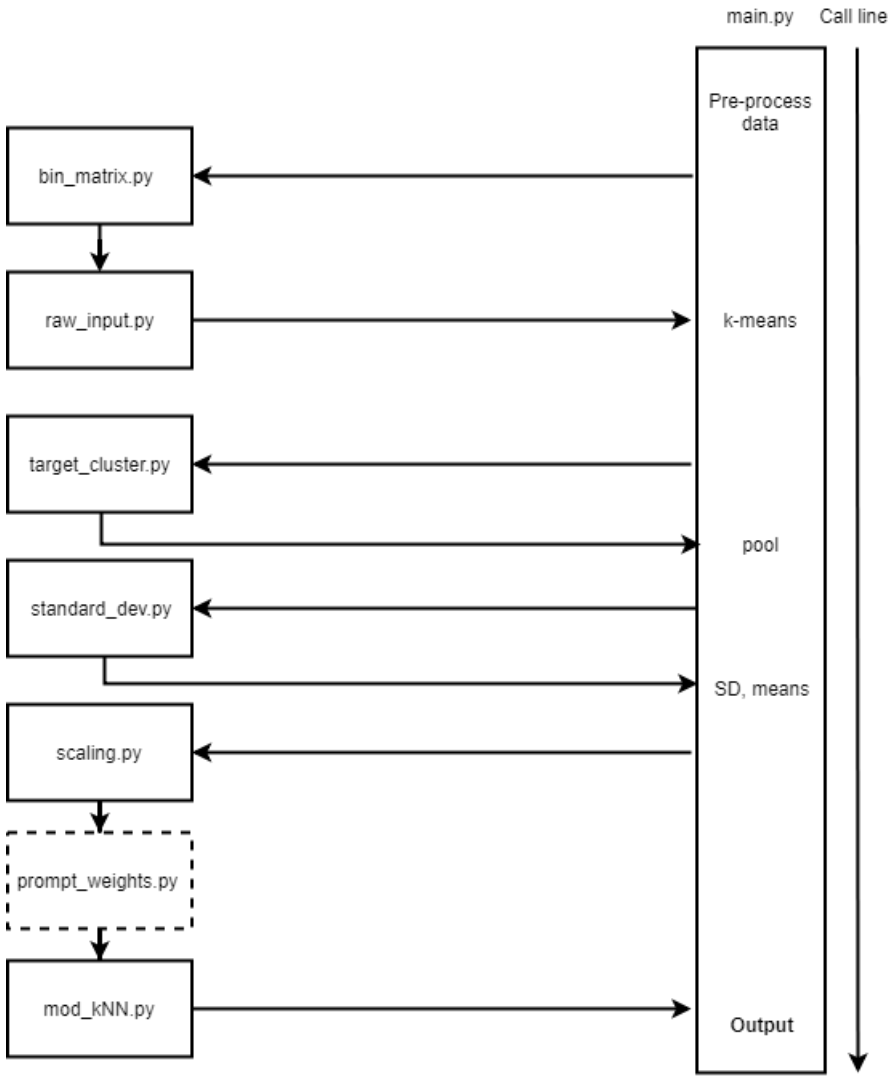

Figure 3.1: Rough sketch of the hierarchy of different scripts that make up the expert system.

For plotting, matplotlib is imported as it has a strong resemblance to MATLAB, a software that is often used at universities for educational purposes or in the industry [13] [14]. Table 3.1 summarizes all modules that the expert system has imported.

Table 3.1: Imported modules

General modules:	**Description**
os	To easy change directories (folders) when retrieving well data.
numpy	Treats data structures in Python as arrays and matrices. Simplifies linear algebra.
pandas	Easy-to-use data structures. Able to read EXCEL.
Other modules:	
k_means	Performs k-means clustering; Lloyd's algorithm.
parwise_distances	Calculates the Euclidean distance between vectors.
manifold	Performs multidimensional scaling.
matplotlib.pyplot	For plotting in Python.
Axes3D	For 3D plotting in Python.
Scripted modules:	
bin_matrix	Stores well data in suitable manner.
raw_input	Creates the input needed to perform k-means clustering.
target_cluster	Extracts the target cluster.
standard_dev	Calculates standard deviations and mean values.
scaling	Standardizes the well data.
prompt_weights	Enables an engineer's ability to weight formation layers.
mod_kNN	Uses nearest neighbor method to rank the most relevant wells.

3.2.2 Pre-processing of input data

As we recall, we have assumed that in the earliest phase of planning to drill a new well the information about the well's geological profile must exists. The geological profile of the well is taken as input and the expert system should be able to produce an output that selects the most relevant reference wells. In order to do so, there must also exist a database of available wells to select from. This was illustrated in Figure 1.1 in Chapter 1. The Norwegian Petroleum Directorate (NPD) is used as the available database of wells due to easy access to well data in EXCEL, XML, or CSV format for over 1000 wells that have been drilled on the Norwegian Continental

Shelf [8]. Figure 3.2 shows the general form of the original data that we import in the EXCEL format.

	A	B	C	D	E	F	G	H	I	J
1	Brønnbane navn	Topp dyp [m]	Bunn dyp [m]	Litostrat. enhet	Nivå	NPDID for litostrat. enhet	Dato for boreslutt	NPDID for brønnbanen	Dato oppdatert	Dato synkronisert OD
2	1/2-1	94	1777	NORDLAND GP	GROUP	113	04.06.1989	1382		23.01.2018
3	1/2-1	1777	3059	HORDALAND GP	GROUP	67	04.06.1989	1382		23.01.2018
4	1/2-1	3059	3121	BALDER FM	FORMATION	6	04.06.1989	1382		23.01.2018
5	1/2-1	3059	3407	ROGALAND GP	GROUP	131	04.06.1989	1382		23.01.2018

Figure 3.2: The well data extracted in EXCEL format.

From EXCEL format, we can use Pandas to extract the rows and columns that we find necessary. The chosen columns of interest have been 0, 1, 2, 3, and 5 which corresponds to the name of the well, top of formation in TVD, bottom of formation in TVD, name of the lithostratigraphic unit, and the NPDID for the lithostratigraphic unit, respectively, as these columns give information about a well's geological profile. The selection of these attributes (features) is often referred to as *feature engineering* and is a vital part of any process that seeks to train a suitable ML model [32] [1]. Note, later in this chapter, how the name of the well is not necessarily directly involved to solve the problem of the expert system but rather, the information is available and can easily be written to an object which again makes it easy to pinpoint which exact well is a relevant well. We should also mention that the TVD of the bottom of a formation is also a feature that is not directly involved in solving the problem, we will come to that in a bit.

The input well, as well as the available database from NPD, can be retrieved by calling the `pd.read_excel()` function. We select all rows of the EXCEL sheet to include the whole database of wells, and also we want to make sure that the input well is always the first well that is imported, this places the well at index 0 which results in easy tracking. The function returns the class DataFrame which can be inspected in the PyCharm. DataFrame has a formation that is relatable to how we visualize EXCEL [21] [1]. The sections that involves retrieving and arranging the input well data is found in Appendix A.1, line 62-126.

Finding an existing database that is large enough, and one that also has an easy importable format, has already saved us a lot of time. In Chapter 4, we look into how the system selects reference wells for a well that has been drilled by the collaborative firm. Half a day's work was spent searching through its Final Drilling Program to gather well data regarding its geological profile. Luckily, we were able to find a well that is relevant with the available database from NPD.

[1] Suitable, in the sense that selecting casing design as a feature for a well will not contribute to describe the well's geological profile. Selecting correct features that solves the problem at hand is therefore important.

3.2.3 Defining the class: Well

Python is well suited for *object oriented programming*. Object oriented programming enables us to combine data and functionality and wrap it inside an object [19]. In Python, this can be done by defining a `class Well` that lets you treat every well that is taken as input as a single object where data and functionality can be stored [24]. This makes retrieving (or writing) information from (or to) any object (well) quite easy, which we will see when we produce the output of the expert system.

The well data retrieved in the pre-processing stage is directly written to the `class Well`. Defining the class can be found at the start of **`main.py`** (Appendix A.1, line 29-46).

Excerpt 2

```
class Well:
    def __init__(self,name=None,index=None,top=None,bottom=None,
    fm=None,fm_names=None,thickness=None,thickness_sum=None,
    fm_rel=None,thickness_rel=None,depth_rel=None):
        self.name = name
        self.index = index
        self.top = top
        self.bottom = bottom
        self.fm = fm
        self.depth = self.bottom[-1]
        self.fm_names = fm_names
        self.thickness = thickness
    def __repr__(self):
        return '{}'.format(self.name)
```

The defined attributes of `class Well`, in the same order as Excerpt 2, are:

- String, Name of well
- Integer, Index of the well
- Integer, Top of formation, TVD
- Integer, Bottom of formation, TVD
- Integer, Unique NPDID of formation
- Integer, Total depth, TVD
- String, Name of formation layers
- Integer, Thickness of formation layers

where any attribute can be reached by the command `self.attribute`, where *self* is the object.

3.3 Storing data for Computation

In the process of selecting relevant wells based on geological profiles one is forced to look at an object's attributes and have some sort of justification of the choices made. One approach would be to call any object and retrieve the attribute we wish to look at whenever it is needed. Since we are going to look at every well in the database anyway, we created the **`bin_matrix.py`** script as a solution on how we want to store well data in a convenient manner. The function takes all objects as input and creates three $m \times n$ matrices; `X_formation`, `X_thickness`, and `X_depth`, where m is the number of wells that exists in the available database while n is the total number of unique formations. For our case, we have 1663 wells and 212 different categories of geological groups and formations, resulting in three 1663×212 matrices.

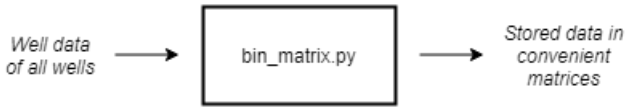

Figure 3.3: Input and output of **`bin_matrix.py`**.

Excerpt 3

```
import numpy as np

def bin_matrix(Objects):
    dim = (len(Objects), 212)
    X_formation = np.zeros(dim)
    X_thickness = np.zeros(dim)
    X_depth = np.zeros(dim)
    temp = []
    for i in range(len(Objects)):
        for j in range(len(Objects[i].top)):
            X_formation[i][Objects[i].fm[j]] = 1
            layer = Objects[i].bottom[j] - Objects[i].top[j]
            X_thickness[i][Objects[i].fm[j]] = layer
            temp.append(layer)
            X_depth[i][Objects[i].fm[j]] = Objects[i].top[j]
        Objects[i].thickness = temp
        temp = []
    return X_formation, X_thickness, X_depth, Objects
```

By having these exact dimensions we have an easy way of tracing every object that exists in the database and at the same time, storing any information regarding this object. We can at all time know that the first retrieved object from the available database will be at the first row. So, for any object retrieved as the i^{th} object will

be at row i, equivalently, the information regarding any layer j will be stored at the j^{th} column (Excerpt 3, line 10 and 11). Pay attention to how we can easily update an object's attribute by writing to its defined class attribute (Excerpt 3, line 17).

For `X_thickness` and `X_depth` we have stored the thickness of a formation and the top of a formation, respectively. The thickness is calculated based on the TVD of top and bottom of the formation while the top of formation is retrieved directly. Note how we have used certain features to define a new feature; thickness, this is in literature often called *feature extraction*. For `X_formation` we have used a binary approach; 0/1 for NO/YES. This means, that if a formation layer j exists in a given object i, we add a 1 in the i^{th} row and j^{th} column. Otherwise, we add 0. This is because we are only interested if other wells in the database have the same geological layers as our input well.

3.4 Preparing Data for k-means

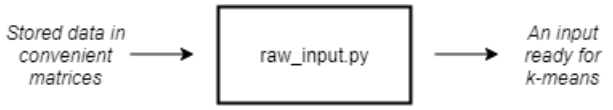

Figure 3.4: Input and output of **`raw_input.py`**.

Any function is defined to take a certain input, do something, and create an output. The same goes for the k-means algorithm that is imported from Scikit-learn. From [29], *"The input to the k-means algorithm must be a array-like or sparse matrix* $\boldsymbol{X}$*, with the shape of n_samples and n_features, also defined as the observations we wish to cluster".* **`raw_input.py`** shall take the output of **`bin_matrix.py`**, namely `X_thickness` and `X_depth`, to create the necessary input that the k-means algorithm demands. **`raw_input.py`** is called by `raw_input = cluster(X_thickness,X_depth, d)` (Appendix A.1, line 156).

Excerpt 4

```
1  def cluster(thickness,depth,fm):
2      raw_input = []
3      for i in range(len(thickness)):
4          temp = []
5          temp2 = []
6          for j in fm[0]:
7              temp.append(thickness[i,j])
8              temp2.append(depth[i,j])
9          temp.extend(temp2)
10         raw_input.append(temp)
11     return raw_input
```

`d` is a *list of lists*, which is basically the pythonic way of writing matrices [23]. It holds the geological profile of every single well, i.e. each list (well) in `d` has the list of the unique NPDID of the formations layers that make up the well's geological profile.

Our main objective is to select good reference wells, to our input well. Our approach has been to only use information of the formation layers existing in the input well. Since we always know that index 0 will be our input well at all times, its geological profile is assessed in Excerpt 4, line 6. The function then extracts all information from every other well in the database in these exact layers, ignoring any other layer that the input well does not have. This means, that wells that are later selected as the most relevant can (and often will) have formations layers that the input well does not have. This approach is based on the assumption when ignoring the existence of faults and folds, the deposition and consolidation of formations layers creates a certain pattern in their geological profile. That is, for the same depositional environment one will have the pattern that a certain layer lies on top of another layer, like how Nordland is often on top of Hordaland, and Hordaland on top of Rogaland and so on. By assuming so, regardless if there could exist other formations in a selected well or not, the complete geological profile of the input well will have a match somewhere in the geological profile of the selected well.

The output, `raw_input`, is a matrix of dimensions $m \times 2n$ that is ready for k-means. m is still the number of wells in the database. n is the number of layers our input well has. We have $2n$ since we want to look at both thickness and the TVD of top of each formation when applying k-means. This means that `raw_input` is a set of m objects, where each row m is a feature vector $\boldsymbol{x}$ of dimensions $1 \times 2n$ and thereby suitable as input for k-means.

3.4.1 Running k-means

Given that `k_means` is already imported from Scikit-learn [29], the clustering analysis is simply performed by the command `kmeans_raw = kMeans(raw_input,n_clusters=num_clusters, return_n_iter=True)`. `num_clusters` is the number of clusters one wish to have, and in our thesis this is set to fulfill Hypothesis 1, *"The expert system should be able to identify that different fieldsexists on the NCS"*. The algorithm will by default be run 10 times with different cluster centroids initializations in feature space, the set parameter `return_n_iter=True` will return the number of iterations the k-means ran on its best run. As long as the number of iterations are less than the maximum number of iterations that is set to 300 (by default), the k-means will have converged.

A simple algorithm as the k-means is also served with disadvantages, therefore care should be taken when analyzing results gained from the k-means algorithm. For instance, the k-means will cluster data even if it is fed an uniform distribution of data. Making sure of that the data one is analyzing is clustarable, is therefore important [7]. Also, the selection of the predefined k clusters can be quite complex. Usually, one wish to select a k that minimizes the WCSS and use this to evaluate if the resulting clustering is valid or not [6]. Other disadvantages involves sensitivity

to scaled data or getting stuck at a local minimum [2]. These problems will be discussed later in Chapter 5.

3.5 Retrieving the Cluster of Interest

As we recall from Chapter 2, the results gained from a clustering analysis is such that objects that lie in the same cluster are similar while objects in different clusters are dissimilar. The main objective of the function written in **`target_cluster.py`** is to take the results gained by k-means as input and retrieve the cluster of wells which our input well lies in. We call this the *target cluster*. The output of the k-means returns the coordinates of all k cluster centroids in feature space, in addition to the cluster membership of every object, i.e. which cluster an object has been assigned to. **`target_cluster.py`** assesses the latter to produce an ordered list of lists `clusters`. Each list in `clusters` refers to a cluster indexed by the k-means algorithm; list i will refer to cluster i, and will consist of the indices of wells that have been assigned to this cluster. By knowing the indices, we can easily retrieve the wells lying within any cluster by calling the defined objects.

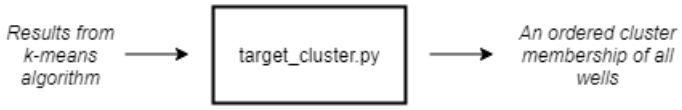

Figure 3.5: Input and output of **`target_cluster.py`**.

Excerpt 5

```
def target_cluster(kmeans,num_clusters):
    clusters = []
    clust_target = kmeans[1][0]
    for i in range(num_clusters):
        cluster = []
        for j in range(len(kmeans[1])):
            if kmeans[1][j] == i :
                cluster.append(j)
            else:
                pass
        clusters.append(cluster)
    return clusters, clust_target
```

[2] A local minimum is a solution but not the best solution

3.6 Evaluating wells in the target cluster

Now that we have the target cluster, we can assume that the wells lying within this cluster must be the most relevant wells for our input well. `main.py` extracts the wells that lies within the target cluster so that further evaluation of these wells can be done (Appendix A.1, line 168-182):

Excerpt 6

```
1  if input_well == True:
2      depths = []
3      pool = []
4      for i in clusters[clust_target]:
5          depths.append(Objects[i].depth)
6          pool.append(Objects[i])
7  spekter = range(depths[0] - 500, depths[0] + 500)
8  j = 0
9  while j < len(depths):
10     if depths[j] not in spekter:
11         depths.pop(j)
12         throw = pool.pop(j)
13         j += 1
14     else:
15         j += 1
```

Wells are gathered in the list of objects `pool` (Excerpt 6, line 1-6). The `while` loop does a removal of wells that we seem as irrelevant for the input well. The loop defines a range of $\pm$ 500 meters of the total depth of the input well, in TVD. If a well's total depth is not within this range, we deem it irrelevant and remove it. This is based on that the *target depth*, the depth, in TVD, of which the drilling operation shall reach, should be relevant.

3.6.1 Standardizing

The output we wish to produce for the expert system is a listed ordering of the most relevant wells for the input well. The remaining task is therefore to do an evaluation on which of these wells is more relevant than the other. This is done by evaluating the *nearest neighbor* of the input well. As presented in Chapter 2, the nearest neighbor method has the greedy property of selecting whichever object that is nearest to the object of interest. For our case, this would be to locate the nearest wells that are most similar to the input well within the target cluster.

We recall that the output gained at `raw_input.py` is a $m \times 2n$ matrix. The first n columns, from now on referred to as the left half, carries the information of thickness of each n formations layers existing in the input well. The last n columns, the right half, carries the information of TVD of top of the formation for each of the n layers. m is still the number of wells in the database.

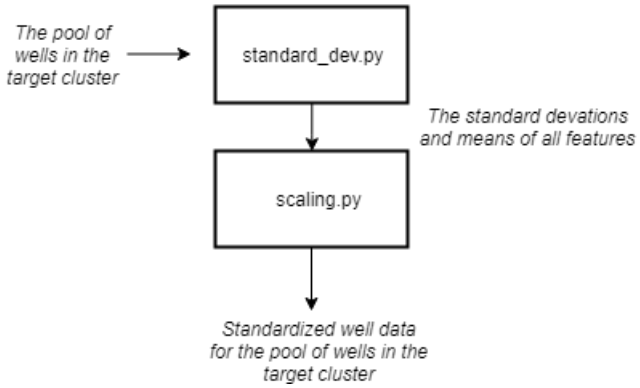

Figure 3.6: Input and output of the standardization module.

By knowing so, we can say that the left half is unordered; as thickness of formation layers is random. The right half is not though. The right half is ordered due to the use of TVD. For each column we move to the right in the right half of the matrix, the number lying within the next column will have increased. Now, if we were to have a nearest neighbor approach on this matrix, which is basically just calculating the Euclidean distances between each row vector m, we may have the problem that the right half could be dominating the left half due to their values being higher. To avoid this, we have standardized the data to have zero mean and unit variance. Standardization enables us to put different features on the same scale, by enhancing small subtle values and dampening high dominating values. The thought, is that every feature of the input well will have a certain standard score. By evaluating an arbitrary object's standard score in relation the the input well's standard score, for every feature, we should be able to locate the nearest neighbors.

`standard_dev.py` calculates the mean and the standard deviation for each feature of the wells lying in `pool`. Since the feature vector $\boldsymbol{x}$ is of dimensions $1 \times 2n$, we have that the arrays containing the means and standard deviations, `SD_thickness`, `SD_depth`, `mean_thickness`, and `mean_depth`, have the same dimensions. These values are then used in `scaling.py` to standardize the selected data. This is done by scripting the equation for standardization presented in Chapter 2, Equation (2.20). The produced output of `scaling.py`, `input_matrix`, is the standardized well data from wells in the target cluster. The dimensions of `input_matrix` is $m \times 2n$, here m is no longer the whole database but it is reduced to be the m wells in the target cluster.

3.7 Weighting

Before applying the nearest neighbor method, a function that enables an engineer to weight the importance of certain formation layers was made. The user interface provides an option where the engineer can choose to continue the analysis with or without weighting the different formation layers in `input_matrix`. See Figure 3.8. The weighting is triggered by a binary 0/1 for NO/YES in the call made to **`prompt_weights.py`** script (Appendix A.1, line 199).

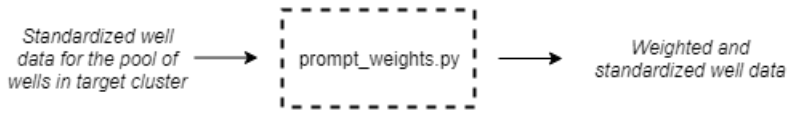

Figure 3.7: Input and output of **`prompt_weights.py`**.

```
Your target well has the following layers:
NORDLAND GP
UTSIRA FM
SKADE FM
HORDALAND GP
GRID FM
ROGALAND GP
BALDER FM
SELE FM
LISTA FM
HEIMDAL FM
LISTA FM
VÅLE FM
ÅSGARD FM
VIKING GP
DRAUPNE FM
HEATHER FM
SKAGERRAK FM
There are a total of 17 layers.
Weighting: Layers that are weighted with high values will be prioritized. Layers are weighted in relation to each other.
If you wish to weight layers please select: 1. If not, choose default by selecting: 0. Type your selection: 0
```

Figure 3.8: Enabling weighting of formation layers through user interface.

The function has two weighting arrays; W a 1×2 array, and w a $1 \times n$ array, n is still the number of layers existing in the input well. W, weights how we should prioritize thickness in formation layers in relation to the offset in TVD of top of formations. w, weights how we should prioritize the different formation layers in relation to each other. This means, that W will initially focus on how the weighting should be between the left half and the right half of the `input_matrix`. Then, w weights which formation layer that may have to be prioritized due its complexity or any other factor that the engineer bases his or her weighting on. This can also be seen in their dimensions. Therefore, the applied method of weighting in **`prompt_weights.py`** is that the first column in W weights up w and generates the weighting for the left half, then the second column in W weights up w, generating the weighting for the right half. The result is the total weighting array `weights` of dimensions $1 \times 2n$ which is used to weight each row in `input_matrix`.

Example 3.7.1 *For* $W = [1, 2]$ *and* $w = [1, 1, 1, 1]$ *the* **`prompt_weights.py`** *will produce weights* $= [1, 1, 1, 1, 2, 2, 2, 2]$, *where the number of formation layers* $n = 4$.

The weighting part of the expert system is applied mainly to ensure a degree of freedom to the engineer that is using the system. This is important, as any solution that comes out of a ML system should not be a definite solution to the task at hand but rather, the solution serves as helpful information that engineer can use in further decision making.

3.8 Selecting most relevant wells

When running the expert system, the weighting part is often selected as default. Meaning that it is basically skipped and has no influence on the data we assess. Therefore the next function after standardization is one that shall find the nearest neighbors of the input well by taking `input_matrix` as input. The nearest neighbor algorithm is not explicitly used in the sense that is neither used as a classifier nor regression but rather, the written function **`mod_kNN.py`** is based on the greedy property of the nearest neighbor rule (Chapter 2.4) and uses Euclidean distance to arrange the ordering of the most relevant wells. Objects with shorter distances will be more relevant than objects with longer distances as the distance will be a measure of deviation. The distances is labeled as the Nearest Neighbor Value and is written to the the object's attribute (Excerpt 7, line 6):

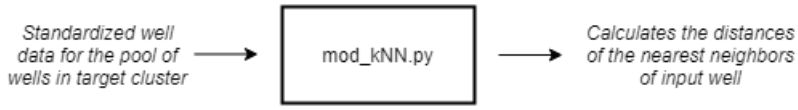

Figure 3.9: Input and output of **`mod_kNN.py`**.

Excerpt 7

```
1 from sklearn.metrics.pairwise import pairwise_distances
2 def kNN(input_matrix,pool):
3     neighbor = pairwise_distances(input_matrix, input_matrix)
4     # Write to object's attribute
5     for i in range(len(pool)):
6         pool[i].nn = neighbor[i,0]
7     return pool
```

Now, we can simply sort the objects in `pool` after the Nearest Neighbor Value and print them to console. This is the final output of the expert system, and it shows an ordered selection of the most relevant wells for a new well that is to be drilled (Appendix A.1, line 205-207 and line 229-232).

Excerpt 8

```
def sort_func(Well):
    return Well.nn
pool_final = sorted(pool, key=sort_func)
...
print('Target well: ', pool_final[0].name)
print('Well name', 'Nearest Neighbor Value')
for i in range(1,len(pool_final)):
    print(pool_final[i].name, pool_final[i].nn)
```

Chapter 4

Testing the Expert System

4.1 Introduction

This chapter will present and analyze the results obtained by the expert system that is developed. We divide the chapter into two sections; the first section will look into the results obtained with the test set consisting of 16 wells drawn from two different fields on the NCS. We recall, that the expert system is developed based on the assumptions and decisions made on a smaller test set and then made scalable, so that the system simply needs to retrieve a larger available database of well data. The second section, will therefore present the results obtained when including the whole database that is available at the NPD. We will also be including a well that was drilled by the collaborative party in order to get a measure of credibility of the system. In addition, a few assumption will be made later in the chapter in hope for improvements, we therefore summarize the assumptions that already have been made:

Assumption 1: *We should be able to find relevant wells based solely on a well's geological profile.*

This is the naive approach that is fundamental for solving the problem formulated in the thesis.

Assumption 2: *Faults and folds does not exist, meaning, that the way formations are layered exhibits a certain pattern.*

Which we used to justify why we only consider the formations layers in that exists in the input well.

4.2 Test set

The test set consists of eight wells from the Ivar Aasen field and eight wells from the Valhall field, 16 wells in total. The wells are listed in Table 4.1.

Table 4.1: The test set consists of the following wells:

Ivar Aasen	Valhall
16/1-1	2/8-4
16/1-7	2/8-6
16/1-9	2/8-8
16/1-11 A	2/8-9
16/1-11	2/8-10
16/1-16 A	2/8-11
16/1-16	2/11-1
16/1-19 S	2/11-4

We have selected well 2/8-4 to be the new well that is to be drilled. For the k-means in `main.py, line 155`, we set the numbers of clusters `num_clusters` = 2 to test **Hypothesis** 1, in other words, we want to be able to distinguish the Ivar Aasen field from the Valhall field. The weighting array $W = [W_1, W_2]$ in `prompt_weights.py` shows how we wish to prioritize thickness in formation, W_1, in relation to the offset in depth of formation tops, W_2. The values that are set for W are stated in the tables to come. Due to no experience within the field, we select the weighting array w to be default; an $1 \times n$ array of ones which weights the different formations layers equally. This gives the following configuration; `Input_file = 2/8-4.xls` [1], `n_clusters = 2`, and `W =` $[W_1, W_2]$.

The expert system will identify and extract the cluster of wells that the input well 2/8-4 lies within, namely the target cluster. A modified Nearest Neighbor search is then used within the target cluster to list the final solution of the most relevant wells of the input well. The ordering of most relevant wells are then printed based on the Nearest Neighbor Value. Note that there are only six relevant wells that are listed in the presented tables. Well 2/11-1 has been excluded due to the depth assessment in `main.py`; if the total TVD of the well is not within ± 500 meters, it seen as not relevant (Appendix A.1, line 168).

Table 4.2, 4.3, and 4.4 shows the final results when running the test set through the expert system.

[1] We use .xls to denote that well 2/8-4 is taken from the Excel file Input.xls. See Appendix ??.

Table 4.2: Selected reference wells for well 2/8-4.
W: [1, 1]

Reference Well	Name	Nearest Neighbor Value
1	2/11-4	5.85
2	2/8-9	6.70
3	2/8-10	7.00
4	2/8-6	7.59
5	2/8-11	7.72
6	2/8-8	8.32

Table 4.3: Selected reference wells for well 2/8-4.
W: [2, 1]

Reference Well	Name	Nearest Neighbor Value
1	2/8-6	10.76
2	2/8-10	11.38
3	2/11-4	11.48
4	2/8-11	11.72
5	2/8-9	11.94
6	2/8-8	12.29

Table 4.4: Selected reference wells for well 2/8-4.
W: [1, 2]

Reference Well	Name	Nearest Neighbor Value
1	2/11-4	6.29
2	2/8-9	9.07
3	2/8-10	10.75
4	2/8-11	12.69
5	2/8-6	13.12
6	2/8-8	13.97

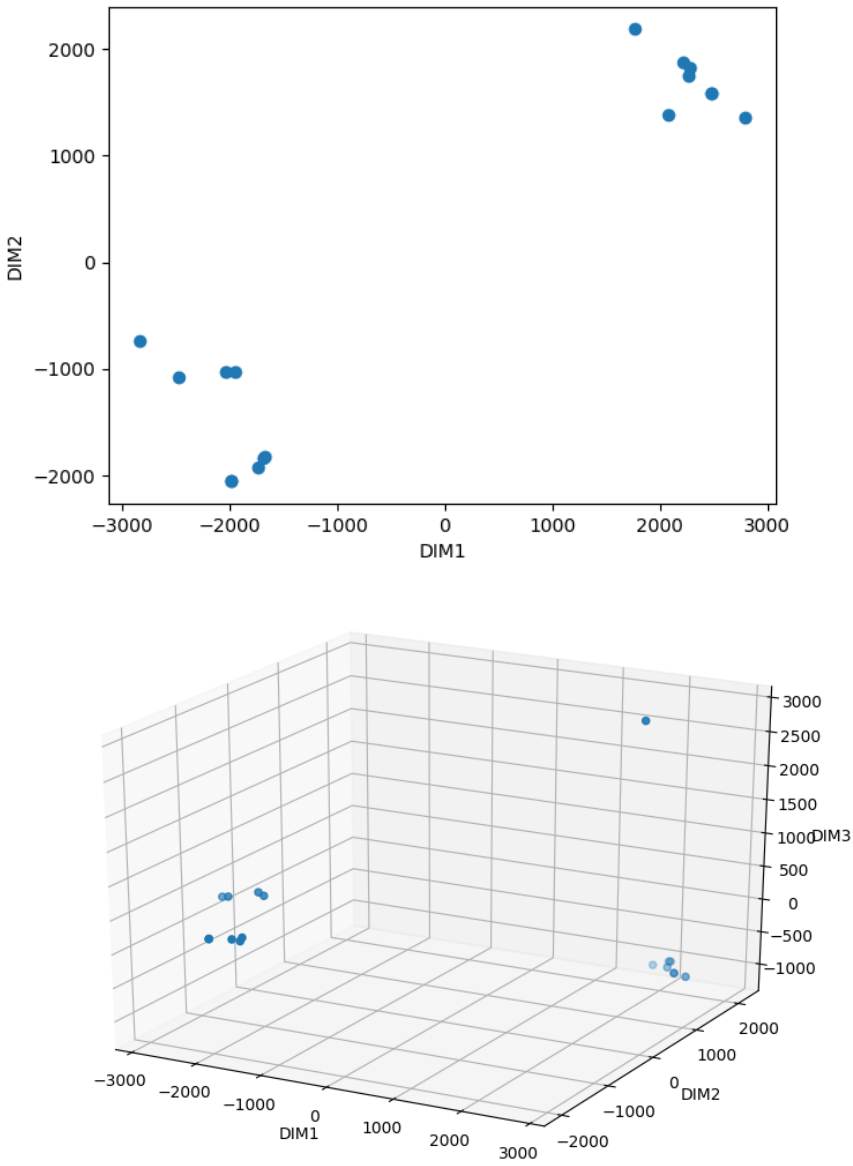

Figure 4.1: MDS is clearly a strong visualization tool, given that the Stress-1 value is reasonable. Here, the Stress-1 value is $\sigma_1 = 0.00897$. The dimensional reduction technique enables the engineer to look at the data in an intuitive way to gather information.

4.2.1 Analysis

MDS is used to visualize the high dimensional data that is taken as input in a low dimensional space. It is run separately of the k-means algorithm with the sole purpose of trying to visualize what the k-means is solving. $\mathbb{R}^3$ is the selected dimension of the MDS space but we have included plots both in $\mathbb{R}^2$ and $\mathbb{R}^3$ in Figure 4.1 to get a good view of the results when running the script with the test set. From Figure 4.1, we can clearly see that the objects in $\mathbb{R}^2$ are separated in two clusters, Valhall in the bottom left corner and Ivar Aasen in the top right corner. When we include the plot in $\mathbb{R}^3$ space, more information is revealed. The outlier that is present in the Ivar Aasen cluster (on the right) is much more clear now that the dimension is increased. The MDS technique returns a Stress-1 value of $\sigma_1 = 0.00897$, which indicates that the low dimensional representation has less than one percent badness-of-fit after being reduced from high to low dimensional space. Making it a good representation of our original data.

4.2.2 Remarks

The purpose of using a test set is to develop an expert system that is scalable. A smaller set of objects will save computational and increases the efficiency towards creating a final system. At the same time, it enables us to better analyze the results gained at each step throughout the script, making it easier to work with errors and deviations to form improvements for the expert system.

4.3 Complete database with known outcome

At this stage, the test set has enabled the expert system to reach a certain level of credibility. We now run the system on the database of wells that is available at the NPD's official website. Also, a well that has been drilled by the collaborative firm, referred to as *Well X*, is included and we select it to be the new well that is to be drilled, making a total of 1663 wells in the available database. The final results retrieved by the expert system can then be compared to the Final Drilling Program of *Well X* to assess the quality of the system.

The weighting array is set to $W = [1, 1]$ throughout the thesis if not stated otherwise. This means, that thickness in formation and the offset in depth of formation tops are equally weighted. The weighting array w is again default. The number of clusters is chosen to test **Hypothesis** 1 and is set to $n\_clusters = 120$, which is the number of fields that exists in the available database from NPD [8]. The configuration is summarized as; `Input_file = Well X.xls` , `n_clusters = 120` , and `W = [1,1]` .

Table 4.5 shows the results after the initial run when using the database from NPD. Starred wells (* or **) [2], are wells that in reality were picked to be reference wells for input well.

Table 4.5: Selected reference wells for *Well X*.

Reference Well	Name	Nearest Neighbor Value
1	Well A*	5.29
2	Well B*	7.11
3	Well C**	7.21
4	Well D**	7.84
5	Well E	8.41
6	Well F	11.94

The system is able to select four out of a total of eight wells that were originally selected as reference wells for *Well X*.

4.3.1 Analysis

As an initial run, being able to retrieve four out of eight reference wells and also ranking them as the most relevant wells, is an acceptable result. However, we seek to have a system that can be more accurate. To achieve this, we make a few assumptions in hope for an improvement of the expert system.

Assumption 3: *The layers that are categorized as No Formal Name and Undifferentiated, are more disruptive than helpful in the search of good reference wells.*

[2] * and ** is to distinguish importancy. In the final Drilling Program of *Well X*, the ** wells are highlighted as important sources for certain choices that were made to plan for its operation. As the expert system only bases its choice on geological profiles, it can not distinguish which of the starred wells are more relevant than the other beyond this knowledge.

There exists layers that are categorized as *No Formal Name* [3] and *Undifferentiated* [4] in the database of NPD. These are layers that has not yet, or can not, be assigned to a formation or group category. By knowing so, it would only add more uncertainty if the expert system were to examine a certain Undifferentiated layer in one well with an Undifferentiated layer in another, as there is a good chance that their properties are different. The same goes for No Formal Name layers. With this assumption we allow ourselves to exclude these layers of uncertainty until a better definition of the layers is achieved, in hope for a better result. The layers are removed by the `while loop` in **`main.py`** (Appendix A.1, Line 131). The following table presents the results in the adjusted run:

Table 4.6: Selected reference wells for *Well X*.
Removed uncertain layers.

Reference Well	Name	Nearest Neighbor Value
1	Well A*	5.44
2	Well G*	5.55
3	Well B*	6.00
4	Well C**	6.01
5	Well D**	7.37
6	Well F	7.77
7	Well H*	10.23

We observe that the cluster has two new members, Well G* and Well H*, while Well E was not picked, allowing the system to be a bit step closer of selecting all eight wells that were originally picked.

Now, for the remaining two reference wells that the system is not able to retrieve, it would make sense to make another naive assumption; that the remaining two wells that were originally selected lie somewhere close to the target cluster where the input well lies in. We formulate the assumption as:

Assumption 4: *All relevant wells for the input well must lie somewhere close in the neighborhood of clusters.*

What we wish to assume, is that, if there exists some sort of a neighborhood $N(c_i)$ of i clusters c, where all wells *within* this neighborhood $N(c_i)$ have relevancy to the input well, while all other wells that lie *outside* $N(c_i)$ have no relevancy. Then surely, Assumption 4 must be a reasonable guess.

[3] No formal name is used for units that have not yet received a formal name, as for example shaly intervals in the Hordaland Group in the North Sea. Exceptionally, informal names have been used to assign reservoir intervals, for example Intra Balder Formation sandstone, Intra Draupne Formation sandstone and Intra Heather Formation sandstone. [8]

[4] Undifferentiated has been used for group-intervals that based on the data available could not be subdivided into formations. This is often the case far from the type area or in condensed sections. [8]

A short script is written to find the cluster that is closest to the target cluster in order to see if the remaining two wells are present as assumed. This is done by using the nearest neighbor rule; we locate the closest cluster centroid and extract the wells lying in this cluster, see `main.py` (Appendix A.1, Line 235). The wells in Table 4.7 has arbitrary ranking and we can observe that the system is capable of finding the two remaining wells in the closest neighbor cluster.

Table 4.7: Wells lying in the cluster closest to the target cluster.

Well
Well I
Well J
Well K
Well L
Well M*
Well N*

To assess the consistency of the system, we did 30 runs to see how often it would be able to select the eight originally picked reference wells. Table 4.8 shows how many times the reference wells were found in the targeted cluster and in the cluster closest to it. Table 4.9 present the same wells with their average nearest neighbor value through 30 runs. The returned `return_n_iter` parameter did never exceed the maximum set number of iterations of 300, this means that the k-means converged in all runs. The run time of the expert system is heavily dependant on the time an engineer uses on the weighting function but if we ignore that, the expert system is able to propose a solution within 20 seconds on average.

Table 4.8: The precision of the expert system's ability to select the originally picked reference wells through 30 runs.

Well	In target cluster	In nearest cluster	Not found
Well A*	30/30	0/30	0
Well B*	30/30	0/30	0
Well C**	30/30	0/30	0
Well D**	29/30	0/30	1
Well G*	30/30	0/30	0
Well H*	24/30	5/30	1
Well M*	0/30	26/30	4
Well N*	0/30	26/30	4

Table 4.9: The average nearest neighbor value for the selected

Well	Average nearest neighbor value
Well A*	5.14
Well G*	5.71
Well B*	6.00
Well C**	6.05
Well D**	7.40
Well H*	10.02
Well M*	only in neighbor cluster
Well N*	only in neighbor cluster

Chapter 5

Discussion

This chapter shares our thoughts on some of the essential decisions that were made during the thesis that led to the final result. First, we will try to justify our selection of features that solves the problem of selecting relevant wells, here we will focus on the subject *dimensionality*. Further, the k-means algorithm and multidimensional scaling method is put on the spot; how is it actually used in our case and how reliable are the results? We will also look at how the gained results corresponds to the hypotheses presented in Chapter 1. We summarize the hypotheses as

Hypothesis 1: *The expert system should be able to identify that different fields exists on the NCS.*

Hypothesis 2: *The expert system should be able to select relevant reference wells.*

5.1 Representative Data

Feature engineering is an important part of training any ML project successfully and involves the selection of well suited features that can be used to train a model. For an unsupervised learning which focuses on cluster analysis, it would mean that the selected features must be clusterable. But how do we know that the data we are working with can be clustered at all? Hypothesis 1 is made on the idea that the profile of a well *can* be uniquely described by the pattern of which the different formations layers forms the profile. If one were to drill a hole at a certain area, study its core sample and define the geological profile of the core. If we then drill a hole simply a meter away from that hole, this core sample will be, at some point along its profile, different from the first one. Surely then, geological profiles of wells can be put into certain categories. What these categories are, is yet unknown, hence, we formed Hypothesis 1 to test if we could categorize them into fields.

From the start of the thesis we had a naive approach by assuming that if we could find wells that are a good match of the input well's geological profile then surely, these wells must be relevant reference wells. But at that point, what was a good match? Was it simply that the formations of the input well should exist in

the other wells, or was it necessary to include more features to determine a good match? Through user involvement and exploring how the engineers work with this sort of problem, it was narrowed down to that thickness of formation and where the top of the formation places in TVD, plays an important role. This led to the feature vector $\boldsymbol{x}$ of dimension $1 \times 2n$ where n is the number of formations layers that exists in the input well. $2n$ is the required dimension to evaluate both thickness and the offset in TVD of top of formations as describing a well's geological profile must involve the formation layers it consists of. An afterthought is to include the *"layer"* of sea water that lies on top of the geological profile as an feature, since this is also important when planning to drill a well.

5.1.1 The Curse of Dimensionality

The process of defining the feature vector $\boldsymbol{x}$ is based on the *curse of dimensionality*. At an early stage of the thesis, the k-Means was run with all formations as features, i.e. the feature vector $\boldsymbol{x}$ of dimension $1 \times 2n$ where n was the total number of formations layers that exists *in the available database*, which is 211 different formations. Though, this selection of feature vector will have a lot of zeros in it, as a layer may not exist in a given well, it will still force the feature space to be a configuration in $\mathbb{R}^{422}$. With increasing dimensions the distribution of samples in feature space becomes more sparse, i.e. the term *closeness* between two samples becomes less intuitive [1]. If the dimension were to go to infinity, the Euclidean distance between the nearest and the furthest point in feature space would converge to be the same [36]. This high dimensional problem is widely known as the *curse of dimensionality*. Different techniques to avoid this problem involves dimensionality reduction methods or simply an increase of the number of training samples. But the latter is not always easy, in fact, the number of samples needed to reach the desired distribution in feature space grows exponential with the number of dimensions [32] [3].

In k-means, by studying the obtained Within-Cluster Sum of Squares (Chapter 2, Section 2.6) for both feature vectors with the same number of clusters, we clearly see that with increased dimensionality we have a more sparse distribution of our objects in feature space. With n being all formations layers in a database, the returned WCSS value was $36,861,868,939$. While for n restricted only to the formations in the input well, the WCSS returned $1,805,904,145$ [2]. The first thoughts on the WCSS values are that they are out of this world; the numbers are incredibly high. But then again, how intuitively can we interpret data in high dimensionality space? We can not. It is even hard to state that if the WCSS are good or not. But what we do know, is that the WCSS is a minimization of the configuration in feature space and that the k-means we ran with the selected feature vector were able to solve our problem, while the intitial feature vector were quite inconsistent in the returned target cluster. Therefore, by focusing on the problem at hand;

[1] Less, is an understatement. Pay attention to the WCSS values in the next paragraph.

[2] The WCSS values will always differ slightly but the k-means converges around the same value. The returned number of iterations that gave the best result is an indication of that the k-means has in fact converged.

finding relevant wells for the input well, the selection of the feature vector $\boldsymbol{x}$ to be of the size $1 \times 2n$, where n is the number of formations layers that exists in the input well, is justified by the fact that we are reducing a configuration $\mathbb{R}^{422}$ to $\mathbb{R}^{2n}$.

5.2 k-means

By choosing a feature vector $\boldsymbol{x}$ where the dimension is restricted only to the formations existing in the input well, what does this say about the output of k-Means? Since the features we use is only related to the input well, the clusters we gain must exhibit the following properties

- objects within the same cluster are similar to *the input well*
- objects from different clusters are dissimilar to *the input well*

How other objects relate *to each other*, which the initial feature vector of size $\mathbb{R}^{422}$ would have ensured, is not apparent in our results. The approach in this thesis is clearly to use clustering as a means to an end and not to produce a solution in the algorithm itself. We wish to pinpoint where to look for relevant wells in the large database of available well data, and then do further analysis from there. Although this approach does not go well with Hypothesis 1, important knowledge was gained by testing that hypothesis. We do believe though, that if our expert system were to be in an optimal state, and we run the system where all wells available get their turn to be the input well, good clusters may occur and both Hypothesis 1 and 2 would be fulfilled. But this also requires that a solution of what the most relevant wells for a well is, exists.

A choice of not standardizing the input data of k-means was made. It is based on that longer distances tend to be more accurate than shorter distances, which results in that patterns are more visible with longer distances in the feature space [17] [3]. By inspecting the different scales of thickness in formation and the TVD of top of formation, we reasoned that they are indeed different but not to the degree that we think one will totally dominate the other. A few test runs were executed with standardized input for k-means and the returned target cluster did often have arbitrary wells for every run. The approach was inconsistent and therefore neglected.

5.3 Multidimensional Scaling

We recall, that MDS produces a mapping that shows how proximities are related to each other. Since the selection of our feature vector only regards to the formations within the input well, the mapping in MDS space would have a different meaning. Like we pointed out for the k-means, the proximities in MDS space shows how they are related *to the input well*. Objects lying close to the input well, is more similar

[3] The source is for MDS but both MDS and k-means uses Euclidean distance to converge.

than objects lying far from the input well, how other objects relate to each other is obscured.

Results from the MDS method shows that the curse of dimensionality is in fact a problem. MDS has an $m \times m$ distance matrix as input, which means that we will be increasing dimensionality of the MDS space when the number of objects are increased. In Chapter 4, the test set of 16 wells showed good results when using MDS. The Stress-1 value was $\sigma_1 = 0.00897$ which indicates a low badness of fit, and the representation in MDS space has two clear clusters for Ivar Aasen and Valhall wells, this means that the MDS works as a visualization technique in this case. When running MDS on the available database of 1663 wells, the returned Stress-1 value was $\sigma_1 = 0.370$ which is quite high as the desired value should be towards zero. The representation in $\mathbb{R}^3$ is given in Figure 5.1 and does not work in our favor of visually interpreting high-dimensional data.

Stress values for high-dimensional data is hard to interpret and the solution of decreasing this high Stress-1 value is to increase the number of dimensions to reduce to [17]. As the MDS is mainly used as a visualization technique, increasing the MDS representation to any space above $\mathbb{R}^3$ would not make sense as it is hard to intuitively visualize data above this dimension. MDS is therefore discarded when we include the available database.

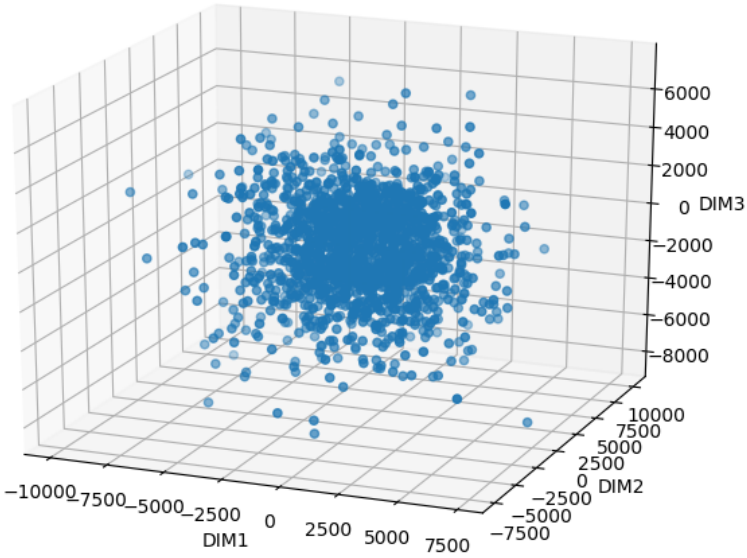

Figure 5.1: MDS for the complete available database gives a bad representation of the high-dimensional data.

Chapter 6

Summary and Conclusion

This thesis has looked into a machine learning approach for determining relevant reference wells for a new well that is to be drilled in the NCS. On the naive assumption that the problem could be solved based solely on a well's geological profile, an expert system using unsupervised learning methods, such as the k-means and the nearest neighbor rule, was developed. The programming language Python, and the machine learning library Scikit-Learn, are the essential tools that were used throughout the development, and the selection of these is based on the desire to motivate for in-house development within an oil firm, as they are programs that are readily available and easy to learn.

The following hypotheses were made to test the expert system:

Hypothesis 1: *The expert system should be able to identify that different fields exists on the NCS.*

Hypothesis 2: *The expert system should be able to select relevant reference wells.*

The expert system was developed based on results and assumptions made on a smaller test set of 16 wells, eight wells from the Ivar Aasen field and eight from the Valhall field. The test set showed good results when achieving to separate two clusters that represented each field, therefore Hypothesis 1 and 2 is fulfilled.

A database at the NPD and a well drilled by the collaborative firm, *Well X*, was made available for the expert system in order to test a real scenario. The Final Drilling Program of *Well X* was used as a solution for which wells that were originally picked for *Well X* in the real operation. The expert system showed good results by selecting all eight wells that were originally picked, where six of these ranked in the top 7 most relevant wells. All eight were chosen if we looked at the top 13 most relevant wells. The expert system is in this case, able to fulfill Hypothesis 2 but not Hypothesis 1. On average, the complete system spends 20 seconds purposing a solution.

6.1 Further Work

As only a single well from the collaborative firm has been put to test, we suggest that a large number of labeled data should be run through the expert system to test its credibility. This requires that information regarding wells drilled by the collaborative firm (or any other firm), and the reference wells selected for these, are made available, and that time and effort is put in to extract and pre-process well data for the implementation in our expert system. This action can improve the expert system further.

More testing needs to be done on the weighting function, `prompt_weights.py`. In this thesis, the weighting function is just a simple proposal that serves to enable input from the engineer that is working with the problem. This means, that not enough testing has been done on the function as the candidate is not the suited engineer to weight formations up against each other. If an optimal weighting function can be achieved one can start looking in to implementing a learning model in the function itself. If there is a good weighting function that the engineer is actively using, methods within reinforcement learning can be implemented to learn and understand the best weightings in different scenarios. Such a learner would only become smarter with time and with frequent use.

More user involvement must be enabled to ensure that the best practice is implemented to the system. The amount of user involvement in this thesis has unfortunately been sub-optimal. Availability and the fact that the thesis has been a maturity project, where the quality of the questions asked became better and better towards the end, are the main reasons for this.

It would be interesting to try principal component analysis (PCA) on the data. What differs in PCA and MDS is that PCA looks for features, or rather, some transformation of these features called *principal components*, that explains the most variance in our data base. Based on this, PCA can select the necessary principal components to represent a sufficient amount of the data in a lower vector space. We do admit that too much time and focus were put into MDS. Actually, the initial approach in the feasibility study sketched that MDS should be performed to reduce dimensionality to $\mathbb{R}^2$ or $\mathbb{R}^3$ configuration, then k-means would be performed in the MDS space, given that the Stress-1 value was acceptable. This approach was neglected.

We summarize the work of interest as bullet points:

- Test the expert system for more cases of which we know the outcome of, in order to measure its credibility.
- Put the weighting function to test and determine if a better solution is needed.
- Share work with community and enable more user involvement to ensure that best practice is implemented.
- Try PCA to see if the dimensionality of the selected feature vector can be reduced.

Bibliography

[1] *class pandas.DataFrame*. https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html. Accessed: 2018-06-12.

[2] *Coursera - Machine Learning by Standford University. Adjunct Professor Andrew Ng*. https://www.coursera.org/learn/machine-learning/home. Accessed: 2017-09-05.

[3] *Curse of Dimensionality - Georgia Tech*. https://www.youtube.com/watch?v=OyPcbeiwps8. Accessed: 2018-06-16.

[4] *Digitalisering forandrer livene våre*. https://www.aftenposten.no/brandstudio/feature/v/equinor/digitalisering/. Accessed: 2018-03-18.

[5] *Digitalisering skal gi bedre offshore-operasjoner*. https://sysla.no/offshore/digitalisering-skal-gi-bedre-offshore-operasjoner/. Accessed: 2018-03-18.

[6] *Disadvantages of the k-means clustering*. https://www.inovex.de/blog/disadvantages-of-k-means-clustering/. Accessed: 2018-06-16.

[7] *Drawbacks of k-means*. https://stats.stackexchange.com/questions/133656/how-to-understand-the-drawbacks-of-k-means. Accessed: 2018-06-16.

[8] *Factpages, Norwegian Petroleum Directorate*. http://factpages.npd.no/factpages/. Accessed: 2018-06-01.

[9] *Features - iQx*. https://www.agr.com/our-capabilities/software/iqx/features. Accessed: 2018-06-16.

[10] *Graphical example of Vornoi partition*. https://upload.wikimedia.org/wikipedia/commons/d/d9/Voronoi_growth_euclidean.gif. Accessed: 2018-06-13.

[11] *iQx - Intelligent Well Data Management*. https://www.agr.com/our-capabilities/software/iqx. Accessed: 2018-06-16.

[12] *Machine Learning - What it is and why it matters*. https://www.sas.com/en_us/insights/analytics/machine-learning.html#. Accessed: 2017-09-03.

[13] *MATLAB; Math. Graphics. Programming*. https://se.mathworks.com/products/matlab.html. Accessed: 2018-01-15.

[14] *Matplotlib: Python plotting*. https://matplotlib.org/index.html. Accessed: 2018-01-15.

[15] *Monotonic Function*. http://mathworld.wolfram.com/MonotonicFunction.html. Accessed: 2018-06-13.

[16] *Msc in Data Science: Nearest Neighbors Algorithms in Euclidean and Metric Spaces*. https://www.youtube.com/watch?v=rho8QqiHOe4. Accessed: 2018-06-13.

[17] *Multidimensional Scaling*. http://www.analytictech.com/borgatti/mds.htm. Accessed: 2018-06-16.

[18] *NumPy*. http://www.numpy.org/. Accessed: 2018-01-15.

[19] *Object Oriented Programming*. https://python.swaroopch.com/oop.html. Accessed: 2018-06-16.

[20] *Pandas Data Analysis Library*. https://pandas.pydata.org/. Accessed: 2018-01-15.

[21] *pandas.read$_e$xcel*. https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_excel.html. Accessed: 2018-06-12.

[22] *PyCharm*. https://www.jetbrains.com/pycharm/. Accessed: 2018-06-16.

[23] *Python - Lists*. https://www.tutorialspoint.com/python/python_lists.htm. Accessed: 2018-06-16.

[24] *Python classes*. https://docs.python.org/3/tutorial/classes.html. Accessed: 2018-06-12.

[25] *Quotes about Python*. https://www.python.org/about/quotes/. Accessed: 2018-06-16.

[26] *Reinforcement Learning*. https://www.technologyreview.com/s/603501/10-breakthrough-technologies-2017-reinforcement-learning/. Accessed: 2017-09-05.

[27] *Scikit-learn*. http://scikit-learn.org/stable/. Accessed: 2018-01-15.

[28] *SciPy*. https://www.scipy.org/. Accessed: 2018-01-15.

[29] *sklearn.cluster.k$_m$eans*. http://scikit-learn.org/stable/modules/generated/sklearn.cluster.k_means.html. Accessed: 2018-06-12.

[30] *SMACOF Algorithm*. http://scikit-learn.org/stable/modules/generated/sklearn.manifold.smacof.html. Accessed: 2018-06-13.

[31] Richard S. Sutton and Andre G. Barto. *Reinforcement Learning: An Introduction*. Bradford Book, 1998.

[32] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn TensorFlow*. O'Reilly, 2017.

[33] Hastie, Tibshirani & Friedman. *The Elements of Statistical Learning, Second Edition*. Springer, 2009.

[34] Ingwer Borg & Patrick J.F. Groenen. *Modern Multidimensional Scaling*. Springer, 2005.

[35] Ismail Bin Mohamad and Dauda Usman. *Standardization and Its Effects on K-Means Clustering Algorithm*. Maxwell Scientific Organization, 2013.

[36] Michael Steinbach, Levent Ertöz, and Vipin Kumar. *The Challenges of Clustering High Dimensional Data*. Springer, 2004.

[37] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Massachusetts Institute of Technology, 2009.

[38] *What is the difference between artificial intelligence and machine learning?* https://www.forbes.com/sites/bernardmarr/2016/12/06/what-is-the-difference-between-artificial-intelligence-and-machine-learning/#60b859c62742. Accessed: 2017-09-03.

Appendix A

Python Source Code

A.1 main.py

```
######### The Expert System #########
# Shall select relevant wells for a new well that is to be drilled.
#
# In total, the expert system is expressed by the following scripts:
#   - main.py
#   - bin_matrix.py
#   - formation.py
#   - mod_kNN.py
#   - prompt_weights.py
#   - raw_input.py
#   - relevancy.py
#   - scaling.py
#   - standard_dev.py
#   - target_cluster.py
#########################################

## General import
import os
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

## Import from Scikit-learn
from sklearn.cluster import k_means as kMeans
from sklearn import manifold
from sklearn.metrics.pairwise import pairwise_distances

## Define class Well to make objects
class Well:
    def __init__(self,name=None,index=None,top=None,bottom=None,fm=None,
                 fm_names=None,thickness=None,thickness_sum=None,
                 fm_rel=None,thickness_rel=None,depth_rel=None):
        self.name = name                    # Name of well
        self.index = index                  # Index of well
        self.top = top                      # Top of fm, TVD
        self.bottom = bottom                # Bottom of fm, TVD
```

```
37          self.fm = fm                                # Unique number of fm
38          self.depth = self.bottom[-1]                # Total depth, TVD
39          self.fm_names = fm_names                    # Name of fm
40          self.thickness = thickness                  # Thickness of fm
41          #self.fm_rel = fm_rel                       # Integer,
42          #self.thickness_rel = thickness_rel         # Integer,
43          #self.depth_rel = depth_rel                 # Integer,
44
45      def __repr__(self):
46          return '{}'.format(self.name)
47
48  ## Empty lists
49  a = []                                              # Top of fms, TVD
50  b = []                                              # Bottom of fms, TVD
51  c = []                                              # Name of fms
52  d = []                                              # Unique number of fms
53  a_temp = []
54  b_temp = []
55  c_temp = []
56  d_temp = []
57  ref = []                                            # Name of wells
58  input_well = False
59
60  ######### Part 1: Pre-processing of input data
61  ## Retrieve input well
62  os.chdir('C:\\Users\MinhHoan\OneDrive      NTNU\Prosjekt og master\
63  Masteroppgaven\Python\Input')
64
65  if os.listdir() == []:
66      pass
67  else:
68      input_well = True
69      input_file = pd.read_excel('Input.xls', usecols=[0,1,2,5,3],
70                                 names=[0,1,2,3,4],skip_footer=8363)
71
72  for i in range(len(input_file)):
73      if i+1 not in range(len(input_file)):
74          a_temp.append(input_file[1][i])
75          b_temp.append(input_file[2][i])
76          c_temp.append(input_file[3][i])
77          d_temp.append(input_file[4][i])
78          a.append(a_temp)
79          b.append(b_temp)
80          c.append(c_temp)
81          d.append(d_temp)
82          a_temp = []
83          b_temp = []
84          c_temp = []
85          d_temp = []
86          ref.append(input_file[0][i])
87      elif input_file[1][i] <= input_file[1][i + 1]:
88          a_temp.append(input_file[1][i])
89          b_temp.append(input_file[2][i])
90          c_temp.append(input_file[3][i])
91          d_temp.append(input_file[4][i])
92
93  target_depth = b[0][-1]                             # Total depth, TVD
```

```
94
95  ## Retrieve the database that is available from NPD
96  os.chdir('C:\\Users\MinhHoan\OneDrive      NTNU\Prosjekt og master\
97  Masteroppgaven')
98
99  Data = pd.read_excel('Datasett.xls', usecols=[0,1,2,5,3],
100                       names=[0,1,2,3,4])
101 for i in range(len(Data)):
102     if i+1 not in range(len(Data)):
103         a.append(a_temp)
104         b.append(b_temp)
105         c.append(c_temp)
106         d.append(d_temp)
107         ref.append(Data[0][i])
108     elif Data[1][i] <= Data[1][i+1]:
109         a_temp.append(Data[1][i])
110         b_temp.append(Data[2][i])
111         c_temp.append(Data[3][i])
112         d_temp.append(Data[4][i])
113     else:
114         a_temp.append(Data[1][i])
115         b_temp.append(Data[2][i])
116         c_temp.append(Data[3][i])
117         d_temp.append(Data[4][i])
118         a.append(a_temp)
119         b.append(b_temp)
120         c.append(c_temp)
121         d.append(d_temp)
122         a_temp = []
123         b_temp = []
124         c_temp = []
125         d_temp = []
126         ref.append(Data[0][i])
127
128 ## Removing layers that are categorized as No formal name or
129 # Undifferentiated
130 i = 0
131 while i < len(d[0]):
132     if d[0][i] == 111 or i == 182:
133         a[0].pop(i)
134         b[0].pop(i)
135         c[0].pop(i)
136         d[0].pop(i)
137         i += 1
138     else:
139         i += 1
140
141 ## Generate objects based on the defined class, Well
142 Objects = []
143 for i in range(len(a)):
144     Objects.append(Well(top=a[i], bottom=b[i], name=ref[i],fm=d[i],
145                         index=i,fm_names=c[i]))
146
147 ######### PART 2: MACHINE LEARNING METHODS - KMEANS
148
149 ## Make input data ready for k-means and multidimensional scaling
150 from bin_matrix import bin_matrix
```

```
151 X_formation,X_thickness, X_depth, Objects = bin_matrix(Objects)
152
153 ## Call function cluster to prepare input data for k-means
154 from raw_input import cluster
155 num_clusters = 120
156 raw_input = cluster(X_thickness,X_depth,d)
157
158 ## k-means
159 kmeans_raw = kMeans(raw_input,n_clusters=num_clusters,return_n_iter=True)
160
161 ## Call function target_cluster to find cluster of which the input well
162 # lies in, the target cluster
163 from target_cluster import target_cluster
164 clusters, clust_target = target_cluster(kmeans_raw,num_clusters)
165
166 ## Lets examine the total TVD of all wells in the target cluster.
167 # We remove wells that are much deeper than the input well
168 if input_well == True:
169     depths = []
170     pool = []
171     for i in clusters[clust_target]:
172         depths.append(Objects[i].depth)
173         pool.append(Objects[i])
174 spekter = range(depths[0] - 500, depths[0] + 500)
175 j = 0
176 while j < len(depths):
177     if depths[j] not in spekter:
178         depths.pop(j)
179         throw = pool.pop(j)
180         j += 1
181     else:
182         j += 1
183
184 ######### PART 3: MACHINE LEARNING METHODS - NEAREST NEIGHBOR
185 ## Call function SD to get standard deviatons of fm layers
186 from standard_dev import SD
187 SD_thickness, SD_depth, mean_thickness, mean_depth = SD(X_thickness,
188                                                         X_depth,pool)
189
190 ## Call function scaling to get zero mean and unit variance over the
191 # distribution of each fm layer
192 from scaling import scaling
193 input_matrix = scaling(pool, X_thickness, X_depth, SD_thickness,
194                        SD_depth, mean_thickness, mean_depth)
195
196 ## Call function prompt_weights to
197 # enable engineer's ability to weight fm layers
198 from prompt_weights import prompt_weights
199 input_matrix = prompt_weights(pool,input_matrix)
200
201 ## Find the nearest neighbors of input well and rank them
202 # according to their distances
203 from mod_kNN import kNN
204 pool = kNN(input_matrix,pool)
205 def sort_func(Well):
206     return Well.nn
207 pool_final = sorted(pool, key=sort_func)
```

```
208
209 ## Use MDS to visualize data
210 dmatrix = pairwise_distances(raw_input)
211 mds = manifold.smacof(dissimilarities=dmatrix,n_components=3,
212                       metric=True,max_iter=3000,random_state=1,
213                       eps=1e-9,normalize=True)
214 # Plot - 3D space
215 fig = plt.figure()
216 ax = Axes3D(fig)
217 ax.scatter(mds[0][:, 0], mds[0][:, 1], mds[0][:, 2])
218 ax.set_xlabel('DIM1')
219 ax.set_ylabel('DIM2')
220 ax.set_zlabel('DIM3')
221 plt.show()
222 # Plot - 2D space
223 plt.scatter(mds[0][:, 0], mds[0][:, 1])
224 plt.xlabel('DIM1')
225 plt.ylabel('DIM2')
226 plt.show()
227
228 ## Print outpust of expert system
229 print('Target well: ', pool_final[0].name)
230 print('Well name', 'Nearest Neighbor Value')
231 for i in range(1,len(pool_final)):
232     print(pool_final[i].name, pool_final[i].nn)
233
234 ## Find nearest cluster by locating centroids
235 centroid_distances = pairwise_distances(kmeans_raw[0],kmeans_raw[0])
236 j = 10000000
237 count = 0
238 for i in range(len(centroid_distances[clust_target])):
239     if centroid_distances[clust_target,i] < j and \
240             centroid_distances[clust_target,i] > 0:
241         j = centroid_distances[clust_target,i]
242         count = i
243     else:
244         continue
245
246 print('Nearest centroid has the following wells:')
247 print('Well name')
248 for x in clusters[count]:
249     print(Objects[x].name)
250 print('Stress-1:', mds[1])
```

A.2 bin_matrix.py

```
######### Function bin_matrix #########
# A function that takes all objects as input to create three
# m x n matrices X_formation, X_thickness, X_depth
# m = number of wells in data set
# n = number of formations existing in NPD database, which is 212
# Each well (object) has a certain set of fm layers that make up
# the total geological profile. These layers have unique IDs defined
# by NPD. X_formation, X_thickness, and X_depth stores information
# about if the fm exists in object, thickness of fm, and depth (TVD)
# of top of fm respectively.
# Example; If well i has the layer Nordland with unique ID 113.
# X_formation gets a binary 1 in the ith row and 113th column.
# 0 is given if the fm does not exist. For X_thickness and X_depth,
# instead of binary 0/1, we store the information directly into the
# ith row and 113th column. The purpose, is to have a place where
# information is easily stored and traceable since we always have
# that the row index refer to a well, while the column index refers
# to a given fm layer.

# Parameters
# ----------
# Objects : list
#   The number of objects (wells) that is available in the current
#   database.

import numpy as np

def bin_matrix(Objects):
    dim = (len(Objects), 212)
    X_formation = np.zeros(dim)
    X_thickness = np.zeros(dim)
    X_depth = np.zeros(dim)
    temp = []
    for i in range(len(Objects)):
        for j in range(len(Objects[i].top)):
            X_formation[i][Objects[i].fm[j]] = 1
            layer = Objects[i].bottom[j] - Objects[i].top[j]
            X_thickness[i][Objects[i].fm[j]] = layer
            temp.append(layer)
            X_depth[i][Objects[i].fm[j]] = Objects[i].top[j]
        Objects[i].thickness = temp
        temp = []
    return X_formation, X_thickness, X_depth, Objects
```

A.3 raw_input.py

```
######### Function cluster #########
# Shall prepare the necessary input that is needed to
# perform k-means on the data that was originally taken
# as input. X_thickness, X_depth, and d is taken as input.
# The function uses d to locate the fm layers that exists
# in the input well. It then extracts all information from
# every well in the database in these exact layers, ignoring
# any other fm layers that a well may have.
# The output, raw_input is a m x 2n matrix that is ready for
# k-means. m is the number of wells in the database. n is the
# number of layers our input well has. We have 2n since we want
# to look at both fm thickness and depth of top of fm.

# Parameters
# ----------
# thickness : ndarray
#   A m x i matrix. m is the number of wells in the database.
#   i the number of fm layers that exists in the database.
#   It carries the information of thickness of each fm layer for
#   every well.
#
# depth : ndarray
#   A m x i matrix. m is the number of wells in the database.
#   i the number of fm layers that exists in the database.
#   It carries the information of TVD top of fm of each layer
#   for every well.
#
# fm : list
#   The lists holds the geological profile of each well in the
#   database, that is, what fm layers a well consists of.

def cluster(thickness,depth,fm):
    raw_input = []
    for i in range(len(thickness)):
        temp = []
        temp2 = []
        for j in fm[0]:
            temp.append(thickness[i,j])
            temp2.append(depth[i,j])
        temp.extend(temp2)
        raw_input.append(temp)
    return raw_input
```

A.4 target_cluster.py

```
######### Function target_cluster #########
# Function that takes the results gained in k-means as input.
# The output produced is the information of which cluster
# every well in the database is in. The function also returns
# the target cluster, which is the cluster where our input well
# lies in. The expert system is scriptet so that kmeans[1][0]
# will always be the input well.

# Parameters
# ----------
# kmeans : tuple
#   It is the result gained from performing k-means. Has the
#   information of the coordinates of all centroids
#   and which cluster each well belongs to.
#
# num_clusters : int, optional, default: None


def target_cluster(kmeans,num_clusters):
    clusters = []
    clust_target = kmeans[1][0]
    for i in range(num_clusters):
        cluster = []
        for j in range(len(kmeans[1])):
            if kmeans[1][j] == i :
                cluster.append(j)
            else:
                pass
        clusters.append(cluster)
    return clusters, clust_target
```

A.5 standard_dev.py

```
######### Function SD #########
## Calculates the standard deviation in thickness and TVD of
# top of fm for each layer. We only evaluate fm layers that the
# target well has in its geological profile.
# Example: When looking at a layer, say, Nordland. It will take
# the information for thickness in that layer, for all wells.
# Calculate the mean thickness of Nordland. Then, calculate the
# standard deviation in Nordland. This is done for all layers
# that exists in the input well. The same is done with TVD of
# top fm. The standard deviations are later used to standardize
# the data (scale).

# Parameters
# ----------
# thickness : ndarray
#   A m x i matrix. m is the number of wells in the database.
#   i the number of fm layers that exists in the database.
#   It carries the information of thickness of each fm layer for
#   every well.
#
# depth : ndarray
#   A m x i matrix. m is the number of wells in the database.
#   i the number of fm layers that exists in the database.
#   It carries the information of TVD top of fm of each layer
#   for every well.
#
# pool : list
#   The pool of wells. It is the wells lying within the target cluster
#   as objects.


import numpy as np

def SD(thickness,depth,pool):
    n = len(pool)
    count = 0
    mean_thickness = []
    mean_depth = []
    SD_thickness = []
    SD_depth = []
# Calculate mean values
    for i in pool[0].fm:
        temp_thickness = []
        temp_depth = []
        for j in range(len(pool)):
            temp_thickness.append(thickness[pool[j].index,i])
            temp_depth.append(depth[pool[j].index,i])
        mean_thickness.append(sum(temp_thickness)/ n)
        mean_depth.append(sum(temp_depth) / n)
# Calculate standard deviations
    for i in pool[0].fm:
        denominator_thickness = []
        denominator_depth = []
        for j in range(len(pool)):
                denominator_thickness.append((thickness[pool[j].index,i]-
```

```
56                                                  mean_thickness[count])**2)
57                  denominator_depth.append((depth[pool[j].index,i]-
58                                                  mean_depth[count])**2)
59          count +=1
60          SD_thickness.append(np.sqrt(sum(denominator_thickness) / (n-1)))
61          SD_depth.append(np.sqrt(sum(denominator_depth) / (n-1)))
62      return SD_thickness, SD_depth, mean_thickness, mean_depth
```

A.6 scaling.py

```
######### Function scaling #########
# Shall standardize the well data of the wells within
# the target cluster. By having zero mean and unit variance, the
# the subtle valued thickness in fm are enhances while at the same
# time, the dominating values of depth of top fm are dampened.
# This makes it easier to assess the true deviation when we look for
# relevant wells. It takes the standard deviation of each layer, one
# for thickness and one for depth of top fm as input.
# It produces a input_matrix that has same length as pool.
# The input_matrix is used in kNN to rank the wells lying in the
# target cluster after least to most deviation.

# Parameters
# ----------
# pool : list
#   The pool of wells. It is the wells lying within the target cluster
#   as objects.
#
# thickness : ndarray
#   A m x i matrix. m is the number of wells in the database.
#   i the number of fm layers that exists in the database.
#   It carries the information of thickness of each fm layer for
#   every well.
#
# depth : ndarray
#   A m x i matrix. m is the number of wells in the database.
#   i the number of fm layers that exists in the database.
#   It carries the information of TVD top of fm of each layer
#   for every well.
#
# SD_thickness : list
#   A list that holds the standard deviation of thickness for each layer
#   that exists in the input well.
#
# SD_depth : list
#   A list that holds the standard deviation of TVD top of fm for each
#   layer that exists in the input well.
#
# mean_thickness : list
#   A list that holds the mean of thickness for each layer
#   that exists in the input well.
#
# mean_depth : list
#   A list that holds the mean of TVD top of fm for each layer
#   that exists in the input well.

from sklearn.metrics.pairwise import pairwise_distances
import numpy as np

def scaling(pool,thickness, depth, SD_thickness, SD_depth,
            mean_thickness, mean_depth):
    input_matrix = []
    for i in range(len(pool)):
        row_temp = []
        row_temp2 = []
```

```
56          count = 0
57          for j in pool[0].fm:
58              scale_thickness = (thickness[pool[i].index,j] -
59                                 mean_thickness[count]) / SD_thickness[count]
60              scale_depth = (depth[pool[i].index,j] - mean_depth[count]) \
61                            / SD_depth[count]
62              row_temp.append(scale_thickness)
63              row_temp2.append(scale_depth)
64              count += 1
65          row_temp.extend(row_temp2)
66          input_matrix.append(row_temp)
67      input_matrix = np.asarray(input_matrix)
68      return input_matrix
```

A.7 prompt_weights.py

```
######### Function prompt_weights #########
# A function that enables an engineer to weight certain fm layers over
# the others. Weighting is triggered by binary 0/1 for NO/YES.
# We have two weighting arrays: W and w.
# W weights how we should prioritize thickness in fm layers
# in relation to the offset in top of fm.
# w weights how we should prioritize the different fm layers
# in relation to each other. For example, if there exists more
# complicated fm layers (drilling wise) we should weight it
# higher than others to penalize any deviation.

# Parameters
# ----------
# input_matrix : ndarray
#   A i x 2n matrix. Where i is the number of wells that lies within
#   the target cluster. n is the number of fm layers that the input
#   well consists of.
#
# pool : list
#   The pool of wells. It is the wells lying within the target cluster
#   as objects.

import numpy as np

def prompt_weights(pool,input_matrix):
    print('Your target well has the following layers:')
    for i in pool[0].fm_names:
        print(i)
    print('There are a total of', len(pool[0].fm_names), 'layers.')
    print('Weighting: Layers that are weighted with high values will '
          'be prioritized. Layers are weighted in relation to each '
          'other.')
    prompt = int(input('If you wish to weight layers please select: 1.'
                       'If not, choose default by selecting: 0. '
                       'Type your selection: '))
    weights_thickness = []
    weights_depth = []
    if prompt == 1:
        # Assigning weights to layers by prompt
        weighting = list(input('Weight layers by same ordering as '
                               'given above. No spacing.'))
        # Check that weighting dimensions are correct
        while True:
            if len(pool[0].fm_names) == len(weighting):
                W = np.ones(2,dtype=int)
                w = np.asarray([int(i) for i in weighting])
                for i in range(len(w)):
                    weights_thickness.append(w[i]*W[0])
                    weights_depth.append(w[i]*W[1])
                weights_thickness.extend(weights_depth)
                weights = weights_thickness
                for i in range(len(pool)):
                    for j in range(len(input_matrix[0])):
                        input_matrix[i,j] = weights[j] * input_matrix[i,j]
                break
```

```
56                 else:
57                     weighting = list(input('Number of weights do not match. '
58                                            'No spacing. Try again: '))
59         else:
60             W = np.array([1,1])
61             w = np.ones(len(input_matrix[0]),dtype=int)
62             for i in range(int(len(w)/2)):
63                 weights_thickness.append(w[i] * W[0])
64                 weights_depth.append(w[i] * W[1])
65             weights_thickness.extend(weights_depth)
66             weights = weights_thickness
67             for i in range(len(pool)):
68                 for j in range(len(input_matrix[0])):
69                     input_matrix[i, j] = weights[j] * input_matrix[i, j]
70         return input_matrix
```

A.8 mod_kNN.py

```
######### Function kNN #########
## A modified nearest neighbor that shall find the distances
# from input well to every other well lying in the target cluster.
# The nearest neighbor algorithm is not explicitly used but rather,
# the written function is based on the same methodology for the
# purpose of ordering the most relevant wells.

# Parameters
# ----------
# input_matrix : ndarray
#   A i x 2n matrix. Where i is the number of wells that lies within
#   the target cluster. n is the number of fm layers that the input
#   well consists of.
#
# pool : list
#   The pool of wells. It is the wells lying within the target cluster
#   as objects.


from sklearn.metrics.pairwise import pairwise_distances
def kNN(input_matrix,pool):
    neighbor = pairwise_distances(input_matrix, input_matrix)
    # Write to object's attribute
    for i in range(len(pool)):
        pool[i].nn = neighbor[i,0]
    return pool
```