DEGREE PROJECT IN MATHEMATICS,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Text Feature Mining Using Pre-trained Word Embeddings

HENRIK SJÖKVIST

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

Text Feature Mining Using Pre-trained Word Embeddings

HENRIK SJÖKVIST

Degree Projects in Financial Mathematics (30 ECTS credits)
Degree Programme in Industrial Engineering and Management
KTH Royal Institute of Technology year 2018
Supervisor at Handelsbanken: Richard Henricsson
Supervisor at KTH: Henrik Hult
Examiner at KTH: Henrik Hult

TRITA-SCI-GRU 2018:167
MAT-E 2018:28

Royal Institute of Technology
School of Engineering Sciences
KTH SCI
SE-100 44 Stockholm, Sweden
URL: www.kth.se/sci

Abstract

This thesis explores a machine learning task where the data contains not only numerical features but also free-text features. In order to employ a supervised classifier and make predictions, the free-text features must be converted into numerical features. In this thesis, an algorithm is developed to perform that conversion.

The algorithm uses a pre-trained word embedding model which maps each word to a vector. The vectors for multiple word embeddings belonging to the same sentence are then combined to form a single sentence embedding. The sentence embeddings for the whole dataset are clustered to identify distinct groups of free-text strings. The cluster labels are output as the numerical features.

The algorithm is applied on a specific case concerning operational risk control in banking. The data consists of modifications made to trades in financial instruments. Each such modification comes with a short text string which documents the modification, a *trader comment*. Converting these strings to numerical trader comment features is the objective of the case study.

A classifier is trained and used as an evaluation tool for the trader comment features. The performance of the classifier is measured with and without the trader comment feature. Multiple models for generating the features are evaluated. All models lead to an improvement in classification rate over not using a trader comment feature. The best performance is achieved with a model where the sentence embeddings are generated using the SIF weighting scheme and then clustered using the DBSCAN algorithm.

Keywords — Word embeddings, Feature engineering, Unsupervised learning, Deep learning, fastText, Operational risk

Sammanfattning

Detta examensarbete behandlar ett maskininlärningsproblem där data innehåller fritext utöver numeriska attribut. För att kunna använda all data för övervakat lärande måste fritexten omvandlas till numeriska värden. En algoritm utvecklas i detta arbete för att utföra den omvandlingen.

Algoritmen använder färdigtränade ordvektormodeller som omvandlar varje ord till en vektor. Vektorerna för flera ord i samma mening kan sedan kombineras till en meningsvektor. Meningsvektorerna i hela datamängden klustras sedan för att identifiera grupper av liknande textsträngar. Algoritmens utdata är varje datapunkts klustertillhörighet.

Algoritmen appliceras på ett specifikt fall som berör operativ risk inom banksektorn. Data består av modifikationer av finansiella transaktioner. Varje sådan modifikation har en tillhörande textkommentar som beskriver modifikationen, en *handlarkommentar*. Att omvandla dessa kommentarer till numeriska värden är målet med fallstudien.

En klassificeringsmodell tränas och används för att utvärdera de numeriska värdena från handlarkommentarerna. Klassificeringssäkerheten mäts med och utan de numeriska värdena. Olika modeller för att generera värdena från handlarkommentarerna utvärderas. Samtliga modeller leder till en förbättring i klassificering över att inte använda handlarkommentarerna. Den bästa klassificeringssäkerheten uppnås med en modell där meningsvektorerna genereras med hjälp av SIF-viktning och sedan klustras med hjälp av DBSCAN-algoritmen.

Nyckelord — Ordvektorer, Attributgenerering, Oövervakat lärande, Djupinlärning, fastText, Operativ risk

Acknowledgements

I would like to thank...

Professor Henrik Hult, my thesis supervisor at the Department of Mathematics at KTH Royal Institute of Technology for his guidance, support and interesting discussions throughout the thesis project.

Richard Henricsson, Ph.D and the rest of the Model Validation & Quantitative Analysis team at Handelsbanken Capital Markets for commissioning this exciting project and providing the data and resources required to conduct it.

Bob Dylan for inspiration.

My friends, family and loved ones for their support and encouragement during my years at KTH.

Stockholm, May 2018

Henrik Sjökvist

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning and Feature Engineering

The fields of machine learning and artificial intelligence have seen a recent surge of commercial interest. The continuing development of more powerful and less expensive computer hardware, breakthroughs in training algorithms and the ever-increasing availability of data have enabled widespread deployment of machine learning models. What was once regarded as little more than an arcane field of puzzle solving in computer science is now finding its way into just about every industry.

Machine learning enables computers to perform tasks such as pattern recognition, forecasting, classification and anomaly detection. Fundamentally, these are all examples of tasks where an agent attempts to extract meaningful information from large sets of data. A human can learn to perform such tasks with experience. A doctor can learn to recognize patients' symptoms and diagnose them correctly with a reasonable probability of success. A financial analyst can take data and use mathematics and their experience to determine whether or not to invest in a company. Provided enough data and a suitable model, a machine could also learn to perform or assist with such tasks.

In machine learning, a model is typically trained on a large set of historical data. Each data point consists of a number of observations of various characteristics, known as *features*. The machine learning algorithm then trains the model by taking these features as input. For instance, consider the problem of predicting housing prices on the real estate market. A machine could learn to predict prices by processing data of historical real estate sales and training a model on that data. Suitable features to include in the model could for instance be house size, number of bedrooms, age of the house and average price of houses in the same area. Typically, a perfect dataset of relevant features is not readily available. Instead, a data scientist can use domain knowledge and intuition to craft features. This process is commonly known as *feature engineering*.

Feature engineering can be tedious and highly technical. The time spent on crafting high-quality features can easily exceed the time spent implementing and training the actual machine learning model. The success of the model often hinges on the underlying quality of the data and the feature engineering. There exist scenarios where algorithms can effectively extract and select features automatically, however this is not the case generally. In general a human is still required to help the computer with data preprocessing and feature engineering.

Consider again the problem of predicting housing prices. Notice that all the features (size, number of bedrooms, age and price of comparables) are numerical. Most

machine learning algorithms accept only numerical features as input. Algorithms that accept other forms of data, such as text or images, have some internal way of representing those data types numerically. There exist many types of interesting characteristics that cannot be initially represented numerically in a model without some form of numerical representation schema. In the housing prediction problem, one might for instance consider it a good idea to provide the model with information regarding whether a home is a detached house or an apartment. The characteristic of a real estate object being a detached house is certainly not numerical. This is known as a *categorical feature*. Conveniently, categorical features can easily be represented as numerical values by a simple encoding. Let the feature have the value 1 if the object is a detached house and 0 if the object is an apartment.

Representing categorical features numerically is simple. Consider a more challenging case; assume that for each real estate object in the housing dataset there is a short text description of the house. This could be the property description from the real estate ad. The text is a free-text and is not categorical since the writer is free to formulate the text in any way rather than choosing from a predetermined list of allowed texts. How to represent such a free-text numerically? This is a much more complex problem and is the central topic of this thesis.

The purpose of this thesis is to develop and evaluate models for converting free text strings into numerical values that can be used as features. It will be shown that a good approach for this is to represent the text strings as real-valued vectors, known as *word embeddings*. There exist advanced neural network models that can efficiently train such vectors. Furthermore, it will be shown that for the purpose of feature generation one can make use of pre-trained models which have been trained on enormous datasets. This eliminates the need for training new word embedding models in order to generate features.

Operational Risk Management

This thesis has been commissioned by Svenska Handelsbanken (SHB) and has been carried out at the Model Validation & Quantitative Analysis department of the bank. The techniques explored in this thesis could be applied to any machine learning problem of similar nature in any industry. However, for this thesis a very specific application is considered and studied. The problem considered in this thesis is similar to the example of predicting housing prices, but rather than studying real estate, the case studied in this thesis concerns operational risk management.

The Basel III regulatory framework defines operational risk as *"the risk of loss resulting from inadequate or failed internal processes, people and systems or from external events"* [22]. This is a broad definition. Essentially, operational risk covers most risks associated with human error, failure of operational processes and external factors which are not linked to any other main category of risks such as financial or market risk.

Operational risks are of paramount importance to financial institutions. Unlike financial risk and market risk, operational risk is difficult to detect and hedge against. This is because operational risks often arise from human error. Such errors could be with or without malicious intent. Since large banks like Handelsbanken deal with very large transactions, simple human errors could prove extremely costly.

There are many examples of large banks which have taken heavy losses due to factors relating to operational risk.

- In 1995 a derivatives trader at Barings Bank, the oldest merchant bank in the United Kingdom, made a series of unauthorized trades. When finally discovered, the losses due to these trades had reached $1.4 billion and the 233 year old bank was declared insolvent and later acquired for £1 [29].

- In 2005, a trader at Mizuho Securities was instructed to sell one share of a particular stock at ¥610,000. Instead, the trader issued an order to sell 610,000 shares at ¥1 a share. Despite discovering the mistake within 85 seconds of the order, the error ended up costing the firm $255 million [5].

Clearly, it is very much in the risk control department's interest to have the ability to identify anomalous behavior in the trading department.

The data studied in this thesis specifically concerns modifications made to existing trades. When a trader at Handelsbanken modifies an existing trade s/he is required to enter a free-text comment into the trading platform explaining and documenting the modification. The comments are typically very brief and full of nomenclature. Nevertheless it is reasonable to expect that these comments contain information relevant for identifying and predicting suspicious behavior. Hundreds of thousands of such modifications of trades are made annually resulting in a dataset far too large to analyze manually for anomalies. Thus machine learning is a natural approach to this problem. SHB had models in place prior to the work conducted during this thesis, but those did not make use of the free-text comments in the data. Thus, the objective of this thesis is twofold; the first goal is to explore techniques for converting the free-text trader comments into meaningful numerical features, the second goal is to test the existing anomaly detection models with and without the trader comment features to see if the performance improves when the information from the comments is incorporated.

Research Questions

- *How can word embeddings be employed to represent short text strings as numerical features with minimal information loss?*

- *What is the predictive impact of including such features?*

Delimitations

- The text studied in this thesis is in Swedish, however the models used are applicable for any language.

- The data used contains modifications of trades in exchange rate derivatives (FX) only. The reason for this is that the vast majority of labeled data available was for FX trades. Again, the models used are applicable for any type of trade and more generally for non-financial applications as well.

Thesis Outline

Chapter 2 provides a brief review of machine learning and natural language processing tools relevant for this thesis. Chapter 3 covers relevant literature and positions the thesis. Chapter 4 describes the dataset used in this study. Chapter 5 provides

a detailed description of the algorithm developed for this thesis and its implementation. Chapter 6 presents results of the algorithm tests. Finally, Chapter 7 discusses the results, provides concluding analysis and suggests future research directions.

Chapter 2

Theory

This Chapter provides the reader a review of prerequisite theory relevant to understanding the methods used in the thesis. Readers familiar with the topics covered here can skip ahead to Chapter 3.

Section 2.1 covers the basics of machine learning, Section 2.2 introduces the reader to the various machine learning models used in this thesis, Section 2.3 covers relevant NLP models and Section 2.4 describes some probabilistic modeling.

2.1 Machine Learning Preliminaries

Machine learning is a subfield within computer science where computers are trained to perform certain tasks without the need for an explicit set of rules to follow. In order to do this, the computer requires a *model* and a set of *data*.

Fundamentally, the machine learning model is a framework consisting of the following items:

- A mathematical description of the machine learning task at hand.
- A training algorithm for finding a solution to the task
- A set of *hyperparameters*. These are parameters of the model which must be specified prior to learning a solution.

Typically, the hyperparameters specify the way in which a solution is learned and/or the structure of the solution. For instance, this could be the number of iterations to run a training algorithm. The process of selecting the optimal hyperparameter values, known as *hyperparameter tuning* is usually not trivial. An approach to tuning hyperparameters is to specify a range of hyperparameter values to evaluate and then choose the set of values which perform the best according to some evaluation criteria.

In practice, standard machine learning models are usually implemented through open-source code libraries. Thus the user is only required to choose a model and specify the (range of) hyperparameters.

Crucial to success in a machine learning project is the availability of data. Each data point is a collection of observations of random variables, each representing a characteristic of the subject. These random variables are known as data *features*. Assuming all features can be represented numerically, one can conveniently represent

a dataset of n data points each with j features as an $n \times j$ matrix $\mathbf{X}$.

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{np} \end{bmatrix}$$

Each row of the matrix $\mathbf{X}$ is a data point, denoted as the feature vector $x_i = (x_{i,1}, x_{i,2}, \dots, x_{i,p})^T$. The p-dimensional vector space spanned by the vectors x_i, $i = 1, 2, \dots, n$ is known as the *feature space* of the dataset.

The machine learning model parses the data and searches for patterns. Many factors influence the success of the machine learning model's attempt to model patterns in the data, mainly:

- **Data quality**. Noise in the data will distort the solution and make finding true patterns difficult. There may also be limitations in the way data has been observed and recorded. The machine learning model can only work with the data features which have been recorded. Moreover, the data may be fundamentally uncorrelated with the phenomenon being studied. The performance of the machine learning model stands or falls with the data quality. Applying all the best practices in modeling will not help if there is no pattern in the data to model.

- **Data quantity**. Machine learning models typically require large amounts of data to train on in order to perform well. Many models rely directly on the law of large numbers to achieve asymptotic convergence to optimal solutions with increasing data quantity. Certain models require more data than others.

- **Choice of model**. Machine learning models are designed to perform a certain set of tasks well. A model for product suggestion is probably unsuitable for speech recognition. Selecting an appropriate model is crucial.

- **Computational resources**. Training machine learning models can be very computationally strenuous. Some models are unfeasible to train efficiently on a standard personal computer. Access to powerful hardware can be a crucial requirement for success in a machine learning project.

  Computers use GPU's to perform the fast matrix calculations required to render complex computer graphics. Such operations are similar to the ones required to train certain machine learning models. Thus, using GPU's to train models can massively reduce training time [25].

  The emergence of cloud computing has opened up the possibility for users to run their code on virtual machines with third party hardware. This enables individual users the access to large amounts of computing power on demand.

Fundamentally, machine learning models can be divided into two categories: *supervised learning* and *unsupervised learning*.[1] They are used for different tasks and require different types of data.

[1] Sometimes *reinforcement learning* is included as a third main category of machine learning models.

Supervised Learning

In supervised learning, one of the data features is designated as a *response variable*. The task in supervised learning settings is then to create a model capable of predicting the response y_i, based on the input features x_i, $i = 1, 2, \ldots, n$. If the y_i are quantitative, such as in the case where y_i represents a real estate price to predict, the learning task is referred to as *regression*. If the y_i are categorical, such as in the case of predicting whether or not a patient has a particular disease, the learning task is referred to as $classification$. Moreover, in the case of classification, the response y_i is referred to as the *label* of that data point.

In supervised learning, one typically splits the data into two subsets. One is referred to as the *training data*. Like the name implies, this data is used to train a machine learning model. Given a set of data points with their associated responses $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ a supervised learning model attempts to find patterns in the features which can be used to predict the responses. The other subset is known as the *test data*. The trained model is tested on this data to evaluate performance. Since the model hasn't seen the test data during training, this makes it possible to detect if the model has been *overfit* to patterns only present in the training data.

When a model has been trained and evaluated, it can be used to predict the responses of new data. This allows the model to make predictions about responses which have not been recorded yet.

Unsupervised Learning

In unsupervised learning, the data consists of only observed features and no response. Thus, unsupervised learning problems do not concern prediction, since there is nothing to predict. Rather, unsupervised learning models focus solely on finding geometric structures in the data. The task of finding groups of data points which are geometrically proximate in the feature space is known as *clustering*. For instance, clustering can be used to identify segments in a customer base, i.e. clusters of customers who are similar.

Since predictions cannot be evaluated when data is unlabeled, there is no need to allocate some data as test data. Rather, the model can be trained on the full dataset.

2.2 Machine Learning Models

This thesis project employs both supervised and unsupervised techniques. This section covers the various machine learning models used.

k-Means Clustering

k-means clustering is a unsupervised model which partitions a dataset into k clusters. Each data point thus belongs to exactly one cluster. Consider a dataset $\mathbf{X}$ consisting of n data points. Let $C_1, C_2, \ldots, C_k$ be a partition of the data index set $\{1, 2, \ldots, n\}$. These sets represent the k clusters, i.e. if $i \in C_j$ then the data point x_i is contained in cluster j.

The objective of k-means clustering is to find the partition which best describes the data. But by what criterion should one measure the fit of the clustering? The standard approach is to find the cluster partition such that the total within-cluster

squared Euclidean variation is minimized [12]. I.e. for a given partition, the objective function is the squared Euclidean distance $\|x_i - x_{i'}\|^2$ between all data points belonging to the same cluster, summed over all clusters. In other words, k-means clustering is defined by the optimization problem:

$$\underset{C_1,\ldots,C_k}{\text{minimize}} \quad \sum_{j=1}^{k} \frac{1}{|C_j|} \sum_{i,i' \in C_k} \|x_i - x_{i'}\|^2$$

Finding a global optimal solution to the problem above is NP-hard [1]. The number of possible ways to partition a set of n data points into k clusters is a Stirling number of the second kind, $S(n,k)$, where

$$S(n,k) = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{n}{k}$$

This number explodes as n or k increase. Thus, a locally optimal solution to k-means clustering is instead found through a heuristic algorithm.

Crucial to defining the clusters are the cluster *centroids*. The centroid c_j of cluster j is the mean point of all the data points in the cluster, i.e.

$$c_j = \frac{1}{|C_j|} \sum_{i \in C_j} x_i$$

The centroid can be thought of as the center of mass of the cluster. Below, a heuristic algorithm for finding local optima to the k-means problem is presented.

Algorithm 1: k-means clustering heuristic algorithm

```
assign unique initial values for k centroids c_1, ..., c_k;
do
    for each data point x_i do
        assign x_i to the cluster whose centroid is closest;
    end
    for each cluster C_j do
        update centroid values c_j = (1/|C_j|) Σ_{i∈C_j} x_i;
    end
while centroid values changed from previous iteration;
```

This algorithm converges in $\mathcal{O}(nkpi)$ where i is the number of iterations needed until convergence. It has been shown in [27] that in the worst case, the number of iterations is $i = 2^{\Omega(n)}$. This means that the algorithm is superpolynomial in the worst case if run until convergence. In practice however, if the data has cluster structures the algorithm converges very quickly [4]. In the case of slow convergence, early stopping criteria can be employed to significantly reduce run time with little performance loss [24].

DBSCAN

Another commonly used unsupervised clustering algorithm is *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN). Whereas in k-means a fixed number of clusters are generated based on the positions of centroids, in the DBSCAN

algorithm the number of clusters is dynamic and decided by the structure of the data and some hyperparameters.

Again, consider the problem of clustering a dataset $\mathbf{X}$ of n data points. The DBSCAN algorithm takes two parameters: ε and minPts. Clusters are formed in the following way; for a data point x_i, count the number of data points within a radius of ε from x_i, this region is known as the ε-neighborhood of the data point. If the number of data points in the ε-neighborhood exceeds minPts, a new cluster is formed. If minPts is not exceeded, the data point is labeled as noise, i.e. not part of a cluster. This highlights a critical difference between DBSCAN and k-means, in DBSCAN not all data points are required to belong to a cluster, they can instead be labeled as noisy outliers. Furthermore, the number of clusters is not an input parameter but rather learned by the algorithm itself. This is an attractive property since a suitable number of clusters to look for is typically not known and can be hard to discover in high-dimensional data. However, ε and minPts must still be tuned to the data for good performance.

In Algorithm 2, the DBSCAN algorithm in pseudocode is presented. The algorithm parses through all data points and first checks that the data point has not yet been visited and labeled. If not, then a function `findNeighbors()` is called which returns a list N of all data points within the ε-neighborhood of the data point in question. If the number of elements in the list of neighbors is less than minPts then that data point is labeled as noise since it is not found in a dense region. Noisy data points are labeled -1. If the number of neighbors is not less than minPts then a new cluster is created with a unique identifier.

Next, the algorithm attempts to expand the cluster. This is done by parsing through the list of neighbors N to see if any of those data points are close to other dense groups of data points. For each data point in N, first check if the data point has previously been labeled as noise. This is a possibility since a data point may not itself have enough neighbors to start a new cluster but could be a neighbor of another data point which does fulfill that requirement. Thus if a neighbor has been labeled as noise, change the label to the cluster label. Then check if the neighbor has been labeled as belonging to any cluster, if so then there is nothing to expand into in this iteration of the loop so continue to the next. This includes the points which are relabeled from noise to the current class label since if they have previously been labeled as noise their ε-neighborhoods have already been checked and found to contain too few elements. If the data point was unlabeled then give it the label of the current cluster. Then call `findNeighbors()` again to find the list of neighbors M of this data point. If the size of M also exceeds minPts, then merge N and M by setting N to be the union of N and M.

In Figure 2.2 the expansion process of the DBSCAN algorithm is shown. In the figure, the algorithm is able to add two data points which were not part of the original ε-neighborhood by finding an intersecting dense ε-neighborhood which includes them.

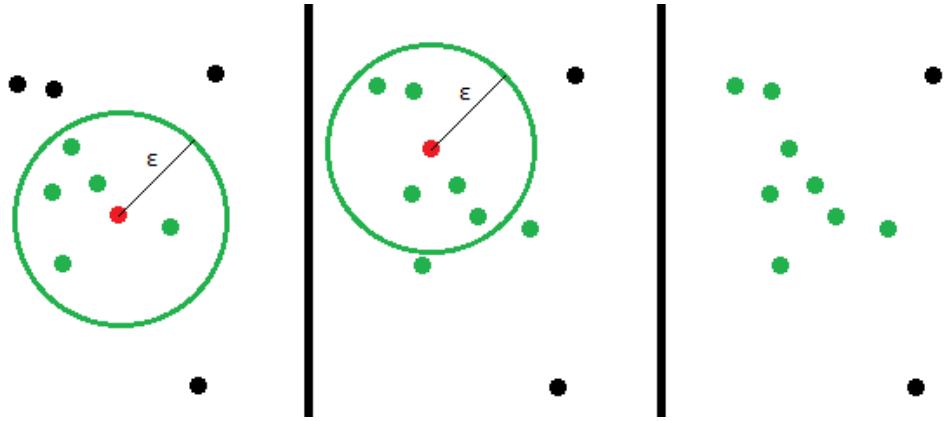


Figure 2.1: The DBSCAN algorithm expanding a cluster

The output of the algorithm is a cluster label for each data point in the dataset. If the label is a natural number, then the associated data point belongs to the cluster corresponding to that label. If the label is -1 then the data point was a remote outlier and has been labeled as noise.

Algorithm 2: DBSCAN algorithm

```
1  C=0;
2  for each data point x_i do
3  |  if x_i.label ≠ Null then
4  |  |  continue //data point has already been visited
5  |  end
6  |  N=findNeighbors(x_i, ε);
7  |  if |N| < minPts then
8  |  |  x_i.label = -1;
9  |  |  continue //classify as noise
10 |  end
11 |  C++;
12 |  x_i.label = C;
13 |  for each neighbor n_j in N do
14 |  |  if n_j.label == -1 then
15 |  |  |  n_j.label = C;
16 |  |  end
17 |  |  if n_j.label ≠ Null then
18 |  |  |  continue
19 |  |  end
20 |  |  n_j.label = C;
21 |  |  M = findNeighbors(n_j, ε);
22 |  |  if |M| ≥ minPts then
23 |  |  |  N = N ∪ M;
24 |  |  end
25 |  end
26 end
```

Given that the ε-neighborhoods typically are small compared to the whole dataset, the average run time complexity of the function `findNeighbors()` which finds all data points in an ε-neighborhood is $\mathcal{O}(\log n)$. The dataset has n elements and `findNeighbors()` is called at most once for each element. Thus, the average run

time complexity of DBSCAN is $\mathcal{O}(n \log n)$ [8].

Beyond the fact that DBSCAN does not require specifying the number of clusters a priori and that its notion of noise makes it robust to extreme outliers, the algorithm also comes with the advantage of being able to find arbitrarily shaped non-linear clusters. Unlike the linearly separable Voronoi cells created by the k-means algorithm, DBSCAN can find clusters of any shape so long as they are connected by a common dense region of data points.

Deep Belief Network

A Deep Belief Network (DBN) is a type of deep neural network which can be used for both unsupervised and supervised learning tasks. In this thesis, a DBN will be used as a supervised classifier. However, the main focus of the thesis is on feature generation for the DBN and not the DBN itself. The reader can choose to view the DBN in this thesis as essentially a black box algorithm replaceable by any other classifier. As such, this section will only give a brief introduction to DBN's. For a more thorough description of DBN's and the vast and interesting topic of artificial neural networks, see for example [10, 15].

In essence, a deep belief network is a created by "stacking" several smaller neural network models known as Restricted Boltzmann Machines (RBM). An RBM consists of a visible and a hidden layer where the hidden layer is trained to model the probability distribution of the visible inputs. The *restricted* property of a restricted Boltzmann machine comes from the fact that no two neurons within the same layer may be connected. In a DBN, the hidden layer of an RBM acts as the visible layer for the next DBN in the stack. The RBM's act as unsupervised feature detectors.

Training a DBN consists of two steps. The first step is an unsupervised learning task where the DBN learns to extract features from the data. In the second step labels are introduced to perform supervised learning for the purpose of classification. A critical property of DBN's is that each RBM can be isolated and trained greedily [11].

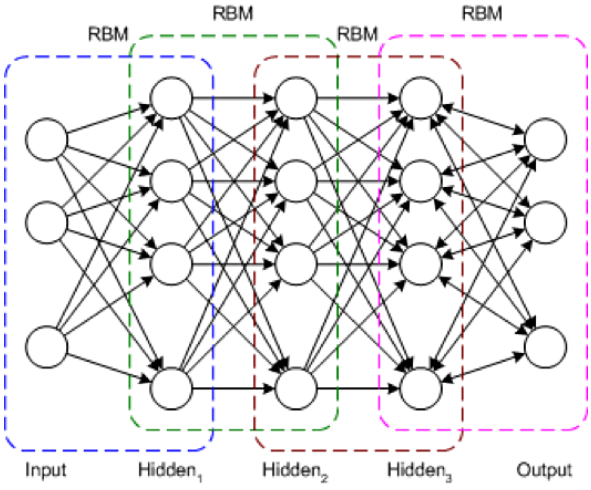


Figure 2.2: Architecture of a deep belief network. Image source:[2]

[2]https://www.ibm.com/developerworks/library/cc-machine-learning-deep-learning-architectures/index.html

2.3 Natural Language Processing

Natural Language Processing is the field in computer science that studies techniques which enable computers to process human language. It is closely related to machine learning, and machine learning techniques are frequently employed to improve the computer's ability to understand language.

Word Representations

This study mainly concerns the usage of numerical *word representations*, typically in the form of real-valued vectors in high-dimensional space. For this reason, such word representations are commonly referred to as *word vectors*. Word representation models are often trained with machine learning algorithms on datasets containing text. Such a dataset is referred to as a text *corpus*.

Bag-of-Words

The simplest word representation model is the *Bag-of-Words* model. In the Bag-of-Words model, the number of dimensions is equal to the number of unique words, with each dimension corresponding to a unique word. For instance, consider the sentence: `The quick brown fox jumps over the lazy dog`. The sentence contains eight unique words: `'the'`, `'quick'`, `'brown'`, `'fox'`, `'jumps'`, `'over'`, `'lazy'`, `'dog'`. Thus, if one wants to create a Bag-of-Words using that sentence as the training corpus, 8-dimensional word vectors are needed. Typically, a word vector in the Bag-of-Words model is a one-hot encoded vector. The word vector for the word `'fox'` thus becomes $v_{fox} = [0, 0, 0, 1, 0, 0, 0, 0]$. We can get a *sentence vector* for `The quick brown fox jumps over the lazy dog` by simply adding together the vectors for each word. This gives us the sentence vector $[2, 1, 1, 1, 1, 1, 1, 1]$

For practical applications, one needs a corpus much larger than just one sentence. As the size of the Bag-of-Words vectors grows with the number of unique words, one quickly realizes that storing one-hot encoded vectors in memory will be impractical. Instead one can use a hashing function and map the words to indices in a hash table. This makes the Bag-of-Words model more scalable with corpus size.

tf-idf

An obvious limitation of the Bag-of-Words model where each element corresponds to the frequency of the occurrence of a particular word is that the model assigns equal importance to each word in a text. Thus, a simple extension would be to weight each word by some measure of the importance of that word to a particular text. This is exactly the reasoning behind the *tf-idf score*. The *tf-idf* score of a word j in a text d of the corpus D is a metric computed as the product of two other metrics:

- The term frequency $\text{tf}(j, d) = |\{i \in d : i = j\}|$, i.e. the word count of j in d.
- The inverse document frequency $\text{idf}(j, d, D) = \log\left(\frac{|D|}{|\{d \in D : j \in d\}|}\right)$. The numerator in the logarithm is the cardinality of D, i.e. the number of texts in the corpus. The denominator is the number of texts in the corpus which contain the word j.

Multiplying these gives the *tf-idf* score:

$$\text{tf-idf}(j, d, D) = \text{tf}(j, d) \cdot \text{idf}(j, d, D)$$

If a word occurs many times in a particular text it will achieve a high term frequency for that text. A word which appears in a particular text but not in many other texts in the corpus will achieve a high inverse document frequency for that text. Thus, the intuition behind the *tf-idf* score is that if a rare word appears many times in a text, then that word is of great importance to that text. A word representation model would thus be to let each element in the word vectors correspond to the *tf-idf* score of the word.

Word Embeddings

Even after weighting the words by a metric like the *tf-idf* score, the Bag-of-Words model is still simple and has severe limitations. Particularly, it is inconvenient to have each element of the embeddings correspond to a single word. A more advanced approach would have the elements of the vector correspond to more complex linguistic and semantic characteristics of a word or text, thus making the size of the vectors independent of the size of the corpus. Such word representations are known as *word embeddings*. Recent research on word embeddings has been focused on developing techniques for learning word vectors of lower dimensionality while still capturing as much of the semantics as possible. Such models are much more useful than the Bag-of-Words-like models for the purposes of this thesis. There now exist techniques which can model the entire vocabulary of a language with word vectors in only a few hundred dimensions yet which are able to capture incredible amounts of semantic patterns. The most well-known class of such techniques is `word2vec`.

Word2vec

The main background literature of this project is the seminal work of the Google Brain team led by Tomas Mikolov [18, 20]. In these two papers, Mikolov et al. present two new models: the *Continuous Bag-of-Words Model* and the *Continuous Skip-gram Model*. These are neural network models which can be used to learn vector representations of words from enormous text corpora at a computational cost much lower than previous neural network models while offering large improvements in accuracy. These techniques and their associated algorithms are commonly referred to as `word2vec`.

Broadly speaking, in the training phase these models consider words and their *context windows*. A context window is a number of words occurring directly before and after the word in question in the text corpus. This provides the context for the word. The Continuous Bag-of-Words model is trained by attempting to correctly classify words based on the words in their context windows. On the other hand, the Continuous Skip-gram Model works the other way, by taking a word and attempting to predict its context window [18].

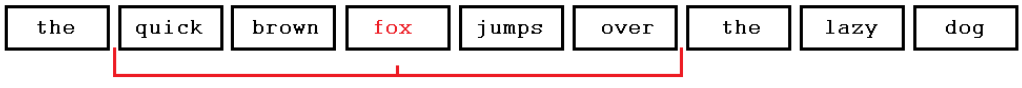


Figure 2.3: An example context window for the word `'fox'`.

Consider the context window in Figure 2.3. The Continuous Bag-of-Words model would learn by attempting to predict the word `'fox'` from the words `'quick',`

`'brown', 'jumps', 'over'`. The Continuous Skip-gram model would learn by attempting to predict the words `'quick', 'brown', 'jumps', 'over'` from the word `'fox'`. The words `'brown'` and `'jumps'` would be weighted more heavily since they appear closer to `'fox'` than the words `'quick'` and `'over'` do.

Since the contexts of words are used for training, the word embeddings produced by these models are able to capture certain semantic patterns. Incredibly, these patterns are modeled as linear vector relations. The famous result of the vector calculation $v_{king} - v_{man} + v_{woman}$ is a vector which is closer to v_{queen} than any other word vector, despite the algorithm not being programmed to know what a man or a woman is [21]. The model learns the common relationship between male and female words simply by the contexts in which these words occur. Such results are not limited to the English language. The same result has been tested and successfully reproduced with the corresponding Swedish words and their word embeddings, as shown in Figure 2.4a. Figure 2.4b shows the (Swedish) word embeddings most similar to the word embedding for `KTH`.

```
Kung - Man + Kvinna
Word: drottning, Similarity: 0.65
Word: prinsessa, Similarity: 0.62
Word: drottninggemål, Similarity: 0.58
Word: gemål, Similarity: 0.57
Word: tronarvinge, Similarity: 0.56
Word: kungen, Similarity: 0.56
Word: tronföljare, Similarity: 0.56
Word: kronprinsessa, Similarity: 0.55
Word: änkedrottning, Similarity: 0.53
Word: prinsessan, Similarity: 0.53
Word: tronarvingen, Similarity: 0.53
Word: kungar, Similarity: 0.53
```

(a) $v_{king} - v_{man} + v_{woman}$ in Swedish

```
Words most similar to kth:
Word: ingenjörsvetenskap, Similarity: 0.71
Word: civilingenjörsexamen, Similarity: 0.71
Word: högskolan, Similarity: 0.71
Word: högskolans, Similarity: 0.70
Word: kemiteknik, Similarity: 0.69
Word: högskola, Similarity: 0.69
Word: civilingenjör, Similarity: 0.69
Word: högskolas, Similarity: 0.69
Word: tekniska, Similarity: 0.67
Word: chalmers, Similarity: 0.67
Word: ingenjörsvetenskapsakademien, Similarity: 0.67
Word: lth, Similarity: 0.66
```

(b) v_{KTH}

Figure 2.4: Examples of word embedding similarity

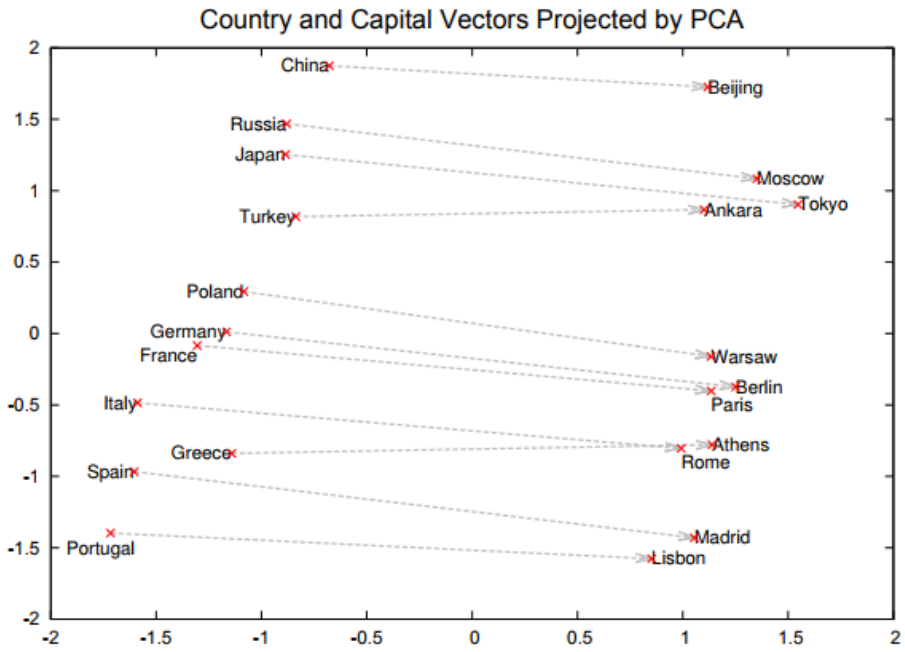


Figure 2.5: PCA projection of skip-gram word vectors. Linear relationship between countries and their capital cities successfully captured by the model, even though no information was provided to the model about what a capital city is. Image source: [20]

Figure 2.5 illustrates how `word2vec` word embeddings can be used to capture the semantic relationship between countries and their capital cities. This type of relationship has been successfully modeled because countries and their capital cities appear in similar linguistic contexts in the training corpus. In essence, sentences of the type *"[city X] is the capital of [country Y]"* are likely to occur for many different capital/country pairs.

FastText

Mikolov has since left Google Brain and now works for the Facebook AI Research team (FAIR) which is where the main developments in the field are now happening. The current state-of-the-art in word embeddings is the `fastText` library which has been made available open-source by FAIR. The algorithms used in `fastText` build upon the Continuous Skip-gram model and can be trained on corpora with billions of words in minutes [6, 14].

A disadvantage of the `word2vec` algorithms is that they ignore the structure of words by assigning different vectors to each word independently. Grammatical inflections are treated as completely separate words. I.e. words like `'sleep'`, `'sleeps'`, `'sleeping'`, `'slept'` are modeled independently of each other. Given a large enough corpus, the `word2vec` model is likely to be able to assign the inflections of `'sleep'` similar word vectors because they appear in similar contexts in the corpus. However, certain languages have much more complicated and rare inflections which makes it possible for certain inflections to not occur frequently enough even in very large corpora. English is a very easy language to model with word embeddings since the number of inflections is relatively low and compound words usually occur in open form (*post office*, rather than *postoffice*). Spanish, on the other hand, has

over 40 different verb inflections, and Finnish has 15 noun inflections [6].

The `fastText` algorithm known as *Subword Information Skip-gram* (SISG) solves the problem of modeling languages with rare word inflections by using *character n-grams*. A character n-gram is a sequence of n letters contained within a word. For instance, the word `sleep` contains the 3-gram `sle`. The SISG algorithm first adds the characters $<$ and $>$ to each word to mark the beginning and end of each word. Then each word is decomposed into all of its character n-grams where $n = 3, 4, 5, 6$. For example, `<sleep>` would be represented by the n-grams `<sl, sle, lee, eep, ep>`, `<sle, slee, leep, eep>`, `<slee, sleep, leep>`, `<sleep` and `sleep>`. Then, a word vector is trained for each of the n-grams of `<sleep>`. Finally, the original word `<sleep>` is assigned the word vector equal to the sum of the word vectors of its n-grams. The point of this is that the words `'sleep'`, `'sleeps'`, `'sleeping'`, `'slept'` share many n-grams and so their word embeddings will be correlated.

Furthermore, SISG remarkably allows for creating word embeddings for words which were not at all present in the training corpus. These are referred to as *out-of-vocabulary* words (OOV). Given an OOV word, as long as sufficiently many of its n-grams are present in the corpus, it can be modeled as the sum of the word vectors of those n-grams. This is a truly astonishing result as it gives the model a much deeper knowledge of the language and allows for just about any sequence of characters to be systematically assigned a word embedding.

Pre-trained Word Embeddings

The fact that these models are able to capture strong semantic patterns as linear vector relations is incredibly powerful for semantic analysis of large text corpora. The potential drawback of these models is that they require enormous quantities of training data (and enormous amounts of computing power in order to train efficiently). However, these models concern *language*. In most applications, the semantics of the natural language should be roughly the same as in a very general setting. In other words, as long as the text used in the data does not have completely different meanings than the natural interpretation (sarcasm, metaphors, allegories, etc) then one can use a pre-trained model trained by somebody else. In the case of this thesis, the text comes from professional traders whose job it is to document the changes they make to transactions. Although they employ frequent use of nomenclature, it is reasonable to expect that they should avoid sarcasm in their comments. Thus, using pre-trained models should work fine in this context.

The `fastText` developers have released large pre-trained models for 294 different languages (including Swedish). The Swedish model contains roughly 1.1 million words and their word vectors and has been trained on Swedish Wikipedia articles. The word vectors have dimension 300. The `fastText` library is under active development. For a detailed guide on how to obtain and install the `fastText` library, see appendix A.

2.4 Probability Theory

A Generative Model for Discourse

In two papers [2] and [3] by Arora et al. an interesting probabilistic model for the process by which discourse is generated is presented. The model provides theoretical

support for certain heuristics in algorithms generating word embeddings. The relevant aspects of the model will be presented here but for a more detailed description see the papers.

In [2] the model is a hidden Markov model (HMM) in which a *discourse vector* $c_t \in \mathbb{R}^d$ performs a discrete random walk on the unit sphere. The discourse vector is a representation of the context of some current discourse at time t, i.e. what is currently being talked about. Given that the discourse vector is in some state at time t, a word w_t from a vocabulary $\mathcal{V}$ is randomly emitted. The discourse vector c_{t+1} for the next time step is then obtained by adding a small displacement vector to the previous discourse vector. In [3], the model is modified slightly. Given a sentence s, the discourse vector is held constant throughout the sentence. The authors argue from empirical evidence that the discourse vector does not tend to change much within a single sentence. As such, in the model the discourse vector c_s moves over sentences rather than time steps.

For any word w_t, a time-invariant word embedding $v_{w_t} \in \mathbb{R}^d$ exists. The probability of a word $w \in \mathcal{V}$ being emitted in sentence s at time t given a discourse vector c_s is modeled as

$$\mathbb{P}(w_t|c_s) = \alpha\mathbb{P}(w_t) + (1-\alpha)\frac{\exp(v_{w_t}^\mathsf{T} b_s)}{Z_{b_s}}$$

where

$$b_s = \beta c_0 + (1-\beta)c_s, \quad c_0 \perp c_s$$
$$Z_{b_s} = \sum_{w\in\mathcal{V}} \exp(v_w^\mathsf{T} b_s)$$

and α, β are hyperparameters. The first term in the probability, $\alpha\mathbb{P}(w_t)$ is a smoothing term which accounts for the fact that there exists some probability of a word being emitted which is independent of the current discourse. The vector $b_s = \beta c_0 + (1-\beta)c_s$ is a shifted discourse vector. Here, c_0 represents some common time-invariant discourse bias. Z_{b_s} is a normalization constant.

Smooth Inverse Frequency

Given a sentence s and the word embeddings for its words, the discourse vector c_s can be estimated. The method for doing so involves first estimating b_s using maximum likelihood estimation. It turns out that the MLE estimate of b_s involves a weighted sum of the word embeddings in the sentence. The weights in this sum are known as *smooth inverse frequencies* (SIF) [3].

Let $L(b_s) = \prod_{w\in s} \mathbb{P}(w|c_s)$ be the likelihood function of the sentence s.

$$L(b_s) = \prod_{w\in s} \mathbb{P}(w|c_s) = \prod_{w\in s} \left(\alpha\mathbb{P}(w) + (1-\alpha)\frac{\exp(v_w^\mathsf{T} b_s)}{Z_{b_s}}\right)$$
$$\log L(b_s) = \sum_{w\in s} \log\left(\alpha\mathbb{P}(w) + (1-\alpha)\frac{\exp(v_w^\mathsf{T} b_s)}{Z_{b_s}}\right)$$

Let

$$l_w(b_s) \triangleq \log\left(\alpha\mathbb{P}(w) + (1-\alpha)\frac{\exp(v_w^\mathsf{T} b_s)}{Z_{b_s}}\right)$$
$$\Rightarrow \log L(b_n) = \sum_{w\in s} l_w(b_s)$$

In [2] it is argued that Z_{b_s} is roughly the same for all b_s, thus let $Z_{b_s} = Z$, $\forall b_s$. Then,

$$l_w(b_s) = \log\Big(\alpha\mathbb{P}(w) + (1-\alpha)\frac{\exp(v_w^\mathsf{T} b_s)}{Z}\Big)$$
$$\nabla l_w(b_s) = \frac{1}{\alpha\mathbb{P}(w) + (1-\alpha)\frac{\exp(v_w^\mathsf{T} b_s)}{Z}}(1-\alpha)\frac{\exp(v_w^\mathsf{T} b_s)}{Z}v_w$$

A first-order Taylor expansion of $l_w(b_s)$ gives

$$l_w(b_s) \approx l_w(0) + \nabla l_w(0)^\mathsf{T} b_s$$
$$= l_w(0) + \frac{1}{\alpha\mathbb{P}(w) + \frac{1-\alpha}{Z}}\frac{1-\alpha}{Z}v_w^\mathsf{T} b_s$$

Now consider again the log-likelihood

$$\log L(b_n) = \sum_{w\in s} l_w(b_s) \approx \sum_{w\in s}\Big(l_w(0) + \frac{1}{\alpha\mathbb{P}(w) + \frac{1-\alpha}{Z}}\frac{1-\alpha}{Z}v_w^\mathsf{T} b_s\Big)$$
$$= \text{constant} + \sum_{w\in s}\Big(\frac{1}{\alpha\mathbb{P}(w) + \frac{1-\alpha}{Z}}\frac{1-\alpha}{Z}v_w^\mathsf{T}\Big)b_s$$
$$= \text{constant} + \Big(\sum_{w\in s}\frac{a}{\mathbb{P}(w)+a}v_w\Big)^\mathsf{T} b_s$$

where $a = \frac{1-\alpha}{Z\alpha}$.

Recall that c_0 and c_s are on the unit sphere. Thus, the convex combination b_s is also on the unit sphere. Given this, note that $\Big(\sum_{w\in s}\frac{a}{\mathbb{P}(w)+a}v_w\Big)^\mathsf{T} b_s$ is maximized when $\sum_{w\in s}\frac{a}{\mathbb{P}(w)+a}v_w$ and b_s are parallel, i.e. when

$$b_s = \frac{\sum_{w\in s}\frac{a}{\mathbb{P}(w)+a}v_w}{\|\sum_{w\in s}\frac{a}{\mathbb{P}(w)+a}v_w\|}$$

Since $\arg\max L(b_s) = \arg\max \log L(b_s)$ the maximum likelihood estimate of b_s is found to be approximately

$$\hat{b}_s \propto \sum_{w\in s}\frac{a}{\mathbb{P}(w)+a}v_w \qquad \blacksquare$$

Here, the weights $\frac{a}{\mathbb{P}(w)+a}$ in the sum are the aforementioned SIF weights.

In order to find c_s, the common discourse bias c_0 must also be estimated. Let $\mathcal{S}$ be a set of many sentences whose common discourse bias is to be found. In the model, this is done via principal component analysis where the estimate $\hat{c}_0$ is set to be the projection of $\hat{b}_s$ onto the first principal component of the matrix X whose columns are the estimates of b_s for all $s \in \mathcal{S}$.

Once c_0 has been estimated, the estimated discourse vector $\hat{c}_s$ for the sentence s can be computed as

$$\hat{c}_s = \frac{\hat{b}_s - \beta\hat{c}_0}{1-\beta}$$

Since principal components are orthogonal to each other, this gives the desired property of $c_0 \perp c_s$.

Chapter 3

Literature Review

This chapter provides a short review of relevant previous studies related to this thesis, and positions the thesis in the wider field of research.

This thesis exists in the intersection of two important and well-studied areas of machine learning. The first is traditional supervised learning with numerical features, this is the most prevalent form of machine learning in academia as well as in industry. The other is document classification, i.e. supervised learning using text strings as input data. A more detailed description of the data used in this thesis will be provided in Chapter 5, but for now suffice to say that the data consists of both numerical features (nominal trade value, time between trade creation and modification, etc.) and text features (short strings documenting modifications made to existing trades).

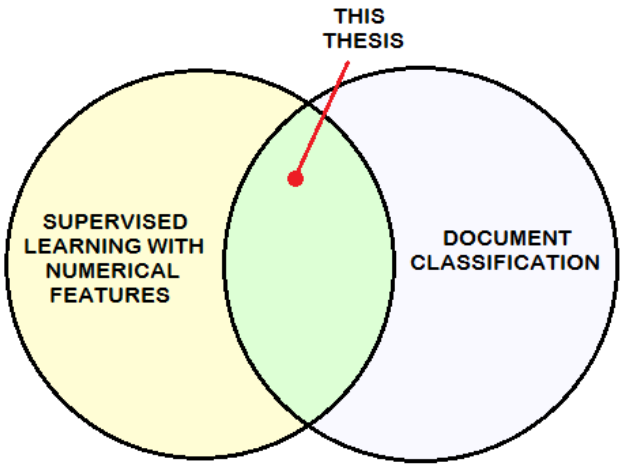


Figure 3.1: Intersection of traditional supervised machine learning and document classification

Academia and industry seems well-equipped to handle the case of a machine learning problem with *only* numerical features or *only* text features. There seems to have been significantly less research into how to deal with problems where the data contains both numerical and text features. This could be either a traditional supervised learning problem where most features are numerical but the dataset also includes one or more text features (as in the case of this thesis) or a document

classification problem where one wants to incorporate some numerical data into the model as well (cf. [16]).

The simplest and most intuitive approaches to the problem of converting text to numerical features are based on the Bag-of-Words model as described in section 2.3. A commonly used model is to create word embeddings for texts where the elements correspond to the frequency of occurrence (in the text) of the most common words (in the corpus) [28]. Again, a natural extension of this model is to weight each word by a suitable metric. Weighting by each word's *tf-idf* score means that rare words which occur frequently in a text are weighted more heavily. Using *tf-idf* weighted embeddings is the most common approach for real-valued feature vector representations of text [9].

Modern research in deep learning has brought on a variety of word embedding models trained by neural networks. These include the previously discussed `word2vec` (2013) [18, 20, 21] and `fastText` (2016) [6, 14] models, as well as numerous other popular similar models such as UC Berkeley's `Caffe` (2014) [13] and Stanford's `GloVe` (2014) [23]. Collectively, the word embeddings produced by such models are known as *distributed word representations*.

There has been some research into how pre-trained distributed representations can be used to improve performance of other machine learning tasks. Turian et al. [26] add word embeddings as extra features to improve performance in some NLP labeling and prediction tasks. However, these are exclusively NLP tasks, i.e. not containing numerical features as well. Correa et al. [7] use `word2vec` embeddings, and combine them in a way similar to in this thesis, for sentiment analysis of tweets; again, a pure NLP task.

Macskassy et al. [17] take the converse approach to the problem. Instead of attempting to convert the text features to numbers for use in machine learning algorithms, they convert the numerical features to text-like representations and use document classification algorithms.

There does not seem to have been much research into the effects of using pre-trained distributed word representations to improve the performance of machine learning models which are not pure NLP models. Given how new the word embedding models used in this thesis are and how quick developments in the field are, it is likely that if the problem is not novel then at least any similar studies rapidly become outdated.

Chapter 4

Data

This chapter describes the data used in this thesis. Beyond a brief description of the contents of the data, the chapter also discusses data quality and the steps taken to prepare data for the modeling.

Confidentiality

The data used in this thesis comes from the trading systems of Svenska Handelsbanken. The data contains information about trades made by SHB traders. Due to the strict confidentiality of such information, the raw data can not be presented in detail within this thesis. Furthermore, in order to preserve client, counterparty and trader confidentiality all sensitive information in the data which could link the trade to an individual, client or counterparty was anonymized prior to the start of this thesis project. This includes the name or other identifier of the client, counterparty and the SHB trader facilitating the trade.

4.1 Data Description

Trade Modifications

The dataset made available by SHB for this thesis contains information about certain trades made on their platforms between the dates 2014-07-01 and 2017-10-03. The raw data contains approximately 190,000 data points. More specifically, the data concerns trades which have been *modified* after the initial trade date. A modification of a trade refers to some property of the trade conditions being intentionally changed after the trade has been initially entered into the trading system.

There are several possible reasons for why such a modification is made. One possibility is to modify a trade in order to correct a mistake made at the initial entering of the trade. The trade may have mistakenly been entered with an incorrect maturity date, incorrect margin, not booked on the right account, double-booked, traded in the wrong direction, entered with missing information etc. These are typical examples of *fat-finger errors* which are central in operational risk.

Another possible reason for why a trade is modified after the trade has been entered is if the trade is made on the behalf of a client who then requests the modification. This may be due to a mistakenly incorrect trade request by the client, again due to a fat-finger error. It could also be so that the conditions of the trade allow for the client to change certain trade properties.

Technical errors are another possible reason for having to modify trades. A system bug may cause values in the trading system to be entered incorrectly or in an invalid way so that the errors must be corrected ex post facto. It could also be that a system error causes the trade to not execute or to execute in an unexpected manner.

Certain trades are designed in such a way that they can or will require modifications in the future. This could again be because the conditions of the trade allow for the trader or client to edit certain trade properties. It may also be so that new information which is not available at the trade inception appears and must be entered into the system.

There are other more obscure ad hoc reasons for why trades must be modified ex post facto, but the categories listed above give a good picture of the main explanations behind the modifications studied in this thesis.

Trader Comments

Regardless of what the reasoning behind a modification of a trade is, when the modification is made the trader administering the modification must type a short text note into the system documenting the modification. This comment is entered into a free text field, meaning the trader can essentially enter any arbitrary sequence of characters, including leaving the field empty.

The purpose of the trader comment is to document any manual modifications being made to trades in the system. Since the trades can often be of a very large value and the changes to them sometimes critical, it is important that a convenient paper trail is established. Simultaneously, writing long and detailed documentation of ones activities is not an especially value-adding activity. Particularly not when considering the fact that a large portion of the work of modifying trades concerns the exact same type of modification and so is likely to be quite repetitive.

As with any practical machine learning application, there are some data quality issues which deserve mentioning. Listed below are some of the main data quality issues with the trader comments.

- **Length.** The average length of a trader comment in the system is very short, with the vast majority of comments being shorter than five words. This is evidence of the low incentive for traders to write descriptive comments. The length of the comments (in terms of number of words) is important because it governs which NLP models are applicable for this particular case.

- **Empty comments.** Some trade modifications have no comment attached to them at all. In most cases this is due to the modification being automatically generated, but there are also some manual modifications in the dataset with an empty comment.

- **Language.** The vast majority of comments in the dataset are written in Swedish. However, the data also contains comments made in many other different languages. These include at least English, Norwegian and Finnish, with the possibility of other languages also being present in the dataset. The presence of multiple languages is problematic because this study uses pre-trained Swedish word embeddings. The word embeddings have only been trained on a Swedish corpus, thus they will not be able to effectively model words from other languages.

- **Misspellings.** Many of the comments seem to have been written very quickly with little concern about correct spelling. This actually turns out to be less of a problem than one might suspect. The comments will be modeled with `fastText` word embeddings which use character n-grams meaning comments with minor spelling errors can still be modeled effectively. See Section 2.3 for a more detailed description of `fastText` and character n-grams.

- **Acronyms.** Acronyms are also common in the trader comments as another means of speeding up the documenting process. Acronyms are potentially a much more severe problem than misspellings. This is because unless the pre-trained word embedding model used contains a trained vector for that acronym it will be very difficult to construct such a vector from the acronym's n-grams since an acronym naturally contains very few characters which additionally tend to appear in a rare sequence.

- **Nomenclature.** Since the traders are discussing their own work in the comments, they contain a lot of finance nomenclature. This could include mentioning things such as *option deltas* and *spot rates* which are not typically common in general text topics. As long as these technical terms are common enough to appear on Wikipedia (which is what the word embeddings are trained on) then this is no problem (rather, it is good to use such words). The problem is when the traders use very obscure or self-invented nomenclature and acronyms. This is very problematic because if a word is rare enough to not be on Wikipedia then it will not be included in the word embedding model. Furthermore if it is not descriptive enough then the model will not be able to reconstruct it using the character n-grams. Since these technical terms are often the most crucial word in the meaning of the comment it is a major information loss if the model can not understand them.

Despite all these flaws the comments still clearly have a lot of information value in them. From a simple ocular inspection, reading the comment is often enough to understand what the change to the trade was. Thus there is still hope that helpful features can be extracted from them. With that said, efforts to improve the way the comments are written with regards to the points listed above would almost certainly improve the results of projects such as this one.

Other Features

The dataset contains many other features with information about the trade modification beyond the trader comment. Almost 50 different features of varying importance are included in the data, only the key ones will be described here. There are essentially two groups of features for each data point. One contains information about the trade; the instrument and instrument type that was traded, a time-stamp for the trade, name of the counterparty, maturity date (if applicable), nominal value, price, etc. The other group of features contains information about the modification made to the trade; a time-stamp for the modification, ID of the trader, the type of modification made, what values were changed and of course the trader comment.

All of the other features in the dataset are either numerical or categorical. Only the trader comment is a true free-text.

4.2 Labeled Data

Two smaller sets of labeled data were provided. These contain modifications of trades which have been manually labeled as the type of suspicious behavior the risk control division wants to detect. These contain 242 and 45 data points respectively. The two sets intersect but the smaller is not fully contained in the larger.

All but two of the labeled data points concern trades in foreign exchange rate derivatives; FX spot trades, FX futures contracts and FX swaps.

4.3 Data Preparation

As with most machine learning projects, a lot of data cleaning was required in order to improve quality and prepare the data for the algorithms. Only the trader comments were cleaned since this thesis focuses purely on that feature. Below are again the same main data quality issues listed, this time with how they were remedied in the data cleaning process.

- **Length.** The length of the existing comments is difficult to do much about. Worth mentioning however is that some of the other remedies, such as expanding acronyms and removing meaningless words, can change the number of words in a comment somewhat. The algorithms used are robust to small changes in the number of words.

- **Empty comments.** It is not possible to add words to an empty comment. The question rather becomes what value to assign the feature if there is no comment. Again, many of the modifications with empty comments were automatically generated by the system and thus removed from the dataset as only modifications made by actual persons are of interest. In the case of manual modifications the property that the trader chose to not input any comment is still somewhat interesting, and the model is capable of assigning a unique word vector to the empty string.

- **Language.** The only translations made were for individual English words which occur often in otherwise Swedish comments, such as for instance the word *trade*. Some efforts were made to remove data points where the comment was entirely in a language other than Swedish. This was done with simple queries until the number of non-Swedish comments was deemed to be acceptably low. If the dataset had contained a larger number of non-Swedish comments an alternative approach could have been to use a language detection algorithm combined with an automatic translator to translate the comments.

- **Misspellings.** As mentioned previously, the use of n-grams make misspellings less of a problem. Still, correct spelling is preferable for the model in this thesis so a small amount of effort was made to correct the most common spelling errors. An interesting alternative approach could have been to create a new feature counting the number of spelling errors, with the hypothesis that spelling errors can be useful in predicting suspicious behavior.

- **Acronyms.** As acronyms are a more severe issue, more time was spent addressing them. This process consisted of identifying common acronyms and creating a translation key mapping each acronym to its expanded form. Again,

increasing the number of words by a few is no problem so an acronym can (and should if necessary) be replaced by more than one word.

- **Nomenclature.** As with the acronyms, any obscure nomenclature was translated and added to the translation key. Many of the terms were highly technical and had to be translated by individuals with more domain knowledge.

Other than the cleaning listed above, many data points were filtered out of the dataset. The dataset contains certain indicators of system generated modifications. Again, since those are not interesting in this type of operational risk control they were removed from the dataset. Furthermore, all trades in instruments other than FX spots, FX futures and FX swaps were removed from the dataset. This was because the vast majority of the labeled data was for FX trades. It is reasonable to expect that the way traders write differs to some degree between instruments. Thus, in order to perform supervised learning the decision was made to exclude the other types of trades. If one were to obtain a set of labeled data for some other type of trade, for instance bond trades, then the same type of modeling could be applied to that data. Finally trades with comments in languages other than Swedish were removed as mentioned above. After all data cleaning, the result was a dataset of approximately 17,000 data points, all in FX trades.

Chapter 5

Methodology

This extensive chapter describes the methodology by which the research in this thesis was conducted. Mainly, this involves describing the design of the algorithm developed for converting trader comments into numerical features, this algorithm will be frequently referred to simply as *the algorithm*. Sections 5.1 and 5.2 describe the algorithm and its modules. Section 5.3 discusses the supervised classifier used to evaluate the quality and performance of the features produced by the algorithm. Section 5.4 covers how the evaluation and comparison of different models was conducted. Finally, Section 5.5 describes the specifications of the machine used to run the programs.

5.1 Brief Algorithm Overview

The goal of this thesis project was to transform free-text comments from the trading platform into quantitative features for supervised learning. To do this an algorithm was designed and implemented. The algorithm takes the comments from the trading platform as input and categorizes them, outputting a number for each comment indicating which category it belongs to. The main algorithm can be divided into four modules which can be run independently:

I **Data cleaning.** This part prepares the free-text comments for the second module by attempting to maximize the information and clarity contained in each word. This is done by expanding abbreviations and acronyms, correcting spelling errors, translating certain English words, etc. as mentioned in Section 4.3. Beyond the initial cleaning an effort is made to remove *stop words* from the comments. These are very common words in the language which contain little semantic information.

II **Word embeddings.** After cleaning the data, the second module of the algorithm maps each comment to a vector. This is done by first loading a large pre-trained `fastText` model. Each word in a comment is mapped to a word vector. OOV words are decomposed into their character n-grams an reassembled using the pre-trained n-gram vectors to obtain the word embedding.

III **Sentence embeddings.** After finding a word embedding for each word in a comment, the word vectors are combined to obtain a single vector representation for the whole comment. Several different ways of combining pre-trained word embeddings into a sentence embedding are explored in this thesis.

IV **Clustering.** The final module takes the whole dataset of sentence embeddings and clusters them in order to form distinct categories of comments. The index of each comment's cluster is returned as the output of the algorithm. Several different clustering models are tested.

In the following section, each module of the algorithm and its implementation will be presented in further detail.

5.2 Algorithm Description

I. Data Cleaning

Much of the initial data cleaning was done in Excel to expand abbreviations and acronyms, correct spelling errors, translate certain English words, as well as filter out unwanted data such as non-FX trades and trades with non-Swedish comments. Again please refer to Section 4.3 for more detail on this.

An important step in this module is the removal of *stop words*. Stop words are very common words in any language which do not carry much interesting semantic meaning. Examples of stop words in English are `'on'`, `'for'`, `'as'`, `'if'`, etc. and in Swedish `'att'`, `'i'`, `'med'`, `'på'`, etc. To understand why removing stop words makes sense one must understand how the algorithm processes a sentence. Consider the following hypothetical trader comment:

```
Updated the rate and maturity date
```

The sentence contains two stop words: `'the'` and `'and'`. Removing the two stop words leaves the following sentence:

```
Updated rate maturity date
```

Although the sentence now reads a bit awkwardly, it is still very clear what the meaning of it is. Very little semantic information has been lost by the removal of the two stop words. Now consider how the algorithm reads a sentence such as this. Each word is individually converted into a single word vector and all of the vectors in the sentence are then combined to form the sentence vector. In fact, the order of the words does not matter. The sentence with the two stop words included contains six words, so six word vectors will be created and then combined. But two of the words contribute very little to the meaning of the word, and so their word vectors are essentially just noise when it comes to finding the sentence representation. By first removing the two stop words only four word embeddings have to be found and combined. This way, each of the four meaningful words contribute more to the final sentence vector.

The implementation of the stop word filtering is done via a Python script using the `Cucco` library.[1] The library supports 50 languages including Swedish and the native stop word list has been used. There is no general consensus on exactly which words are considered to be stop words. The Swedish stop word list used by `Cucco` contains 401 unique stop words.

Another approach to dealing with stop words could be to give them a low weight when combining word embeddings. This is a valid approach and such a weighting

[1] https://github.com/davidmogar/cucco

scheme will be explored later in this thesis to weight other words based on their estimated importance. However, since the semantic information in a stop word is next to zero it is more convenient to simply remove them altogether from the data.

II. Word embeddings

After the data has been cleaned and stop words have been removed the remaining words in the trader comments are ready to be converted into word embeddings. This is done using pre-trained `fastText`[2] word embeddings. The word embeddings used were trained by FAIR on Swedish Wikipedia articles.[3] For a detailed description of how the embeddings were trained, see the following recent article by Mikolov et al. [19]

The Swedish `fastText` model in `.bin` file format is roughly 5 GB and contains ∼1.1 million words. Each word embedding is a real valued vector of dimension 300. Loading such a model into the RAM can be a strenuous effort for some machines. For initial algorithm testing a smaller `word2vec` model containing ∼50,000 words was used.[4] The file size of this smaller model is only a few megabytes, so the model loads instantly. For a more detailed description of how to install, download and use the `fastText` library, please see Appendix A.

The `fastText` library is written in C++ and does not build natively on Windows machines. The implementation of the `fastText` model in this project was done via the official FAIR Python bindings for `fastText`. A Python script loads the pre-trained word embeddings and the dataset of cleaned trader comments. Then for each word in each comment the script calls a function which gets the word embedding for that word from the `fastText` model. This function returns a word embedding for any sequence of characters, including OOV words since their word embeddings can be constructed from the character n-grams.

Consider again the example sentence from before, with the stop words removed. In the second module, each of these four words will be replaced by a pre-trained word embedding.

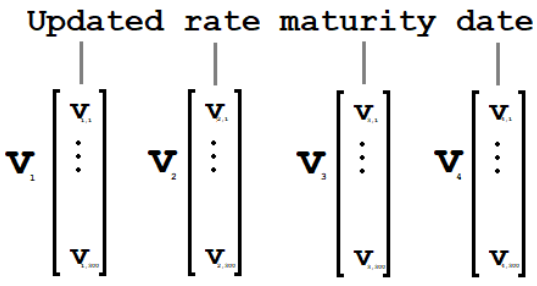


Figure 5.1: Words in an example sentence replaced by their word embeddings

This module is very much the core of the algorithm since it is here the text strings get converted into numerical vectors. Despite its importance, the implementation is

[2] https://github.com/facebookresearch/fastText

[3] The word embeddings used in this thesis (and word embeddings for 293 other languages) can be downloaded at the following website: https://fasttext.cc/docs/en/pretrained-vectors.html

[4] https://github.com/Kyubyong/wordvectors

very simple due to the existence of the `fastText` library, Python bindings and large pre-trained models.

Motivating the Usage of Word Embeddings

At this point it is fair to take a step back and think about why word embeddings are a good approach to this problem. The fields of text mining and natural language processing are rich and contain many other techniques than word embeddings which could have been applied in this scenario. However, a typical text mining problem concerns analysis of longer texts than in this case. Such a problem could for instance be to automatically classify thousands of legal documents, or quickly analyze financial press releases and make investment decisions automatically. In these examples, the texts are typically many paragraphs or even pages long. This means that there are a lot more information sources (words) to interpret in each text. This thesis deals with trader comments with as few as just a single word. Thus, techniques which are designed to interpret longer texts might not be applicable here. It is crucial to use a method which can extract maximal information from each word which is input.

Furthermore, since most of the data is unlabeled, many interesting supervised learning schemes are not applicable to this problem. Instead, unsupervised methods have to be used in the feature mining. Word embeddings are perfect for this situation since each individual word can be mapped to a unique vector.

Since the comments are short, there is not much of a context for most of the words. Thus training a model based on context windows would be very difficult. However, by using a pre-trained word embedding model this problem is eliminated. The words which lack context in the trader comments can appear in some contexts in the Wikipedia corpus and thus they can still be modeled. This allows for the potential to capture information from every single word entered by the trader.

In particular, the `fastText` library is especially suitable. Not only are the word vectors of state-of-the-art quality, the SISG algorithm it uses allows us to model OOV words. Due to the nature of the trader comments, there are plenty of OOV words, be it misspellings or self-invented acronyms. Attempts are made to translate many of these in the data cleaning module, but not all able can be captured. Thus the fact that the algorithm is able to make an educated guess at what a suitable word embedding should be for an OOV word is extremely helpful.

III. Sentence embeddings

At this point, each trader comment has been converted into a set of word embeddings. However, the aim is to obtain a single, simpler numerical representation of each comment. The next step in the algorithm is thus to convert the set of word embeddings into a single vector, a sentence embedding. This can be done in many different ways, two have been explored in this thesis.

Averaging

The simplest way of obtaining a sentence embedding from a set of word embeddings is to let the sentence embedding be the vector mean of the word embeddings. Consider again the sentence $s =$ `Updated rate maturity date` and let $v_{w_1}, v_{w_2}, v_{w_3}, v_{w_4}$ be the word vectors of the words in that sentence. The sentence embedding c for this

sentence would be

$$c = \frac{1}{4}\sum_{i=1}^{4} v_{w_i}$$

and more generally for an arbitrary sentence containing m words the sentence embedding would be

$$c = \frac{1}{m}\sum_{i=1}^{m} v_{w_i}$$

This simple method does surprisingly well at producing sentence embeddings. Recall from Section 2.3 that modern word embedding models are able to capture linguistic patterns as linear vector relations, e.g. the vector relationship between the embeddings for different countries and their capital cities is linear and approximately equal. In an effort to preserve these linear relationships, it seems intuitive to let the sentence embeddings be linear combinations of the word embeddings.

There is some theoretical justification for setting a sentence embedding to be the mean word embedding. In the `word2vec` CBOW model, the probability of a word w_k being emitted as a function of the previous $k-1$ words $w_1, w_2, ..., w_{k-1}$ is

$$\mathbb{P}(w_k|w_1, w_2, ..., w_{k-1}) \propto \exp\left(v_{w_k}^{\mathsf{T}} \frac{1}{k-1}\sum_{i=1}^{k-1} v_{w_i}\right)$$

[2]. In other words, the likelihood of the next word to follow a sequence of words in a sentence depends on how aligned the word embedding of the next word is with the mean word embedding of the previous words.

Averaging is very easy to implement as there exists a function in the `fastText` library which does exactly this.

Weighted Averaging

Is it possible to do better than to simply average the word embeddings to get a sentence embedding? Given that a linear combination seems to be an attractive idea for a model, the natural extension is to have the sentence embedding be a weighted average of the word embeddings. This preserves the model linearity but allows for a weighting scheme to be employed. This is useful because it allows for the model to assign a larger weight to words it thinks are important and vice versa. In a weighted average model, a sentence embedding c is set to be

$$c = \frac{1}{m}\sum_{i=1}^{m} u_i v_{w_i}$$

where u_i is the weight for word w_i.

The obvious question is how to choose the weights u_i. A potential candidate is to set the weights to be the tf-idf scores for their words. Recall from Section 2.3 that the tf-idf score is the product of the term frequency and the inverse document frequency. If a word occurs many times in a comment then the term frequency will be higher. If a word is very uncommon in the dataset then its inverse document frequency will be higher. Thus rare words and/or words which occur many times in a sentence would be weighted more heavily in that comment if tf-idf scores are used for weighting. This is seems like an attractive idea, however it has some flaws. Consider the term frequency when applied to very short sentences such as in this

thesis. For a comment containing less than five words, it is very unlikely that any word is repeated more than once in the same comment. Thus the term frequency will almost always be 1 for every word. The tf-idf score was designed to be applied on longer documents where certain common words could be expected to appear many times in the same document, which definitely is not the case in the data for this thesis.

Consider the generative discourse model from Section 2.4. In the paper [3], the authors argue that the discourse vector c_s for a sentence s is a suitable choice of sentence embedding. It was shown that the discourse vector can be estimated from data by first computing the maximum likelihood estimate $\hat{b}_s \propto \sum_{w \in s} \frac{a}{\mathbb{P}(w)+a} v_w$ and then subtracting its projection onto the first principal component of the matrix whose columns are the estimates of b_s for a large collection of sentences.

This weighting scheme includes a weighted average where the weights are the smooth inverse frequencies (SIF), i.e. $\frac{a}{\mathbb{P}(w)+a}$. For more common words, $\mathbb{P}(w)$ is larger and thus those words are down-weighted. This is essentially the same rationale as for tf-idf weights, only that there is no need for term frequencies in short sentences.

The SIF weighting scheme was implemented as the other way to generate sentence embeddings. A Python implementation of SIF weighting created by the authors exists.[5] However, their implementation uses (English) `GloVe` word embeddings. Reconfiguring their implementation to work with `fastText` embeddings was deemed more inconvenient than writing a new implementation from scratch. Thus, a `fastText` configured implementation of the SIF weighting scheme was written in Python for this thesis. The implementation is in accordance with the description in Section 2.4. However a few details which are not discussed in the paper [3] deserve mentioning. It was shown in Section 2.4 that the MLE estimate of b_s is approximately

$$b_s = \frac{\sum_{w \in s} \frac{a}{\mathbb{P}(w)+a} v_w}{\| \sum_{w \in s} \frac{a}{\mathbb{P}(w)+a} v_w \|}$$

However, in the algorithm presented in [3] b_s is set to be

$$b_s = \frac{1}{|s|} \sum_{w \in s} \frac{a}{\mathbb{P}(w) + a} v_w$$

For most words $\mathbb{P}(w)$ is going to be small, so for word embeddings on the unit circle

$$\| \sum_{w \in s} \frac{a}{\mathbb{P}(w) + a} v_w \| \approx |s|$$

Furthermore, in the source code for the authors' Python implementation of the SIF weighting scheme it is noted that the parameter a is *"usually in the range [3e-5, 3e-3]"*, and the default parameter value is $a = 10^{-3}$. Thus the parameter in the implementation written for this thesis was also set to be 10^{-3}.

[5] https://github.com/PrincetonML/SIF

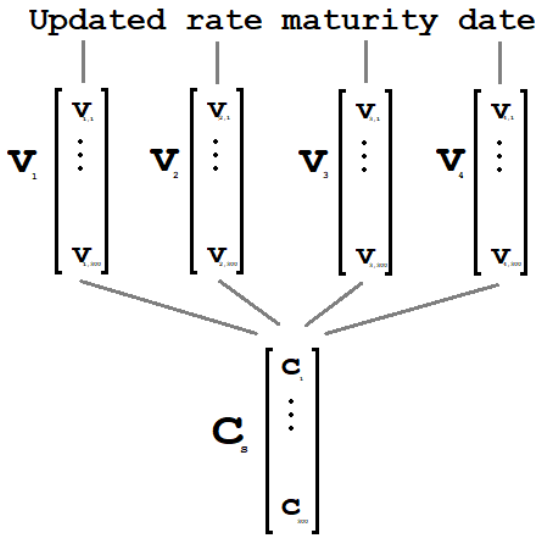


Figure 5.2: A sentence embedding created from word embeddings

IV. Clustering

The fourth and final module of the feature generation algorithm takes a set of sentence embeddings and performs clustering in order to reduce the set of 300-dimensional continuous vectors into a discrete set of categories. This is done in order to create a general-purpose representation of the information extracted from the trader comments which can be input into virtually any machine learning algorithm.

The underlying hypothesis behind this module is that there exists a finite set of general topics which the trader comments discuss. Thus an attempt is made to collect trader comments which concern the same topic and put them into the same bucket.

There are many different approaches to clustering unlabeled data. In order to motivate the choice of clustering algorithm, recall again from Section 2.3 that the word embeddings used in this thesis are able to capture linguistic and semantic patterns as linear vector relations. It is thus reasonable to suspect that certain clusters of distinct but related sentence topics could be linearly separable, or at least nearly linearly separable. This means that a clustering algorithm which produces linearly separated Voronoi cells, such as k-means, could effectively cluster this data. A non-linear clustering algorithm could potentially lead to too high cluster variance.

Two different clustering algorithms have been tested in this thesis. One is the k-means algorithm, which produces linearly separable clusters in accordance with the hypothesis above. The other is the DBSCAN algorithm which in contrast can produce arbitrary non-linear clusters. Both clustering models output a single value for each data point, corresponding to the cluster it belongs to. DBSCAN has a built in notion of noise which means it can classify data points as not belonging to any cluster. The property of a comment being unique enough to not belong to any cluster (meaning it was written in a very unique way) is certainly very interesting, thus the noisy data points are essentially considered to constitute their own cluster despite likely not being a coherent geometric cluster. For a more detailed description of the

clustering algorithms, see Section 2.2.

Hyperparameter Tuning

Both k-means and DBSCAN require the user to set certain hyperparameter values prior to usage. Setting such values a priori can be very difficult without significant domain knowledge. Instead, the parameter values can be tuned by testing a range of values and picking the one which gives the best model performance. This is relatively straightforward in supervised learning since model performance typically is easy to measure. In unsupervised learning it can be far less obvious how to tune the model. In this case however, although the tuning concerns hyperparameters in unsupervised models, the clustering is done in order to produce features for a supervised model. Thus the hyperparameters can be tuned based on their effect on the supervised model where the trader comment feature is used along with all the other features present in the original dataset.

Tuning the parameters is indeed possible in this case, however given that the supervised model used is a deep belief network which is relatively computationally time-consuming to train, there are technical limits to how finely the parameters can be tuned. Training times for the supervised model will be discussed further in later sections but suffice to say that the training time ranges from at least 10 minutes to several hours depending on the amount of data used and the hyperparameters of the DBN.

For the k-means model, the number of clusters k is tuned from $k = 10$ to $k = 200$ with increments of size 10. The DBSCAN model has two parameters which are both tuned in the set

$$\varepsilon \times \text{minPts} \in \{0.06, 0.1, 0.2, 0.3, 0.4, 0.5\} \times \{3, 5, 7, 9, 11\}$$

Additionally, all the parameters are tuned separately for the two different choices of sentence embedding generation techniques discussed in module III.

The clustering algorithms are implemented in Python using the widely used `sklearn` library.

5.3 Supervised Classifier

Again, the objective of the feature generation algorithm is to create features for a supervised classifier used to detect suspicious trading behavior. The method for creating features is general enough to be applied in virtually any supervised classifier. In the case of this thesis, a deep belief network has been used. The construction of this DBN is not of primary interest in this thesis, and the reader may choose to think of the DBN as a general black box classifier. However, for the purposes of transparency and reproducibility the DBN architecture and implementation will be discussed in brief.

The deep belief network takes a large dataset containing many features in addition to the trader comment and performs unsupervised feature detection followed by supervised classification as described in Section 2.2. The supervised learning is a binary classification where each data point is classified as suspicious or not suspicious. For the supervised learning, the labeled data described in Section 4.2 is used. Since this labeled data contains labels exclusively from the suspicious class, an equally large sample is randomly drawn from the data and assumed to be "clean" trades.

As the number of actual operational high-risk events is low in comparison to normal trading behavior it is likely that most if not all of the data points assumed to not be suspicious behavior are correctly classified.

For the classification, the labeled data is split into a training dataset and a test dataset. The training data is used to train the DBN and the test data is used to evaluate the performance. 40% of the labeled data is used for training and the rest for testing. This is different split than what is traditionally recommended for machine learning tasks, typically a larger portion of the data is allocated for training. The reason behind this choice is the low amount of labeled data available. Retaining a larger portion of data for testing creates a more challenging learning task with more room for models to make errors in order to gage performance. Crucially the same training and test data allocation is used for all models for a fair comparison. The empirical results show that the models achieve a very high classification rate despite the lower-than-usual amount of training data. An alternative approach could have been to employ a cross-validation scheme whereby the data is partitioned into many subsets and repeatedly trained using different subsets as test data each time. This allows for a better evaluation of model performance. However, it comes at the cost of increased computational time as the model must be re-trained many times. As the DBN used in this thesis is computationally expensive to train this presents a problem. Both hyperparameter tuning and cross-validation demand repeated re-training of the model. The decision was made to favor more detailed hyperparameter tuning over a cross-validation scheme. Note that the objective here is to *compare* model performance. Exactly what the performances of the models are is not of primary interest in this study, the real goal is to find out which model performs the best.

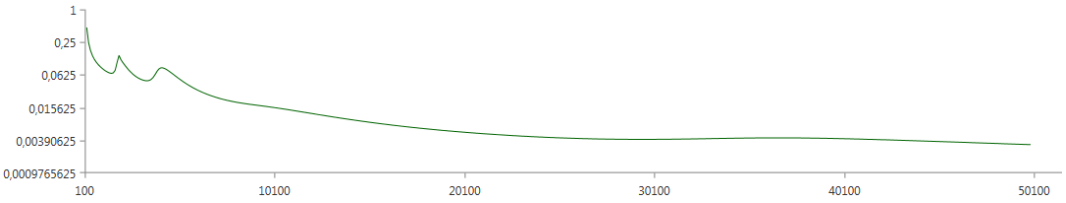


Figure 5.3: Error rate of the DBN during the supervised phase across epochs

The DBN architecture has 10 hidden layers. In the unsupervised learning phase, the model is trained for 100,000 epochs and in the supervised learning phase the model is trained for an additional 50,000 epochs. This has proven to be enough epochs to escape local optima in the early epochs. In Figure 5.3 two such local optima can be identified within the first 5,000 epochs of the supervised learning phase.

The implementation of the DBN used in this thesis is written in C#. The classifier predicts a class label for each data point in the dataset, i.e. it tells the user which data points it considers to be suspicious and which it does not. For the purposes of this thesis, the only interesting classification output is on the test set where the performance of the classifier can be measured. However, the real purpose of the model is of course to make a judgement about the level of operational risk in unlabeled data so that suspicious activity can be monitored and intercepted.

5.4 Evaluation

The evaluation metric used in this thesis is the classification rate, i.e. the fraction of data points in the test data which are correctly classified. A classification rate of 0.68 means that 68% of the test data was correctly classified.

The purpose of evaluation is to see if the models produced can outperform a benchmark, and if so which choice of hyperparameter values gives the optimal model. The benchmark tested against is the performance of the DBN if no trader comment feature is included. A myriad of models and hyperparameter values were evaluated against each other. For each iteration of running the DBN classifier, four distinct model degrees of freedom are set:

- **Sentence embedding generation technique.** The method for generating sentence embeddings from word embeddings is either averaging or weighted averaging with SIF weights.
- **Clustering algorithm.** Either k-means or DBSCAN is used for clustering the sentence embeddings.
- **Hyperparameters.** If k-means is used for clustering, k must be set. If DBSCAN is used, ε and minPts must both be set.
- **Data.** The amount of data used can be varied from iteration to iteration. Using less data speeds up the training time but potentially makes the model susceptible to overfitting and high variance.

The different combinations of sentence embedding generation techniques and clustering algorithms allows for four different models, named accordingly: *Averaged k-means, SIF k-means, Averaged DBSCAN* and *SIF DBSCAN*. Each one of these models can be used with varying hyperparameter values and input datasets.

Training the DBN on the full dataset is a computationally strenuous task. On the machine used in this thesis (described in Section 5.5) the training process takes approximately six hours. During the course of this thesis project, hundreds of models had to be evaluated. In order to facilitate this, much of the parameter tuning was done on a subset of the data containing only the trade modifications made during the time period from 2015-01-01 to 2015-03-01. This subset will be known as the *tuning set* (similar to but not exactly the same as a training set). This time period was chosen because a large portion of the labeled data is dated within this time period. Training the network on the data in this time period takes roughly ten minutes on the machine used.

5.5 Machine Specifications

There are two main computational choke points in this thesis. The first involves the usage of very large pre-trained word embedding models for module II of the algorithm. The file size of the model used in this thesis is roughly 5 GB. This can potentially be a problem on older machines. For instance, a processor with a 32-bit architecture is limited to $2^{32} \approx 4$ GB of memory meaning such a machine cannot load the full model used in this thesis. The second issue involves the training of the deep belief network used for classification. The training time can be closely dependent on the hardware used.

In this thesis project a machine with the following specifications was used to train the DBN:

- Intel Xeon E3-1240 v5 CPU, 3.50 GHz, 4 cores, 8 threads
- 16 GB DDR4 RAM
- Windows 7 64-bit OS

Chapter 6

Results

In this chapter the results of the DBN classifier, using various models for trader comment features, will be presented. First, the results of the hyperparameter tuning will be covered, followed by the results of the tuned models on the full dataset.

6.1 Hyperparameter Tuning

For the hyperparameter tuning a smaller subset of the data, the *tuning set*, was used. It includes only data from 2015-01-01 to 2015-03-01. This relatively short time-span includes all 45 labeled suspicious data points from the smaller of the two labeled datasets described in Section 4.2.

While tuning the hyperparameters of the unsupervised clustering models the classification rates of the DBN were benchmarked against the DBN's performance with no trader comment feature included. On the dataset from 2015-01-01 to 2015-03-01 the benchmark model with no trader comment feature achieved a classification rate of 0.732.

Averaged k-means

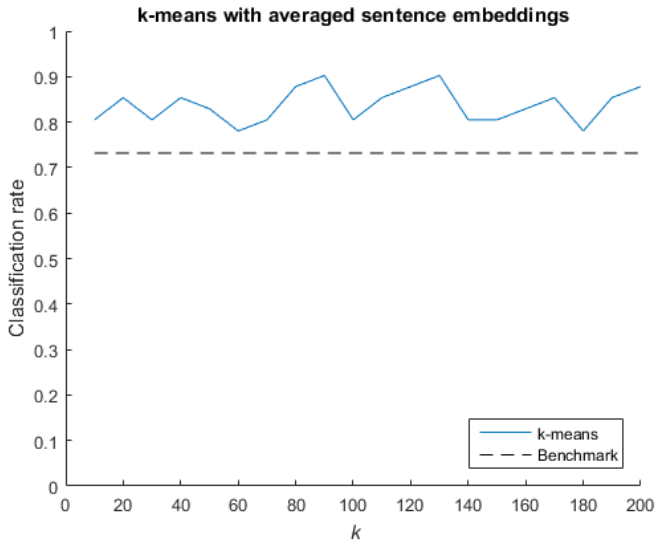


Figure 6.1: Classification rate of the DBN using the Averaged k-means model

Figure 6.1 shows the classification rate of the DBN where the trader comment feature has been generated from the k-means algorithm, with sentence embeddings computed as the average of the trader comments' word embeddings. The classification rate is plotted for various values of k in the interval $[10, 200]$. Also, the classification rate of the benchmark model is plotted. Interestingly, the model with the trader comment feature included achieves a higher classification rate for all tested values of k. This is a promising indicator that including a feature generated from the trader comments does improve the predictive power of the DBN classifier. The highest classification rate achieved by the model in Figure 6.1 is 0.902 which is achieved when $k = 90$ and $k = 130$.

SIF k-means

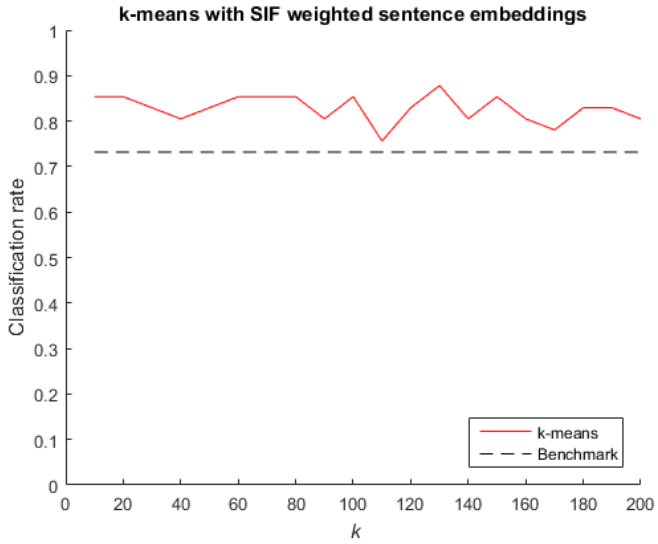


Figure 6.2: Classification rate of the DBN using the SIF k-means model

Figure 6.2 also shows the performance of the DBN with the trader comment feature generated by k-means. However, this time the sentence embeddings are computed using the SIF weighting scheme. Again, the model seems to outperform the benchmark. The highest classification rate achieved by the SIF k-means model is 0.878 at $k = 130$.

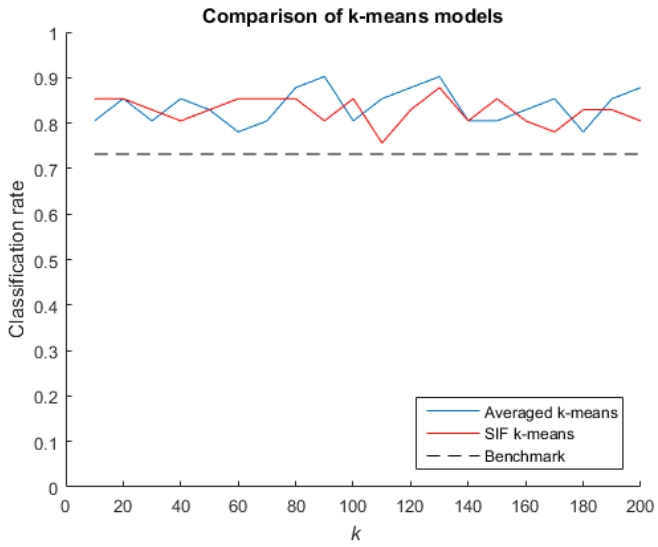


Figure 6.3: Comparison of Averaged k-means and SIF k-means models

Figure 6.3 shows the performance of the two k-means based models plotted in the same window. There are a few interesting things to note. Both models achieve peak performance at $k = 130$, indicating that this is a suitable number of clusters for this dataset. Furthermore, the two models seem to perform comparably and both clearly outperform the baseline model. The Averaged k-means model achieves a slightly higher best classification rate. It was argued in Section 5.2 that the k-means algorithm should perform well if the cluster structure in the data is such that clusters can be linearly separated. A possible explanation for the somewhat inferior performance of k-means on the SIF generated dataset is that the SIF weighting scheme somehow distorted some of the linear relationships in the word embedding data.

Averaged DBSCAN

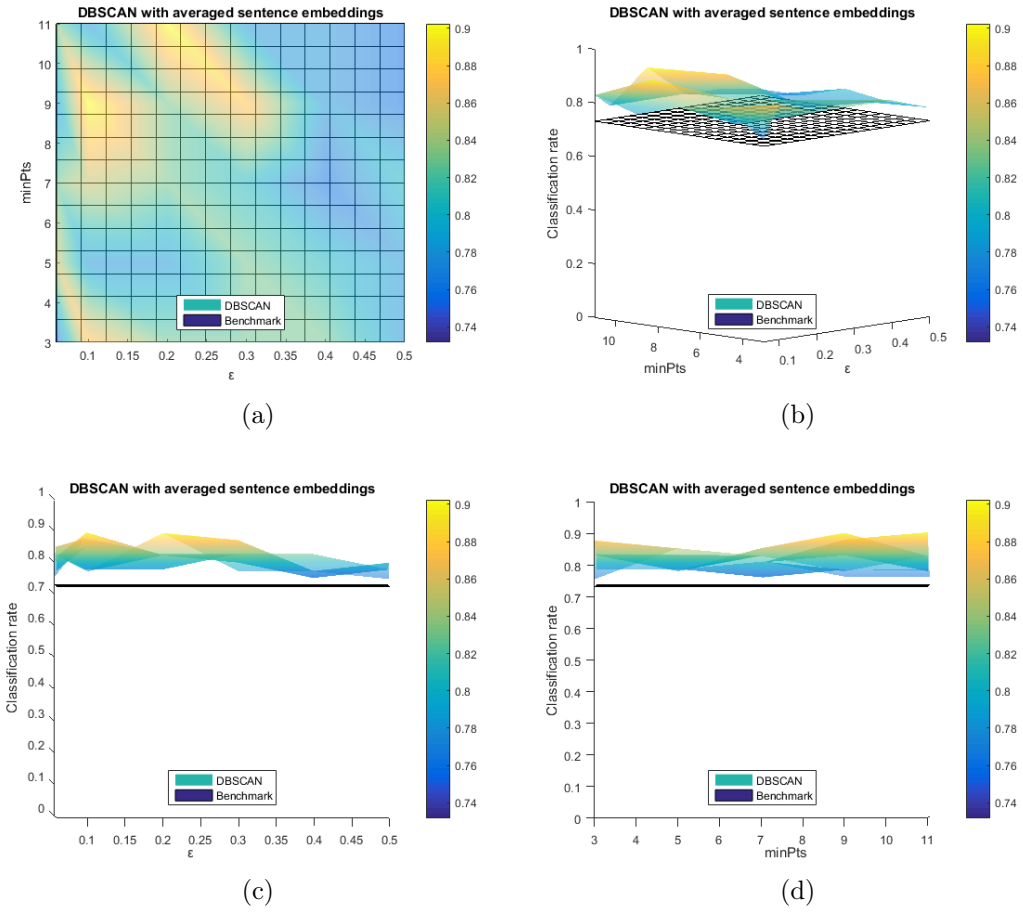


Figure 6.4: Classification rate of the DBN using the Averaged DBSCAN model

The DBSCAN algorithm has two hyperparameters, ε and minPts, which both have to be tuned. Thus, Figure 6.4 shows the classification rate of the DBN with a DBSCAN generated feature as a bivariate function of both hyperparameters. In Figure 6.4 the sentence embeddings input into the DBSCAN algorithm were computed as the average of each trader comment's word embeddings.

As with the case of k-means, the DBN classifier using DBSCAN outperforms the baseline model for all tested hyperparameter values. The best classification rate for this model is 0.902, which is achieved when $(\varepsilon, \text{minPts}) = (0.1, 9)$ and also $(\varepsilon, \text{minPts}) = (0.2, 11)$.

It seems as though the model performs well for large values of minPts and small values of ε (top left of Figure 6.4 (a)). This is interesting because that combination of parameter values corresponds to a model with a very high threshold for creating new clusters. If ε is small, then the ε-neighborhood checked for data density is small. Meanwhile, if minPts is large then a large number of data points must be found within the small ε-neighborhood in order for a new cluster to be initiated. If these parameters are set too extreme, then no new cluster will be created and all data points will be labeled as noise, making the feature useless. However, consider the fact that many trader comments are exactly identical. If two strings are identical, then they will contain the same words which will be mapped to the same word embeddings and

thus produce the same sentence embedding. If two sentence embeddings are equal, then they will be in each other's ε-neighborhood regardless of what ε is set to be. This means that DBSCAN can create clusters even if ε is very low so long as there exist groups of more than minPts number of identical trader comments. Indeed, looking at the cluster data for the parameter pair $(0.06, 11)$, the steepest threshold for cluster creation among the tested parameter values, the DBSCAN algorithm still managed to create 176 clusters.

In contrast, consider the opposite extreme case for DBSCAN parameters. If minPts is small and ε is large (bottom right of Figure 6.4 (a)) then large ε-neighborhoods will be searched and only a small amount of data points are required in order to start or expand a cluster. If these values are set too extreme then a single cluster will be able to spread and cover the entire dataset, thus labeling all data points identically and again making the feature useless. The results in Figure 6.4 show that the model performed poorly particularly for the case of large ε. By investigating the corresponding cluster data, it is found that for $\varepsilon = 0.5$ only one cluster was created regardless of the value of minPts. A few data points were labeled as noise but almost all were put in the single large cluster. This was a case of a single cluster enveloping most of the dataset due to the low threshold for spreading the cluster. Since almost all the data points have the same value for the trader comment feature there is very little information added with the inclusion of that feature. Indeed, the classification rates of the models with $\varepsilon = 0.5$ are nearly equal to the benchmark, confirming that the trader comment feature was unable to add much information in those cases.

Due to the nature of the data, a third extreme case is somewhat interesting to examine as well. Consider the case of both minPts and ε being small (bottom left of Figure 6.4 (a)). Given that the dataset contains many identical comments (making ε irrelevant for the creation of those clusters), having a small minPts and a small ε should lead to a situation where many clusters are created but are unable to expand. This is because there exist points with enough identical comments to exceed the minPts threshold and create a cluster, but due to the small ε-neighborhoods there are no additional neighbors to spread the clusters to. Examining the cluster data for the extreme parameter pair (0.06,3) confirms this hypothesis as this model spawned a staggering 665 distinct clusters. The performance of the DBN on so many clusters was poor (0.756).

SIF DBSCAN

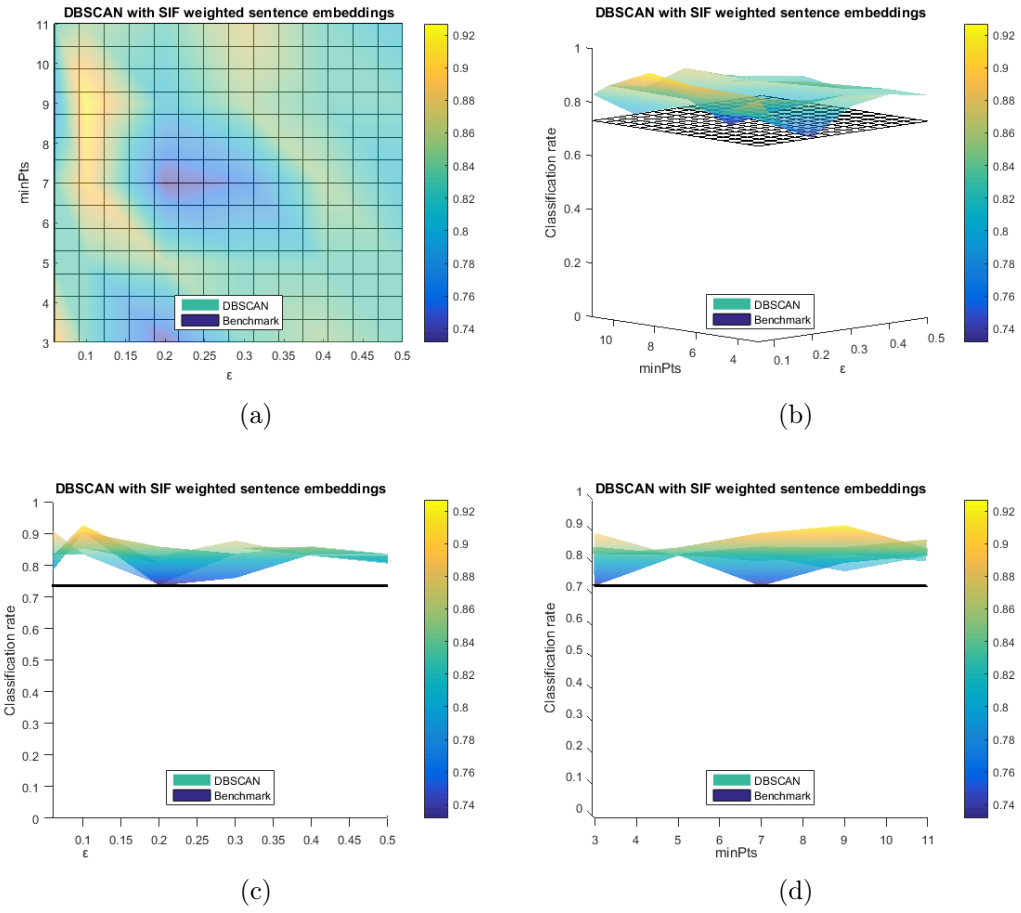
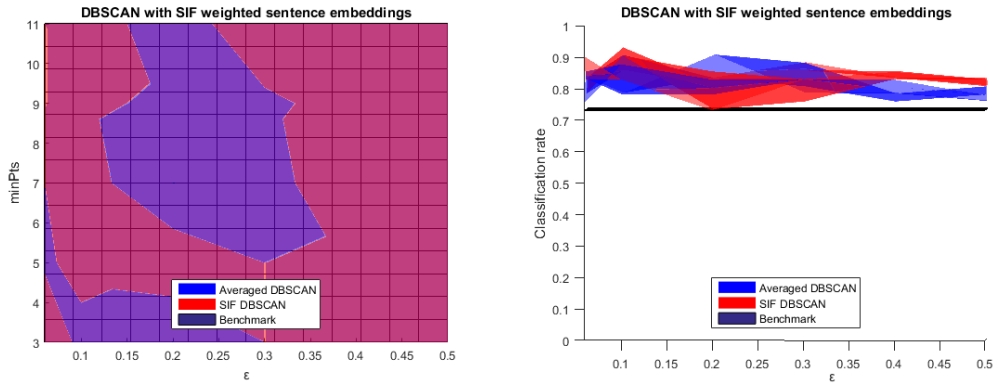


Figure 6.5: Classification rate of the DBN using the SIF DBSCAN model

Figure 6.5 also shows the classification rate of the DBN with DBSCAN features, this time using the SIF weighting scheme to generate sentence embeddings. Just as for the Averaged DBSCAN model in Figure 6.4, the classification rate using the SIF DBSCAN model is highest for low values of ε and high values of minPts. For the SIF DBSCAN, the highest classification rate is 0.927 which is obtained when $(\varepsilon, \text{minPts}) = (0.1, 9)$. Interestingly this is the same pair of parameter values for which the model in Figure 6.4 performs the best.

There is a region in the hyperparameter space in the center of Figure 6.5 (a) where the model performs very poorly, as poorly as the baseline model in fact. This is difficult to explain as anything other than the model in that region preferring a cluster partition which fits the classification purposes exceptionally poorly.



(a) (b)

Figure 6.6: Comparison of Averaged DBSCAN and SIF DBSCAN models

Figure 6.6 shows a comparison of the two DBSCAN-based models. In Figure 6.6 (a) one can see which of the two sentence embedding generation techniques leads to the higher classification rate for various points in the two-dimensional hyperparameter space. Blue areas correspond to the Averaged DBSCAN performing better and red areas to the SIF DBSCAN performing better. A majority of Figure 6.6 is red. The blue area mainly consists of the central parts of the figure for which the SIF DBSCAN model inexplicably performed very poorly.

Figure 6.6 (b) shows that the SIF DBSCAN model achieved both the highest and the lowest classification rate of the two DBSCAN models.

6.2 Results of Tuned Models

In Table 6.1 the classification rates of the tuned models on the tuning set are presented. Two of the models achieved peak classification rate for two different choices of parameter values. For further analysis a choice was made to continue with a single set of parameters for each model.

The Averaged k-means model achieved a classification rate of 0.902 for both $k = 90$ and $k = 130$. Since the heuristic k-means algorithm (see Algorithm 1) converges in $\mathcal{O}(nkpi)$ the choice is made to opt for the lower value of $k = 90$ with a nod to Occam's razor.

The Averaged DBSCAN model also achieved a classification rate of 0.902, for $(\varepsilon = 0.1, \text{minPts} = 9)$ and $(\varepsilon = 0.2, \text{minPts} = 11)$. Either choice would be sensible, however choosing $(\varepsilon = 0.1, \text{minPts} = 9)$ allows for a comparison of the Averaged DBSCAN and SIF DBSCAN models with the same parameter values which could be interesting.

Model name	Sentence embeddings	Clustering technique	Tuned parameters	Classification rate
Benchmark	N/A	N/A	N/A	0.732
Averaged k-means	Averaged	k-means	$k = 90$	0.902
SIF k-means	SIF Weighted	k-means	$k = 130$	0.878
Averaged DBSCAN	Averaged	DBSCAN	$\varepsilon = 0.1$, minPts $= 9$	0.902
SIF DBSCAN	SIF Weighted	DBSCAN	$\varepsilon = 0.1$, minPts $= 9$	0.927

Table 6.1: Model results on the smaller tuning set

After settling for a choice of parameters for each models, another round of testing was done on the full dataset containing all of the labeled data described in Section 4.2. The training time of the DBN on the full dataset was approximately six hours.

Model name	Sentence embeddings	Clustering technique	Tuned parameters	Classification rate
Benchmark	N/A	N/A	N/A	0.876
Averaged k-means	Averaged	k-means	$k = 90$	0.931
SIF k-means	SIF Weighted	k-means	$k = 130$	0.897
Averaged DBSCAN	Averaged	DBSCAN	$\varepsilon = 0.1$, minPts $= 9$	0.912
SIF DBSCAN	SIF Weighted	DBSCAN	$\varepsilon = 0.1$, minPts $= 9$	0.945

Table 6.2: Model results on the full dataset

Again, the DBN using the SIF DBSCAN model yields the highest classification rate, followed by the Averaged k-means model. Also, all of the models outperform the benchmark model on the full dataset as well. The spread between the best and the worst performing models in Table 6.2 is narrower than for the tuning set in Table 6.1. This can be explained by the fact that the lower amount of data in the tuning set causes a higher variance in results.

Comparing the results of Tables 6.1 and 6.2 one can see that the order of the models from best to worst classification rate is the same for both datasets (save the fact that the Averaged k-means and Averaged DBSCAN models achieve equal classification rate on the tuning set). This is excellent because it adds reliability to the hyperparameter tuning done on the smaller dataset. Furthermore it should be noted that every model achieved a higher classification rate on the full dataset compared to the tuning set. This is a neat observation and a nice confirmation of the scalability with data of neural network classifiers.

Chapter 7

Discussion

In this chapter, the results from Chapter 6 will be analyzed and discussed. After that, the research questions from Chapter 1 will be brought back up and evaluated. Finally, some suggestions for future research directions will be given.

7.1 Analysis of Results

The results from Chapter 6 show convincingly that the inclusion of a trader comment feature does improve the predictive ability of the deep belief network classifier. Indeed, not a single model achieved a classification rate lower than the benchmark model, even for the worst performing choices of hyperparameter values.

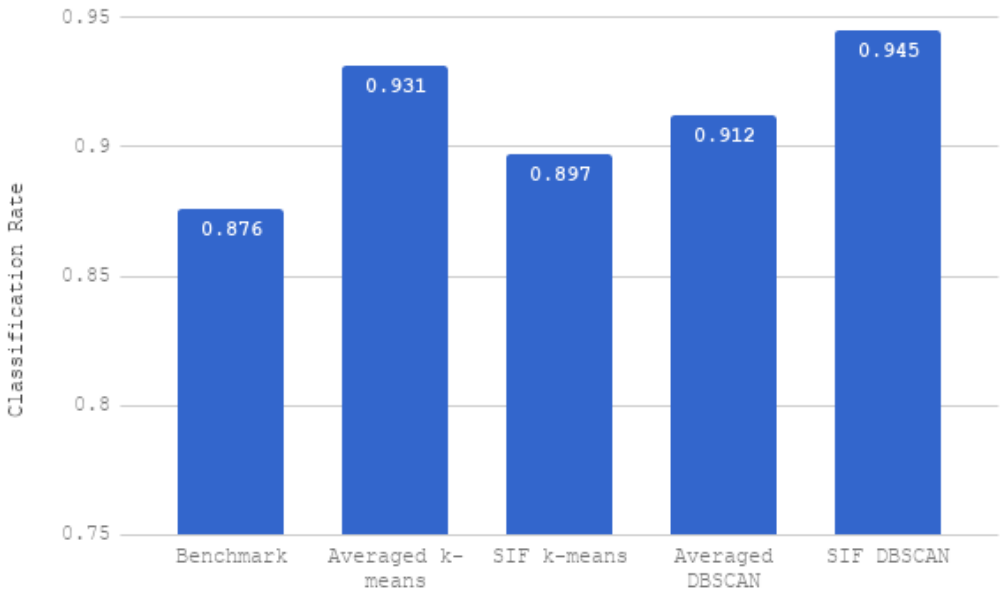


Figure 7.1: Comparison of different models' classification rates on full dataset

On both the tuning set and the full dataset, the model with SIF DBSCAN trader comment features performed the best, followed by the model with Averaged k-means. It's interesting that two opposite combinations of sentence embedding generation technique and clustering algorithm performed the best. If it were the case that

for instance the SIF weighting scheme was a fundamentally superior technique for sentence embedding generation compared to averaging, then one might have expected the two SIF models to outperform the two averaged models. The same holds for the case of either clustering algorithm being superior. However, the results show that this is not the case. Rather, in this experiment the k-means algorithm is superior for averaged sentence embeddings but not for SIF weighted sentence embeddings.

The cause of this occurrence is unknown, however as discussed in Section 6.1 there might be some intuitive and empirical backing for the hypothesis that the SIF weighting scheme disrupts some of the linearity in the word embedding data to the benefit of the DBSCAN algorithm. If the SIF weighting scheme contorts the relationships between word embeddings in a slightly non-linear fashion but which more effectively captures the information contained in the sentence then this could explain the empirical results. The k-means algorithm would struggle to model such non-linear clusters since it partitions the dataset into linearly separable Voronoi cells. Meanwhile the DBSCAN algorithm is perfectly capable of discovering abstract non-linear cluster shapes. The results shown in Figure 7.1 support this hypothesis as the SIF k-means model had the worst performance out of the four models which used a trader comment feature. Meanwhile, the SIF DBSCAN model had the best performance out of all models.

On the set of averaged sentence embeddings, where the linear vector relationships have not been altered, the k-means algorithm should perform well. If the optimal cluster partition indeed is a linear one, then one would expect the DBSCAN algorithm to also fit (nearly) linearly separable clusters. The DBSCAN algorithm would not benefit from its ability to fit non-linear cluster structures. Rather, it would run a risk of overfitting a suboptimal non-linear cluster structure to the data. Again, strong empirical support for this hypothesis is found in the results. Figure 7.1 shows that the two models with averaged word embeddings were much closer in classification rates than the two SIF models. In fact, the results in Table 6.1 show that on the tuning set the two clustering algorithms achieved the exact same classification rates when clustering the averaged sentence embeddings. It seems plausible that the clusters they fit to that data were very similar. In fact, looking at the cluster data from the Averaged DBSCAN (with tuned parameters $\varepsilon = 0.1$, minPts $= 9$) it turns out that the algorithm fit 103 clusters to the data. This is quite similar to the 90 clusters fit by the Averaged k-means model where k was tuned to be 90. On the full dataset, the Averaged DBSCAN model performed slightly worse than the Averaged k-means. Again, this is consistent with the hypothesis.

7.2 Reliability of Results

Due to constrained resources the models had to be tuned on a smaller subset of the full dataset. This creates some uncertainty regarding the reliability of these models. Fortunately, the results of the models using the full dataset, shown in Table 6.2, show strong consistencies with the results of the tests on the tuning set. The models rank the same way in order of performance for both the tuning set and the full dataset. This is encouraging, since it reassures that the relative performance of the models on the smaller tuning set was consistent with the full dataset and not distorted by some local anomalies present in the tuning set.

Although the results can be considered reasonably reliable, it should be noted that given more time and computing power the more reliable methodology would

have been to tune the models on the full dataset, preferably using a cross-validation scheme. This could potentially also have allowed for the testing of other types of models as well.

7.3 Concluding Remarks

The research questions from Chapter 1 will now be assessed based on the results obtained.

- *How can word embeddings be employed to represent short text strings as numerical features with minimal information loss?*

The algorithm described in Chapter 5 is an attempt to do exactly what is described in this research question. The word embeddings are combined to form a single sentence embedding for each text string. The sentence embeddings are then clustered to create the numerical features specified.

Multiple degrees of freedom in the algorithm, such as choice of sentence embedding generation technique and clustering algorithm, allow for a high level of flexibility so that the model produced can be fit to the data effectively.

- *What is the predictive impact of including such features?*

All models tested which included a trader comment feature performed equally or (in most cases) better than the benchmark model with no trader comment. Given this, it seems fair to conclude that the predictive ability of the DBN classifier assuredly was improved by the inclusion of the trader comment feature.

This means that by including the trader comment feature, the DBN can more effectively classify the level of operational risk in the trade modifications. Exactly what this means in monetary terms is very difficult to determine, as with almost all cases of operational risk. Regardless, it is certainly beneficial to be able to improve the classification accuracy of an operational risk classifier.

7.4 Future Research

Non-linear Vector Relations of SIF Weighted Sentence Embeddings

There seems to be strong empirical evidence in this thesis supporting the hypothesis that the SIF weighting scheme in some way contorts some of the linear vector relations between (`fastText`) word embeddings. The k-means clustering algorithm performs well on the dataset of averaged sentence embeddings (where supposedly the vector relations are kept linear) but performs poorly on the dataset of SIF weighted sentence embeddings (where supposedly they are not). The DBSCAN algorithm, which unlike k-means can fit non-linear cluster structures, performs much better on the SIF weighted embeddings.

A study of this phenomenon could lead to greater insights regarding the SIF weighting scheme and word embeddings in general.

Other Applications

In this thesis the algorithm from Chapter 5 was tested on a specific case concerning operational risk in banking. However, the algorithm is very general and can be applied to any situation where one wishes to convert a free-text into a numerical data feature. Thus it would be interesting to study the performance of the algorithm on other datasets from various industries.

Particularly, it would be interesting to study how the algorithm performs on datasets where the free-texts differ a lot from the ones in this study. This could for instance mean a case where the texts are longer or more diverse. It would also be interesting to study a case with a larger availability of labeled data.

Further Exploration of Models

There are many degrees of freedom in this thesis which can be tweaked to produce new alternative and potentially superior models. Things which could be altered to create new models are:

- **Pre-trained word embeddings.** In this thesis, `fastText` word embeddings were used. However, many alternatives exist, such as `word2vec`, `Caffe` and `GloVe`.
- **Sentence embedding generation techniques.** This thesis explores two ways of generating sentence embeddings: averaging and SIF weighting. Other weighted average schemes or more advanced methods could be explored as alternatives.
- **Clustering algorithms.** In this thesis the k-means and DBSCAN algorithms were used. There exist many other unsupervised learning algorithms which could potentially perform well.
- **Classifiers.** The algorithm in this thesis was evaluated using a deep belief network but is designed to work with any classifier. Thus, testing its performance with other classifiers would be interesting.

Bibliography

[1] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine learning*, 75(2):245–248, 2009.

[2] Sanjeev Arora, Yuanzhi Li, Yingyu Liang, Tengyu Ma, and Andrej Risteski. A latent variable model approach to pmi-based word embeddings. *Transactions of the Association for Computational Linguistics*, 4:385–399, 2016.

[3] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. A simple but tough-to-beat baseline for sentence embeddings. 2016.

[4] David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 144–153. ACM, 2006.

[5] Zaw Zaw Aung and Kenji Watanabe. A framework for modeling interdependencies in japan's critical infrastructures. In *International Conference on Critical Infrastructure Protection*, pages 243–257. Springer, 2009.

[6] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[7] Edilson Anselmo Correa, Vanessa Queiroz Marinho, and Leandro Borges dos Santos. Nilc-usp at semeval-2017 task 4: A multi-view ensemble for twitter sentiment analysis. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 611–615, 2017.

[8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[9] George Forman. Bns feature scaling: an improved representation over tf-idf for svm text classification. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 263–270. ACM, 2008.

[10] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[11] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[12] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[14] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[15] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[16] Sofus A Macskassy and Haym Hirsh. Adding numbers to text classification. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 240–246. ACM, 2003.

[17] Sofus A Macskassy, Haym Hirsh, Arunava Banerjee, and Aynur A Dayanik. Converting numerical classification into text classification. *Artificial Intelligence*, 143(1):51–77, 2003.

[18] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[19] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. Advances in pre-training distributed word representations. *arXiv preprint arXiv:1712.09405*, 2017.

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[21] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *hlt-Naacl*, volume 13, pages 746–751, 2013.

[22] Basel Committee on Banking Supervision. Basel iii: Finalising post-crisis reforms, 2017.

[23] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[24] Joaquín Pérez, Rodolfo Pazos, Laura Cruz, Gerardo Reyes, Rosy Basave, and Héctor Fraire. Improving the efficiency and efficacy of the k-means clustering algorithm through a new convergence condition. In *International Conference on Computational Science and Its Applications*, pages 674–682. Springer, 2007.

[25] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[26] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

[27] Andrea Vattani. K-means requires exponentially many iterations even in the plane. *Discrete & Computational Geometry*, 45(4):596–616, 2011.

[28] Sholom M Weiss, Nitin Indurkhya, and Tong Zhang. From textual information to numerical vectors. In *Fundamentals of Predictive Text Mining*, pages 13–38. Springer, 2010.

[29] Yuqian Xu, Michael Pinedo, and Mei Xue. Operational risk in financial services: A review and new research opportunities. *Production and Operations Management*, 26(3):426–445, 2017.

Appendix A

Implementing pre-trained `fastText` & `word2vec` models

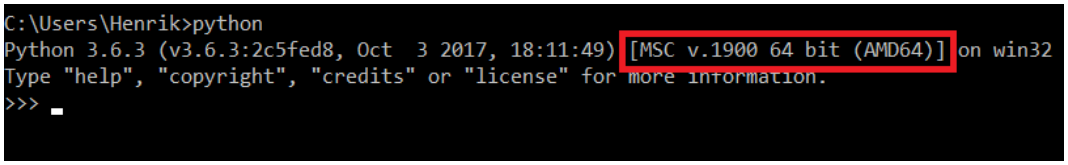
This section explains how to obtain and implement pre-trained word embeddings and their associated packages. Note that the `fastText` library is under heavy development as of the time of writing this report (May 2018). Thus, the instructions in this section may be outdated. It is however unlikely that the packages have been made less user friendly with further development.

Installing fastText

The current state-of-the-art in word embeddings is the `fastText` library, developed by Facebook AI Research. It is open-source and can be cloned straight from GitHub. The original C++ code can only be built on Mac OS and Linux systems. Several options do however exist for those wanting to use `fastText` in a Windows environment. In November 2017, the developers of `fastText` pushed Python bindings for `fastText`. These can be used to run `fastText` in Python scripts and is a convenient way to use `fastText` on a Windows machine.

Dependencies

- Python 2.7 or Python 3.6 (or newer)
- A C++ compiler with C++11 support. This must be the same compiler version as your Python install was compiled with. On Windows, Python typically comes pre-compiled on Microsoft Visual C++. You can find the version of VC++ that your Python install was compiled with by writing `python` in the command prompt.



(a) Check which version of VC++ your Python install was compiled with

```
MSC v.1000 -> Visual C++ 4.x
MSC v.1100 -> Visual C++ 5
MSC v.1200 -> Visual C++ 6
MSC v.1300 -> Visual C++ .NET
MSC v.1310 -> Visual C++ .NET 2003
MSC v.1400 -> Visual C++ 2005  (8.0)
MSC v.1500 -> Visual C++ 2008  (9.0)
MSC v.1600 -> Visual C++ 2010 (10.0)
MSC v.1700 -> Visual C++ 2012 (11.0)
MSC v.1800 -> Visual C++ 2013 (12.0)
MSC v.1900 -> Visual C++ 2015 (14.0)
MSC v.1910 -> Visual C++ 2017 (15.0)
```

(b) Mapping of MSC v.X codes to actaul VC++ versions

Figure A.1: Mapping VC++ version number from Python compilation to actual version ID

With the help of the table above, you can find the name of the VC++ version you want. Again, it must support C++11. so you need VC++ 2012 or later. If your installation of Python was compiled using an earlier version of VC++ you should download a newer version of Python. When you have figured out which VC++ version you need, find and download the corresponding VC++ Build Tools (not the redistributable).

- `NumPy, SciPy, Cython, Pybind11`. These are Python libraries which can be installed using pip.

Furthermore, the following are recommended:

- A 64-bit machine
- At least 8 GB of RAM

Installation

To install `fastText`, first clone the `fastText` repository from GitHub. Then run the `setup.py` script. This will compile and install the library. The easiest way to do this is through the command prompt:

```
$ git clone https://github.com/facebookresearch/fastText.git
$ cd fastText
$ python setup.py install
```

The `fastText` library can then be imported into Python code by `import fastText`.

Using pre-trained models

For most purposes, it is not necessary to train a new word embedding model for every application since the underlying language has the same general semantics. Training models on extremely large corpora is much more computationally intensive

than actually using the model. Luckily, there exist pre-trained open-source `fastText` models of various sizes and languages.

`fastText` models for languages other than English are typically trained on Wikipedia articles. The `fastText` developers have release large pre-trained models in many languages.[1] The models come in two different file formats, `.vec` and `.bin`. The `.vec` file is a text file in which each row is a word followed by its word vector. The `.bin` file is a binary format native to the `fastText` library. If you are using the official `fastText` library to load the models, you should always use the binary file as it loads significantly faster (seconds compared to minutes). Use `.vec` files only if you are using a different library than `fastText` to load the models (such as `Gensim`), or if you are using a pre-trained model which does not exist in a `.bin` format.

The only drawback to using the official `fastText` pre-trained models from Facebook AI Research is that the models are very large which may not be practical for all purposes. The Swedish `fastText` model in `.bin` format is roughly 5 GB which means your machine must be able to efficiently load 5 GB into the RAM. This model contains ~ 1.1 million words, the majority of which are probably superfluous for your applications. An alternative is to use third-party models of smaller size. Initial testing of the algorithm in this thesis was done with a smaller model.[2] This model contains $\sim 50,000$ words, which is plenty for most testing purposes. The file size is a few megabytes, so the model loads instantly.

Working around the `fastText` library

If for some reason you are not able to use the `fastText` library you can still load and use the pre-trained models. The pre-trained models are essentially just words and their associated vectors, so many other libraries are well-equipped to load them. An example is the `Gensim` library, a robust and well-established library for word embeddings which predates both `fastText` and `word2vec`. `Gensim` can be installed using pip, so the setup is much quicker than for the `fastText` library.

Since `Gensim` is a more established library, it contains many useful built-in functionalities which are not yet implemented in the `fastText` library. For example, when loading a pre-trained model one can set a limit for the number of words to load from the model. The official `fastText` models are ordered by frequency, so by loading such a model and setting a limit, e.g. $50,000$ words, the program will load the $50,000$ most frequently appearing words rather than all 1.1 million words. This of course significantly improves loading times for testing purposes.

The main drawbacks of using the `Gensim` library rather than the `fastText` library is that you lose the ability to train your own models, and load times are generally slower. The reason for the load times being slower is because `Gensim` currently does not support the `fastText .bin` format for pre-trained models and as such the models must be loaded from the `.vec` files. In testing, the loading time for the 1.1 million word `fastText` model using the `fastText` library and `.bin` file was a few seconds. The loading time for the same model, but with the `Gensim` library and `.vec` file was about 30 minutes, so the difference is significant.

[1] Pre-trained `fastText` models in 294 different languages: https://fasttext.cc/docs/en/pretrained-vectors.html

[2] Smaller pre-trained `fastText` models in 30+ languages: https://github.com/Kyubyong/wordvectors

	`fastText`	`Gensim`
Installation & setup	Tedious on a Windows machine	Quick and easy using pip
Train new models	Yes	No
Load `.vec` models	Yes	Yes
Load `.bin` models	Yes	No
Loading time	Short	Long
Functionalities	Limited	Many interesting features for data exploration

Table A.1: Comparison of the `fastText` and `Gensim` libraries

The table above illustrates the advantages and disadvantages of the `fastText` library compared to third-party libraries such as `Gensim`. In general, use `Gensim` for light-weight tasks such as initial testing and use the `fastText` library for optimal performance on larger tasks.

Appendix B

Visualizing and Analyzing Trader Comment Clusters

The output of the algorithm described in Chapter 4 is a number of clusters. Each text comment in the input belongs to one and only one cluster. In this section we look closer at the results of one particular run of the algorithm. This section is purely meant as a means for gaining more insight into the clustering results. As such, not everything will be directly relevant for the purposes of using the results of the clustering to perform classification. The results presented here are for the k-means algorithm with $k = 10$, i.e. ten clusters.

Visualizing the Clusters

The sentence embeddings which are clustered are 300-dimensional and thus cannot be meaningfully visualized as such. Instead, by performing principal component analysis on the set of sentence embeddings the dimension can be reduced to 2 or 3 dimensions for visualization purposes.

This makes it possible to visualize the k-means clustering in a two- or three-dimensional plot to some degree. It should of course be noted that 300-dimensional relations cannot be perfectly modeled in two or three dimensions. Some information will be lost when reducing the dimensionality.

Figures B.1 and B.2 show the ten clusters when reduced to two and three PCA dimensions. Each color corresponds to a unique cluster.

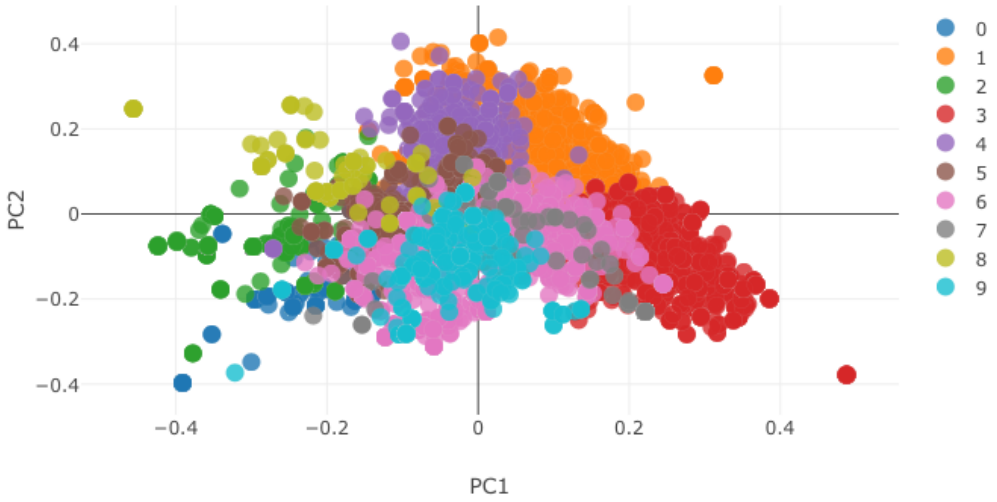


Figure B.1: PCA with two principal components and ten clusters

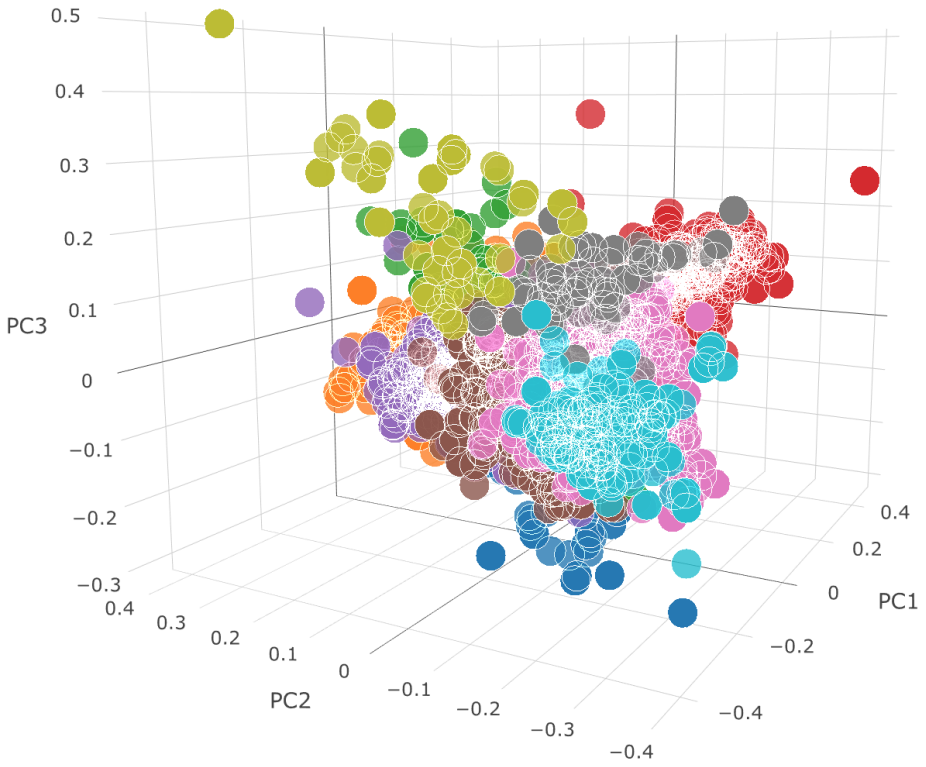


Figure B.2: PCA with three principal components and ten clusters

Trader Analysis

The dataset contains (anonymized) information about which trader made each trade modification. In Figure B.3 the three traders with the highest number of modifications made in the dataset are studied to see whether their comments are distributed similarly.

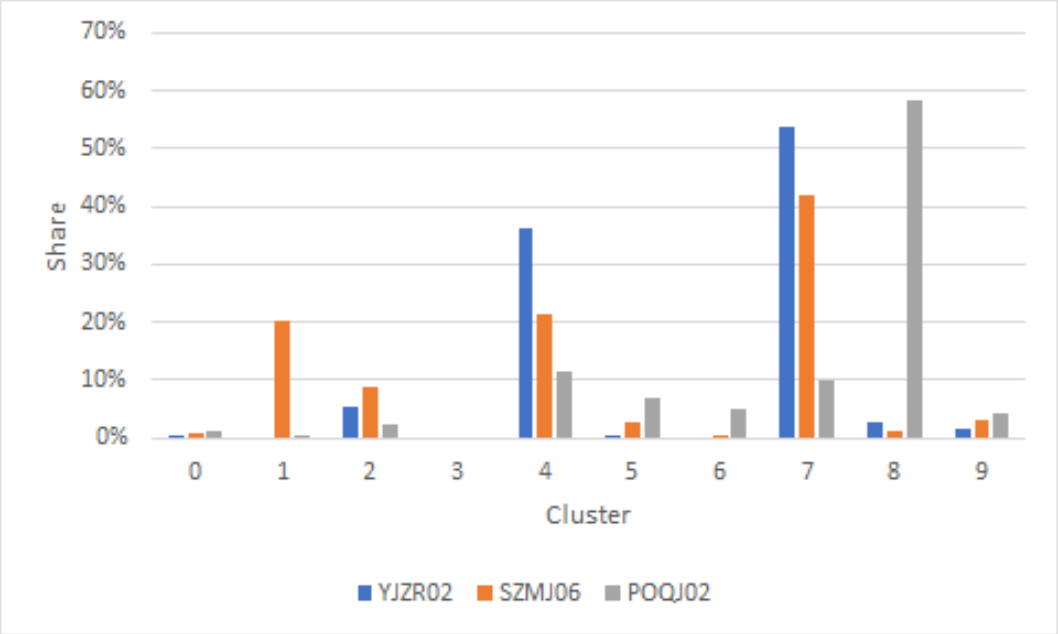


Figure B.3: Cluster distribution of comments made by three traders

Figure B.3 shows each of the three traders and how their comments were clustered. Again, the trader ID's YJZR02, SZMJ06 and POQJ02 are **not** the actual ID's of these traders. There are some clear differences between the traders' distributions of comments. The most dramatic is that trader POQJ02 had more than 50% of their comments end up in Cluster 8 whereas the other two traders had almost no comments in Cluster 8. In Cluster 7 the roles were reversed with POQJ02 having fewer comments than the other two traders.

There are two natural explanations for this. Firstly, different traders may have different areas of responsibility which can be reflected in their comments. For instance, Cluster 1 largely contained changes to trades regarding the split rates. Trader SZMJ06 made a fair amount of such comments, but the other two traders made very few or none. It is reasonable to suspect that trader SZMJ06 has work duties relating to split rates whereas the other two traders do not. Secondly, different people tend to write slightly differently compared to other people. If two traders make different word choices when they document trades, then those comments could end up in different clusters for that reason alone.

TRITA -SCI-GRU 2018:167