Linköping University | Department of Computer and Information Science
Bachelor's thesis, 18 ECTS | Language Technology
2019 | LIU-IDA/KOGVET-G--19/025--SE

A Study on Text Classification Methods and Text Features

Benjamin Danielsson

Supervisor : Arne Jönsson
Examiner : Robert Eklund

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

When it comes to the task of classification the data used for training is the most crucial part. It follows that how this data is processed and presented for the classifier plays an equally important role. This thesis attempts to investigate the performance of multiple classifiers depending on the features that are used, the type of classes to classify and the optimization of said classifiers. The classifiers of interest are support–vector machines (SMO) and multilayer perceptron (MLP), the features tested are word vector spaces and text complexity measures, along with principal component analysis on the complexity measures. The features are created based on the Stockholm–Umeå-Corpus (SUC) and DigInclude, a dataset containing standard and easy–to–read sentences. For the SUC dataset the classifiers attempted to classify texts into nine different text categories, while for the DigInclude dataset the sentences were classified into either standard or simplified classes. The classification tasks on the DigInclude dataset showed poor performance in all trials. The SUC dataset showed best performance when using SMO in combination with word vector spaces. Comparing the SMO classifier on the text complexity measures when using or not using PCA showed that the performance was largely unchanged between the two, although not using PCA had slightly better performance.

Keywords: NLP, text classification, SVM, MLP, PCA, SUC, DigInclude

Acknowledgments

This bachelor's thesis would not have been possible without the help and support several people. First and foremost, I wish to thank my supervisor Arne Jönsson for his help, guidance and feedback during the process of writing the thesis. I also extend my gratitude to Marina Santini who has gone above and beyond in her assistance and advice in formulating my research question, methodology and best practice on how to complete the thesis. Her aid has been invaluable.

I also wish to express my gratitude to the examiner Robert Eklund, both for this help on creating a better paper and his thoughts on typography. Lastly, I also give my thanks to the five other students in our study group where we have assisted each other during the writing process.

Contents

List of Tables

1 Introduction

Natural language processing (NLP) is the field of study that interests itself with the problems and solutions for computers to understand human language. Human language is a complex cognitive system which engages multiple parts of the brain, something not yet fully replicable through the use of computers, however, by creating models and complex computer systems we can try to emulate parts of language and text. With the help of different text classification methods, the goal of the study is to investigate the performance of classifiers when applied to classifying texts into different classes, namely, text categories and text complexity. This could have wider implications for other classification tasks and show the importance of choosing an appropriately processed dataset and features in order to maximize the classifiers performance.

The data used are based on different types of features, the characterization of data. The features used are text complexity measure and word vector spaces, these features are presented to the classifiers instead of the raw text data. The text complexity features attempt to map the text to units that represent the text's readability. Previous research in text complexity measures has created 117 measure which are used for the classifications in this thesis (Falkenjack, Mühlenbock, & Jönsson, 2013, p. 2). The second feature used, word vector spaces, represent the words in a vector space, where words that frequently appear next to other words keep that representation in the vector space in an attempt to maintain the semantics of the text (Jurafsky & Martin, 2018, pp. 118–123).

Principal component analysis (PCA) is also performed to investigate how well the original text complexity measures can be condensed while not disturbing the performance.

1.1 Purpose and Problem Formulation

The purpose of the study is to compare different text classification methods and features and weigh the performance of the created models based on their ability to classify texts to its correct classes, in this case, text category and a binary classification task with classes based on text complexity. The classes chosen are not necessarily of great importance for the thesis, but rather a means to investigate the impact on performance depending on the classes which are used. The classification methods that will be used are *support–vector machine* (SVM) and *multilayer perceptron* (MLP), along with features based on word vectors, PCA and text complexity features. Essentially, the aim is to investigate how the classification performs based on what data is used, how that data is processed, what the classes are, which method chosen, and how the methods are optimized.

Furthermore, two classifiers compared side–by–side can both be better than the other based on the parameters used. With this in mind, optimization of the parameters is one of the first objectives of the study, only then will the results be compared. First and foremost, this comparison between results will be done through the average weighted f-measure of the models, which is a measure of accuracy using precision and recall. All of the tests will be run on multiple sets of data.

1.2 Scope and Limitations

The study aims at investigating the performance of classification methods and the features used for said classification, however the scope of the study limits itself to Swedish corpora and not necessarily all languages in general. The corpora in question are *Stockholm – Umeå Corpus*, a corpus containing a variety of annotated Swedish texts, and *DigInclude* which includes standard Swedish sentences and corresponding simplified version of said sentences.

2 Background

This section will cover the general knowledge required to fully understand the study and its methodology. Explanations for the classifiers, features and the software used to carry out the experiments are given. The software, WEKA, was used to perform the classification and the parameters modified are also explained under the relevant subsections. The different procedures and methods relevant to the study are described, such as to classify or to create word vector spaces.

2.1 SVM and SMO

Support vector machines, or SVM for short, is a method of machine learning that takes an algebraic approach and uses algorithms to create classification models. Although there are different types of SVMs, they tend to aim to find the maximum margin hyperplane. A hyperplane can be imagined as the line which best separates the classes in the training data. An SVM can handle nonlinear data since they map the original dimensional problem space into a higher dimensional space, making the separation easier (Witten, Frank, Hall, & Pal, 2017, pp. 274–276). The training of an SVM consists of calculating and finding the maximum margin hyperplane, while fitting the SVM is an optimization problem and can be done through different methods (Falkenjack, 2018, p. 12, 28).

The optimization of fitting data to a SVM is known as a *constrained quadratic optimization problem*, which is a standard type of optimization task (Witten, Frank, Hall, & Pal, 2017, p. 276). John Platt (1999) created an algorithm for solving such optimization problems called *sequential minimal optimization* (SMO). SMO has since been a popular tool to use alongside

SVM when classifying nonlinear data, for the sake of brevity when SVM and SMO is used in this thesis only SMO will be mentioned. Platt's original implementation has since its creation been found to be slow, modern implementations use clever techniques to cut the runtime, like caching frequent calculations to avoid repetition (Rifkin, 2002, p. 48).

The software WEKA implements SVM with SMO by default and supports an array of different parameters for modification (Frank, Legg, & Inglis, n.d.). One of the more important parameters is the *complexity constant* which controls the trade–off between how important correctly classifying the instances is versus allowing outliers of incorrectly classified instances in the training. Since an SVM attempts to find a hyperplane in the dataset to classify the classes, the complexity constant modifies how the hyperplane is determined. A high value will prioritize drawing the hyperplane to not include incorrectly classified instances, which in turn means the hyperplane's margin is decreased. One may incorrectly surmise that as many correctly classified instances as possible is good and therefore increasing the complexity constant improves performance, but this is not the case. By increasing the constant, the risk of overfitting is greatly increased, likewise, a low complexity constant increases the risk of underfitting (Frank, Legg, & Inglis, n.d.).

A second parameter of importance is the choice of kernel to use. In the context of a SVM, a kernel is essentially a similarity function which calculates dot products of vectors. In SVMs a kernel is used to exploit the *kernel trick* which allows the SVM to classify non-linear data. Kernels can operate in high dimensionality because instead of relying on the coordinates of the datapoints they instead compute using dot product of the support vectors. The general idea behind the method is that mapping the original dimension into a higher dimension allows the data to become linearly separable. There are multiple kernels that can be used, traditionally a kernel called *RBF Kernel* has been widely used (Chang, Hsieh, Chang, Ringgaard, & Lin, 2010, p. 1472), however, in natural language processing it has been shown that *Polynomial Kernels* tends to perform the best (Chang, Hsieh, Chang, Ringgaard, & Lin, 2010, p. 1472; Goldberg & Elhadad, 2008, p. 240)

2.2 Multilayer Perceptron

Multilayer perceptron (MLP) is a type of neural network that expands on single-perceptrons in order to enable classification of nonlinear data. By themselves a simple perceptron can only separate linear data, but by interconnecting several nodes split into at least three layers (*input layer*, *hidden layer* and *output layer*) and equipping the nodes with a nonlinear activation function the combination of nodes enable complex classification such as nonlinear data (Witten, Frank, Hall, & Pal, 2017, pp. 283–285).

Training an MLP consists of continually updating the weights of the nodes given the error of the output nodes and the influence each node had on said output, typically called *backpropagation*. By knowing the desired output given an input the nodes are updated backwards, the function used to change weights are different depending on design but often a gradient descent is used. Gradient descent uses derivates, meaning the function must be differentiable. Using derivates allows the MLP to take advantage of calculating the slope of the function and can therefore increase or decrease the weights by an amount relative to the amount of error. Weights that are in need of great change does so, while weights that only need slight change are changed by a small amount (Witten, Frank, Hall, & Pal, 2017, pp. 286–288).

WEKA supports the implementaion of MLPs and the modification of parameters (Ware, n.d.). *Learning rate*, *momentum*, *batch* size, *training time*, validation set size, validation threshold, and hidden layer setup are the parameters of importance. Other parameters exists but these do not affect the performance, such as logging debugging information or displaying a GUI with real–time information of the training.

The learning rate is the value that backpropagation multiplies with the value of the derivate, the product is subtracted from the function's value. Higher learning rate means that the iterative process of backpropagation makes bigger leaps per iteration. In WEKA, this value is a decimal between 0 and 1 (Ware, n.d.; Witten, Frank, Hall, & Pal, 2017, pp. 287–288). Momentum is a parameter that behaves similarly to the learning rate, however, where learning rate is a unchanging number the momentum is based of the previous iteration. This enables the the search process to be less abrupt and makes changes in proportion to the previous iterations (Witten, Frank, Hall, & Pal, 2017, p. 291).

The batch size determines the number of instances to process at a time, essentially creating subsets of the training set called *mini-batches* (Witten, Frank, Hall, & Pal, 2017, p. 455). If the batch size is set to the total number of instances no mini-batches are created and the MLP behaves as normal (Ware, n.d.). By setting a batch size smaller than the total number of instances available the classifier will update its weights based on the average gradient across the mini-batches. All mini-batches are processed in turn per iteration of the backpropagation loop, once all batches have been processed the weights are updated and the next iteration begins (Witten, Frank, Hall, & Pal, 2017, p. 455). Research has shown that using mini-batches can reduce the time taken for the training while retaining or slightly decreasing the performance of the created model (Tibell, 2015, p. 47, 55).

Training time, validation set size, and validation threshold are interwoven and dependent on each other. The training time parameter sets the number of epochs the classifier is allowed to run before stopping the training, where one epoch is defined as one full iteration of the

dataset. In the case where mini-batches are used, an epoch has been completed when all mini-batches have been iterated over (Ware, n.d.). Once the number of epochs have been iterated the training is complete. That being said, stopping an MLP classifier based on an arbitrary[1] number can have issues with underfitting and overfitting the data (Witten, Frank, Hall, & Pal, 2017, p. 453). Instead, *early stopping* is employed to stop the training once a certain criterion is met. First, the validation set size is determined. WEKA accepts a integer between 1 and 100 which defines the percentage of the training set that is preserved to a validation set. Secondly, the validation threshold is added, which dictates how many times in a row the validation set's error can get worse before the training is stopped (Ware, n.d.; Witten, Frank, Hall, & Pal, 2017, pp. 453–534).

The hidden layer parameter defines the number of nodes and hidden layers used for the MLP. In WEKA, the parameter accepts a list of positive integers where each number represents a hidden layer and the value of the number represents the number of nodes contained within said layer (Ware, n.d.). As an example, if the parameter is set to '245' three layers are created with two, four and five nodes respectively. The software also has four predefined wildcards that can be used instead of a number, namely, 'i', 'o', 't', and 'a' (see table 2.1). The first, 'i', creates a layer with a number of nodes equal to the number of features in the dataset. The second, 'o', equals the number of classes in the dataset. The third, 't' equals the sum of features and attributes. The last, 'a', equals the sum of features and attributes divided by 2.

Table 2.1: Hidden layer's wildcard definitions

Wildcard	Function
i	No. of features
o	No. of classes
t	(No. of features + No. of classes)
a	(No. of features + No. of classes) / 2

2.3 Principal Component Analysis

A principal component analysis (PCA) is typically performed in order to evaluate whether few features account for a large proportion of the variation in the data (Mühlenbock, 2013, p. 74). In a dataset with k features the datapoints are set in k-dimensional space. Regardless of the coordinate system used in said dimensional space, the value of the data lies in the datapoint's variance in each direction (Witten, Frank, Hall, & Pal, 2017, p. 327). With this in mind, PCA creates a new coordinate system based on the datapoints in order to reduce

[1] Arbitrary in the sense that it is not dependent on the information or statistic from the current running classifier, but rather on what the user has deemed acceptable.

dimensionality. It starts by defining a new first axis, called *component*, in the direction that maximizes the variance of points along that axis. The second component is also placed to maximize variance, however, it must be perpendicular to the first but it is allowed to be placed anywhere along that plane. The rest of the axes are placed in accordance to the second, i.e. always perpendicular to the first component and maximizing variance (Witten, Frank, Hall, & Pal, 2017, p. 327). By creating new components that maximize variance, the goal is to find a lower number of axes than what was originally present while still being able to cover a similar amount of variance. In essence, PCA defines new variables based on a combination of the original features, creating new features that retains a similar performance to the original larger number of features (Mühlenbock, 2013, p. 74; Witten, Frank, Hall, & Pal, 2017, p. 327).

The WEKA software supports three parameters, *variance covered*, *maximumAttributes*, and *maximumAttributeNames* (Hall & Schmidberger, n.d.). Only variance covered and maximumAttributes affects the performance of the PCA, while the maximumAttributeNames parameters affect how the output is presented within the WEKA software. Variance covered accepts a decimal between 0 and 1, signifying the variance needed before the analysis is considered completed. The variance will always approach 100% since PCA creates new axes where each axis maximizes variance. The PCA stops once the value of the variance covered parameter is met. The maximumAttributes defines the number of components to keep once the PCA is stopped (Hall & Schmidberger, n.d.).

2.4 Word2Vec

There are multiple ways to create word vectors, also called *word embeddings* (Jurafsky & Martin, 2018, p. 107), originally they were typically created with dimensions corresponding to the words in the vocabulary and mostly differed in what the values were based on (Jurafsky & Martin, 2018, p. 118). Often the values were based on frequency of targeted words co–occurring next to neighboring words. These types of word vector methods created sparse and long vectors, i.e. containing mostly zeros since most words do not occur next to others (Jurafsky & Martin, 2018, pp. 115–116).

These sparse and long vectors could achieve reasonable performance with long training times, but it turns out that dense and short vectors are better than the previously mentioned in almost all natural language processing task (Jurafsky & Martin, 2018, p. 118). There are three important reasons for this behaviour, firstly, sparse vectors contains mostly zeroes where as dense vectors contains significant information. This entails that a few dense vectors can produce a similar performance to that of very many sparse vectors, and complete the task in a fraction of the time. Secondly, the sparse and long vectors rely on using more parameters of

explicit counting which makes it more difficult to generalize and runs a risk of overfitting. Thirdly, dense vectors can better represent synonyms. The sparse and long vectors, relying on each word being a distinct dimension will not represent two synonyms words as similar since these are apart of the dimensions and not the values(Jurafsky & Martin, 2018, pp. 118–119).

Creating short and dense vectors can be done through the use of modern vector semantics, such as the *Word2vec* method. Word2vec was created by Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey A. Dean (US Patentnr US9037464B1, 2013), a team from Google. They developed two algorithms, *skip–gram with negative sampling* (SGNS) and *continous bag of words* (CBOW) (Witten, Frank, Hall, & Pal, 2017, p. 539). The general thought process behind Word2vec is that instead of asking how often word *W* occurs next to word *Q*, the binary question of whether word *W* is likely or not to appear next to word *Q* is asked. This binary task is given to a classifier, however, the predictions made are not important, it is the weights the learned classifier created that are used as the word embeddings (Jurafsky & Martin, 2018, p. 119). A major reason for the success in these types of tasks is that the binary question asked has its gold standard answer in the text document without any hand–labeled data. Simply look up neighboring words to the target word, and classify the question as yes or no if the word can be found in the nearby context.

The previously mentioned SGNS algorithm first labels the target word with another word occurring next to each other as a positive class. It then randomly selects a word from the vocabulary along with the targeted word and labels them as a negative class. This is repeated until the dataset is completed. Afterwards a classifiers is trained to differentiate between the two classes. Finally, the weights created by the classifier are used as the word embedding.

The software WEKA supports Word2vec through the classifier they name StringToWordVector and allows the modification of parameters. These are the *minWordFrequency*, *stopwordsHandler*, *words to keep*, *tokenizer*, *TF– and IDF–transformation* and *data normalization* (Trigg, Inglis, Paynter & Kibriya, n.d.). The minWordFrequency sets the minimum number of times a word has to occur in the dataset for it to be allowed into the binary dataset. The stopwordsHandler allows the selection of an method that removes stopwords; the user can select one or multiple files containing lists of words to be discarded in either clear text or *regular expressions*, or chose a predefined list of words (Trigg, Inglis, Paynter & Kibriya, n.d.). Stopwords are words that are typically deemed unnecessary for the natural language processing task, they are common and frequently occurring word such as ‘the’ and ‘a’ (Jurafsky & Martin, 2018, p. 68). The wordsToKeep parameter defines the maximum number of words allowed per class. It will select the top most frequent words per class with respect to the parameter’s value. If there are words with the same frequency as the last allowed word those additional words

are also included, that is, it is possible to exceed the wordsToKeep (Trigg, Inglis, Paynter & Kibriya, n.d.).

The tokenizer is the method that converts long strings of words into a separate string per word. WEKA uses the preexisting *Java* StringTokenizer class ("Java™ Platform, Standard Edition 8 API Specification", n.d.), but also allows the addition of new delimiters (characters by which the string is cut with).

TF– and IDF–transformation allows the Word2vec method to also use word frequencies, and to modify the raw frequencies into a logarithmic version (Trigg, Inglis, Paynter & Kibriya, n.d.). TF is an abbreviation of term frequency, and by enabling the transformation parameter the term frequency becomes $log_{10}(1 + f_{ij})$ where f_{ij} is the frequency of word i in instance j. A word that occurs 100 times does not mean it is 100 times more likely to be relevant for the instance than a word that only appears once, so by using the logarithm with base 10 this value is smoothed out proportionally to their size (Jurafsky & Martin, 2018, p. 114). IDF is an abbreviation of inverse document frequency, this transformation gives higher weight to words that only occur in a few instances (note that the word may be frequent in the instances they appear in but they only occur in a small number of instances). The equation used is $f_{ij} * log_{10}(No.of\ Instances\ /\ No.of\ Docs\ with\ word\ i)$ where f_{ij} is the frequency of word i in instance j. Combining Word2vec with TF– and IDF–transformation has shown promising results in previous studies (Lilleberg, Zhu and Zhang, 2015) and this type of measure is widely used in information processing (Witten, Frank, Hall, & Pal, 2017, p. 336).

The last parameter, normalization, allows three different settings: to *notNormalizeData*, to *normalizeTestDataOnly*, or to *normalizeAllData*. The setting processes and normalizes the frequencies of words to the average length of all instances. If normalizeTestDataOnly is chosen, the frequencies are changed according to the average length of test instances and not those belonging to the training set (Trigg, Inglis, Paynter & Kibriya, n.d.).

2.5 Stockholm – Umeå Corpus (SUC 2.0)

The Stockholm – Umeå Corpus (SUC) is a balanced corpus and designed after the Brown corpus, meaning it is a collection of texts that has been annotated with part–of–speech tags, morphological analysis and similar annotations (Källgren, 1999). The corpus consists of one million Swedish words in the form of 500 samples of different texts, with close to an average of approximately 2,000 words per text. SUC was revised and a new version, SUC 2.0 was created. It follows the same structure as the previous version but has additional annotations for the textdata (Gustafson–Capková, Hartmann, & Källgren, 2006), such as its text categories (see list on the next page for which categories are used).

- Reportage
- Editorial
- Reviews
- Skills, Trades and Hobbies
- Popular Lore
- Biographies
- Miscellaneous
- Imaginative prose

2.6 DigInclude corpus

The DigInclude corpus (Rennes & Jönsson, 2016) consists of web texts from Swedish public authorities and municipalities that has been split into sentence–pairs. Each sentence–pair feature a standard sentence, and a corresponding easy–to–read version. The sentences were gathered through a web crawler that scraped the websites of municipalities and public authorities that also featured an easy–to–read version of the website. The sentences are therefore manually created by hand with quality equal to what can be expected of Swedish authorities. The dataset also contains a value for each sentence–pair, the alignment measure which indicates how similar the two sentences are to each other (Rennes & Jönsson, 2016). This alignment measure is a scale between 0 to 1, a high score indicates that the sentence pair is similar to each other whereas a low score indicates that the sentence pair is dissimilar (Rennes & Jönsson, 2016).

2.7 Text complexity features

The SUC corpus has also been preprocessed and 117 features have been extracted through the use of SCREAM (Falkenjack, Rennes, & Jönssons, 2016). The features are based on Falkenjack, Mühlenbock, and Jönsson's (2013) previous work, and aims to describe text complexity. They are divided into four categories, shallow, lexical, morpho–syntactic and syntactic features. The shallow features are values that can easily be extracted since they are based on counting words and characters. Lexical features are based on categorical word frequencies, such as how frequently a word is used in the text. The morpho–syntactic features are extracted by analyzing the texts' morphological structure, these are probabilities such as the ratio between the part–of–speech tags or the ratio of content words to filler words. The last category of measures, syntactic, are extracted after parsing the text and consists of features such as the

average length of dependencies or average number of tokens per sentence clause (Falkenjack, Rennes, & Jönssons, 2016, pp. 1–3).

3 Implementation of Classification Tasks

The methodology followed in this thesis began by creating the datasets from the two corpora and turning them into *ARFF*-files which is the file format accepted by WEKA, the software used to perform the classification tasks. The SUC datasets were then tested on the differing classifiers and preprocessing methods, starting with using default settings and then attempts at optimizing the parameters. There were too many parameter setups to be completely exhaustive, but the parameters were systematically modified in an attempt to be as exhaustive as reasonable.

3.1 WEKA

In order to run the text classification tasks, the software WEKA was used. It includes a collection of machine learning algorithms and tools for data preparation, classification, evaluations, and visualizations. The software can be accessed through a graphical interface (GUI), a command line interface in Java, and an application programming interface (API) that can be called through different programming languages. In the case of this thesis the software was accessed through the GUI. The classifications and preprocessing filters relevant to the thesis can all be modified through the use of parameters, which was important for the performance and success rate in terms of correct classifications and weighted average f-measures.

3.2 SUC

The SUC corpus was sanitized and had most annotated data removed since they would not be needed for these tests, and then recreated in the ARFF-file format which is used in

the WEKA software. This process was done with a Python script written by the researcher. The text complexity features where gathered from previous studies (Falkenjack, Rennes, & Jönssons, 2016), which were used as features for the first dataset. Word vectors were created with StringToWordVector and used as features for the other dataset. SMO and MLP were used as classifiers, and PCA was tested with the text complexity measure to investigate the performance of the principal components for the measures.

3.2.1 Data sanitization and ARFF-file

Creating the dataset for use in WEKA was done through a Python-script. The words from the corpus was extracted and ordered by text, one long string containing all words and sentences per text. Each string was also given the associated text-id from the corpus, since the id also identifies which text category the text has been classified as.

Special characters, such as punctuation and numbers, were removed from the dataset. All uppercase letters where transformed to their lowercase counterparts. The text-ids, which up to this point had been unique, were transformed into their respective class. With the sanitization complete, the data portion of the ARFF-file was complete. Each row contained the text in the form of a single string, followed by its class.

The header section for the ARFF-file was added to the top of the textfile. Two features were defined in the header since the data section contained two types of data. The first feature, the text, was defined as a string. The classes of text categories was defined as a nominal feature and contained the nine different text categories.

3.2.2 SMO

The ARFF-file for SUC with Johan Falkenjack's text metrics as features was imported, and no raw text data was used. The SMO/SVM classifier was chosen as the targeted classifier to use. WEKA supports the modification of parameters for SMO, namely, the complexity constant and the type of kernel to use (see chapter 2.1).

For the first classification task the default values was used, the complexity constant was set to 1 and the PolyKernel was chosen. The model was 10-fold cross validated and saved. Once the default task was completed optimization was started. Based on the papers from Chang, Hsieh, Chang, Ringgard, and Lin (2010, p. 1472) and Goldberg, and Elhadad (2008, p. 240) it was decided that PolyKernel would not be changed, since it tends to achieve the best performance of the available kernels for natural language processing tasks. The remaining parameter, the complexity constant, was incremented by 0.5 up to 10 per new model. All created models were 10-fold cross validated, and saved to the computer running the classification tasks for storage and future reference.

3.2.3 SMO + PCA

The SUC dataset with text metric features was imported to the WEKA software. A method called *FilteredClassifier* was selected in the WEKA software. It allowed the data to be run through an arbitrary filter before being passed to the classifier. SMO was selected as the classifier, and PCA was selected as the filter. This allowed the SMO to be exposed to the features created by the PCA. In WEKA there is support for three PCA parameters: variance covered, maximumAttributes, and maximumAttributeNames (see chapter 2.3).

The first task was run using default values for all parameters, variance covered was set to 0.95, and the maximumAttributes was set to –1, unlimited. The model was 10-fold cross validated and saved. The second task used default parameters for the PCA, but replicated the parameters from the previously optimized PCA task, the complexity constant was set to 2.5. It was 10-fold cross validated and saved. Next the PCA was optimized. The maximumAttributeNames was ignored since it does not impact performance of the methods. The maximumAttributes was set to –1 (unlimited) to ensure that all the relevant principal components was used in the training of the SMO classifier. The variance covered was set to a range between 0.5 and 1, and was incremented by 0.05 per model created. The SMO's parameters was replicated from previous the optimization of SMO. All models were 10-fold cross validated and saved.

3.2.4 SMO + StringToWordVector

The SUC dataset containing the documents text as strings and text categories as classes was imported to the WEKA software. Once again the FilteredClassifier was selected. SMO was chosen as the classifier and StringToWordVector as the filter. WEKA supports minWordFrequency, stopwordHandler, words to keep, TF– and IDF–transformation, document normalization, and tokenzier as the parameters for the StringToWordVector (see chapter 2.4).

For the first task default settings were used. MinWordFrequency was set to 1, words to keep to 1000, stopwords were not removed, data was not normalized, TF– and IDF–transformation was activated, and the standard Java tokenzier was used without additional delimiters.

Optimization of the StringToWordVector's parameters was performed. Based on the paper written by Lilleberg, Zhu, and Zhang (2015) and the book by Witten, Frank, Hall, and Pal (2017, p. 336) it was determined that TF– and IDF–transformation would remain activated through all models. Since the dataset had already been preprocessed using a Python-script and all tokens could be delimited by blankspace the tokenizer also remained as is and was not further modified.

The Words to keep was set to a range from 500 to 2000, and was incremented by 500. The minWordFrequency was set to a range between 1 and 25, and was incremented by 5. Removing or allowing the stopwords to remain were both tested. Likewise, normalizing all data or normalizing nothing were both tested. The above parameter ranges were all checked against each other's possible permutation, creating 96 models. These where 10-fold cross validated and saved.

3.2.5 MLP

The SUC dataset with text complexity measures as features was imported to the WEKA software. The MultilayerPerceptron was selected as the classifier. The parameters that WEKA supports for MLPs are learning rate, momentum, batch size, training time, validation set size, validation threshold and the setup of hidden layers (see chapter 2.2).

The first classification task used default settings, learning rate was set to 0.3, the momentum was set to 0.2, and the hidden layer setup up consisted of one layer with the number of nodes being equal to the sum of features and classes divided by 2. The batch size was set to 100, the training time to 500, the validation set size to 0, and the threshold to 20. The model was 10-fold cross validated and saved.

The optimization of the MLP was begun. The training time was set to 5000 so that the MLP would be given a long time before being forced to end. Instead of relying on the training time to stop the classifier, a validation set size and validation threshold was defined. The validation set size was defined to 20, meaning that 20% of the dataset was used for the MLP validation and the remaining 80% for learning and 10-fold cross validation. The validation threshold was set to the recommended 20 (Ware, n.d.). The batch size was set to the default 100. Based on the recommendation of Witten et al. (2017, p. 285) only one hidden layer was used, instead the number of nodes in said layer was modified.

The remaining parameters, learning rate, momentum and number of hidden nodes, was modified with the use of an automatic algorithm. By defining the ranges and increment size of each parameter in the automatic algorithm the software could start making the next model as soon as the previous was completed. This was algorithm was employed since MLPs requires significantly more time to train and evaluate than the previous methods. This meant that the software could be left running and would never stop creating models until the predefined parameters were exhausted. Both the learning rate and momentum was set to a range between 0 and 1, the lowest and highest allowed value for the parameters, with an increment size of 0.1. The number of nodes was set to a range between 1 and 100, with a increment size of 10. The predefined wildcards that WEKA allows were also included (see table 2.1). The models were 10-fold cross validated and saved.

3.3 DigInclude

The DigInclude corpus was sanitized and recreated in the ARFF-file format. With the use of the graphical interface through WEKA the dataset was ran through the SMO classifier using StringToWordVector as the features, and then Multilayer perceptron as another classifier. Attempts at creating the text complexity measures through the SCREAM software were made but failed due to incompatibility.

3.3.1 Data sanitization and ARFF-file

The DigInclude corpus contains approximately 118,000 sentences, each mapped to another corresponding sentence creating the 59,000 sentence pairs along with an alignment score between the sentences in each pair. Creating the dataset used for classification was done using a Python-script. Sentences with a lower alignment score than 0.7 were removed, duplicates were removed, uppercase letters where transformed to their lowercase counterpart and lastly, dates and numbers were replaced with the string 'NBR'. Removing duplicates had three steps, firstly removing sentence pairs that were duplicates of other pairs. Secondly, pairs in which both sentences were the same. Thirdly, sentences that were duplicates of sentences from other sentence pairs. This process resulted in 6,461 unique sentences.

Attempts to create the text complexity measures from SCREAM in order to create a similar dataset such as was used for SUC were performed. However, the the service which had processed SUC could not be used on the DigInclude data. SUC has few but long texts whereas DigInclude has many but short sentences and as such was incompatible with SCREAM and made the service crash.

The sentences were then structured according to the ARFF-file format. Where previously each sentence pair was structured with both sentences on one row, they were instead separated to one sentence per row. Following the sentence, delimited by a comma, the sentence's class was added.

With the data section of the ARFF-format complete, the header section was added. The data contained two features, and as such both are defined in the header. First a feature for the string, the sentence, followed by the class, standard or simplified.

3.3.2 SMO + StringToWordVector

The DigInclude dataset containing the sentences as strings and their respective binary class was imported to the WEKA software. The FilteredClassifier was selected in the WEKA graphical interface, with SMO as the chosen classifier and StringToWordVector as the preprocessing filter. The same parameters that was available when using SMO and StringToWordVector for

the SUC dataset was available, and the general strategy was replicated. The first classification task used default settings, minWordFrequency was set to 1, the words to keep to 1000, the data was not normalized, stopwords were not removed, TF– and IDF–Transformation was activated, and the built–in Java tokenizer was used without additional delimiters. The created model was 10-fold cross validated and saved.

As was done when training on the SUC dataset, the TF– IDF–transformation was activated and no models were created with the parameter deactivated. The tokenizer was also not modified, since the dataset had been preprocesed by a Python-script and could be delimited by blankspace. The words to keep was set to a range from 500 to 2000, with an increment size of 500. The minWordFrequency was set to a range between 1 and 25, and increment size of 5. Whether to normalize all or no data was tested, likewise with whether to remove or let stopwords remain. As before this created 96 models, they were 10-fold cross validated and saved.

3.3.3 MLP

The DigInclude dataset was imported to the WEKA software, and the FilteredClassifier was selected. MultilayerPerceptron was selected as the classifier, and StringToWordVector was selected as the filter. The same procedure as was performed for the SUC dataset when using MLP was replicated for the DigInclude dataset. The first task used default parameters values. The learning rate was set to 0.3, and the momentum to 0.2. The hidden layer used one layer with a number of nodes equal to the sum of the number of features and classes divided by 2. The training time was set to 500, the validation set size to 0, and the validation threshold to 20. The model was 10-fold cross validated and saved.

The optmization of the MLP was started. Instead of relying on the training time to stop the MLP, validation set size and threshold was used. The training time was set to 5000, while the validation set size was defined as 20% and the threshold as 20. The same automatic algorithm that was used for SUC tasks to continuously create models automatically was employed for the DigInclude tasks aswell. The learning rate and momentum was defined as a range between 0 and 1, with an increment size of 0.1. The hidden layer contained one layer and the number of nodes was set to a range between 1 and 100, incriminating by 10. The wildcards were also included. The models were 10-fold cross validated and saved.

4 Results

This section shows the result from all the different classification methods and features using default parameters settings along with their optimized counterparts split by the two datasets. Each table shows the classifier and its weighted average f-measure from the 10-fold cross validation, followed by the parameters modified to achieve said performance.

4.1 SUC Classification Tasks

The results presented below are from the classification tasks performed on the different SUC datasets and features. The classifiers which are presented are SMO and MLP and the filters are PCA and StringToWordVectors, with their respective results and optimization.

4.1.1 SMO

The first classification task on the SUC dataset using Falkenjack's features used a SVM coupled with SMO and used default parameters values; complexity constant was set to 1 and PolyKernel as chosen kernel (*weighted average f-measure* = .596).

The second task used optimized parameters, the complexity measure was set to 2.5 and the kernel to PolyKernel. This setup of parameters increased the performance when compared to the previous task (*weighted average f-measure* = .602), see table 4.1.

Table 4.1: SMO Classification Task

Parameter values	F-Measure	Kernel	Complexity constant
Default	0.596	PolyKernel	1
Optimized	0.602	PolyKernel	2.5

4.1.2 SMO + PCA

SMO was selected as the classifier and PCA as the filter. The first test used default parameters on both methods, the variance covered was set to 0.95 and the maximumAttributes was set to –1, unlimited. The kernel chosen was PolyKernel and the complexity constant was 1. The model was evaluated to a decrease in performance when compared to all previous test (*weighted average f-measure* = .562).

The second task used default parameters values for PCA but the SMO parameters was replicated from the most successful previous SMO performance. The variance covered was 0.95, the maximumAttributes set to unlimited, the kernel chosen was PolyKernel and the complexity constant set to 2.5. The model had a increase in performance in comparison to the previous task using SMO and PCA, but a decrease in performance when compared to only using SMO with optimized parameters (*weighted average f-measure* = .582).

The third task used optimized parameters for both PCA and SMO. The variance covered was set to 0.8, the maximumAttributes to unlimited, the kernel to PolyKernel and the complexity constant to 2.5. The created model had an increased performance in comparison to both previous PCA and SMO tasks, but a decrease when compared to the only using SMO with optimized parameters (*weighted average f-measure* = .590), see table 4.2.

Table 4.2: SMO+PCA Classification Task

Parameter values	F-Measure	Variance covered	maximum–Attributes	Kernel	Complexity constant
Default	0.562	0.95	Unlimited	PolyKernel	1
Default PCA & Optimized SMO	0.582	0.95	Unlimited	PolyKernel	2.5
Optimized	0.590	0.8	Unlimited	PolyKernel	2.5

4.1.3 SMO + StringToWordVector

The first task using SMO in combination with StringToWordVector used default values for both methods. Words to keep was set to 1000, the minWordFrequency to 1, no normalization was used, and no stopwords were removed. The model was evaluated and showed an increase in performance when compared to all previous tasks (*weighted average f-measure* = .732).

The second classification task used optimized parameters for both methods. The words to keep was set to 1500, the minWordFrequency to 10, all data was normalized, and no stopwords were removed. Using these parameters settings the model was created and evaluated, showing an increased performance when compared to all previous tasks (*weighted average f-measure* = .752), see table 4.3.

Table 4.3: SMO+StringToWordVector Classification Task

Parameter values	F-Measure	Words to keep	minWord-Frequency	Normalizing data	Stopwords included or not
Default	0.732	1000	1	No normalization	Stopwords included
Optimized	0.753	1500	10	Normalization	Stopwords included

4.1.4 MLP

The final set of tasks used on the SUC datasets used MLP as its classifier. First, default values was selected for the parameters. Learning rate was set to 0.3, the momentum to 0.2, and the hidden layer setup was set to one layer with the same number of nodes as the sum of the number of features and number of classes divided by 2. The model was evaluated and showed a decrease in performance in comparison to the best previous task, which used SMO and StringToWordVector *weighted average f-measure* = .609).

The second task used optimized parameters for the MLP classifiers. The learning rate was set to 0.3, the momentum to 0.2 and used one hidden layer that had the same number

of nodes as the number of features. The created model showed an increase in performance when compared to the previous MLP task, but a decrease to the overall best previous task (*weighted average f-measure* = .624), see table 4.4.

Table 4.4: MLP Classification Task

Parameter values	F-Measure	Learning rate	Momentum	Number of hidden layers
Default	0.609	0.3	0.2	(number of features + number of classes) / 2
Optimized	0.624	0.3	0.2	Number of features

4.2 DigInclude Classification Tasks

The following section covers the result from the classification tasks performed on the DigInclude dataset. The classifiers used are SMO and MLP and the filter used are StringToWordVector.

4.2.1 SMO + StringToWordVector

The first classification task performed on the DigInclude dataset used SMO as the classifier and StringToWordVector as the filter. The first model used default parameters, with words to keep set to 1000, the minWordFrequency to 1, no data was normalized and stopwords were not removed. The model was evaluated (*weighted average f-measure* = .537).

The second task optimized the classifier and filter. Words to keep was set to 500, the minWordFrequency to 20, no data was normalized and stopwords were not removed. The model was evaluated and when compared to the previous test the new performance showed an increase (*weighted average f-measure* = .565), see table 4.5.

Table 4.5: SMO+StringToWordVector Classification Task

Parameter values	F-Measure	Words to keep	minWord–Frequency	Normalizing data	Stopwords included or not
Default	0.537	1000	1	No normalization	Stopwords included
Optimized	0.565	500	20	No normalization	Stopwords included

4.2.2 MLP

The final tasks of the DigInclude dataset used MLP as a classifier. The first classification task used default settings for the parameters. The learning rate was set to 0.3, the momentum to 02, and the hidden layer setup used one layer and the same number of nodes as the sum of the number of features and number of classes divided by 2. The model created with these settings was evaluated and showed a decrease when compared to the previous SMO and StringToWordVector classifier (*weighted average f-measure* = .477)

The second task used optimized parameters for the MLP classifier. The learning rate was set to 0.3, the momentum to 02, and the hidden layer setup used one layer and the number of nodes was equal to the number of features. The model was evaluated, and showed an increase in performance when compared to the previous MLP task, but a decrease in comparison to the SMO classifier and StringToWordVector filter (*weighted average f-measure* = .515), see table 4.6.

Table 4.6: MLP Classification Task

Parameter values	F-Measure	Learning rate	Momentum	Number of hidden layers
Default	0.477	0.3	0.2	(number of features + number of classes) / 2
Optimized	0.515	0.3	0.2	Number of features

5 Discussion

For the classifications performed on the SUC dataset using the text complexity measures from SCREAM, the best performing model used SMO with optimized parameters. Adding PCA to the tests decreased the performance. Classification done using MLP was evaluated to a similar performance to that of only using SMO. Optimization of MLP increased performance, a slight increase in comparison of the SMO classifier. Best performance for the SUC dataset used word vectors, and with optimization showed an increase in performance in comparison to SMO and MLP.

The classifications performed on the DigInclude dataset was evaluated to poor performance, despite classifying between two classes all results had an average weighted f-measure between 0.4 and 0.6. The best performing classifier was SMO in combination with word vectors features.

5.1 DigInclude vs SUC

Comparing the overall performance of the models between the two datasets, it is clear that the models created on the SUC dataset have significantly better performance. Even though the DigInclude models aims to classify between two classes, the average weighted f-measure barely reaches above 0.5. A simple model choosing the most frequent classes would perform better. On the other hand, the models created for the SUC dataset tries to classify nine text categories but typically have a f-measure of 0.6 to 0.7. Despite choosing between more classes these models perform better.

There could be multiple factors behind this discrepancy. There are many differences between the datasets, since fewer classifications methods were tested on the DigInclude datasets the answer could be that the chosen methods perform poorly and that an untested method has greater performance as was possible for the SUC dataset. Another potential issue is that the two datasets' classes are different. The SUC dataset identifies nine different text categories, while the DigInclude dataset identifies simple or standard Swedish language. The complexity of text could potentially be better described along an axis rather than two binary classes, whereas text categories typically are described as fixed classes.

That being said, I suspect it would be difficult to achieve good performance regardless of chosen method. The reason is that the DigInclude has less datapoints than what the SUC dataset offers. The DigInclude corpus contains approximately 118,000 sentence pairs, but once the duplicates were removed approximately only 6,500 sentences remain. This issue is exasperated when using preprocessing methods, such as StringToWordVector, since even more information is removed due to parameters like minWordFrequency. The aforementioned method tends to improve when removing words that does not met a frequency criterion, and when removing stopwords. In the case of DigInclude such processing did slightly increase performance but leaves little data for successful classification. Finally, part of the dataset is used for evaluation and as such was not included when training the models, leaving even less data to be used for the classification training.

Further research on the dataset could more precisely narrow down the issues and what could be improved but first and foremost I suggest attempts at increasing the number of viable sentence pairs and expanding the corpus.

5.2 On the Topic of Features

The question of how text complexity measures relates to the classification of text categories could be raised. The measures, such as word frequency or ratio of part–of–speech tag, could be thought to be poor features to use for the classification of text categories since it is typically said that the context better describe such categories. However, as shown by this thesis' result the use of said features along with an SMO classifier the performance of the classification task showed good performance in terms of weighted average f-measure. This further illustrates the importance of choosing how the dataset one wish to use is processed. There is value behind investigating how to present the data to the classifier, and in my opinion it is worth to explore what features are used and to not necessarily chose what first comes to mind.

5.3 Optimization

Optimizing the classifiers and the filters can be an arduous task since there are many parameters to modify, and most of said parameters allows for a large range of input. In fact, since the parameters allows for decimal numerals you could argue there are an infinite number of variations. The strategy deployed in the thesis aims at trying a large variety of parameters by setting the lower floor to the lowest allowed numeral, typically 1, and setting the upper roof to a numeral where the performance starts to decrease. This opens up an issue where it is possible a local maximum is found but if the upper bound was increased the global maximum might instead be found.

Since changing one parameter can have impact on other parameters it is not a good idea to optimize one parameter at a time, instead one must try every combination of parameter setups in order to ensure that all parameters performs optimally. As an example, the optimization of the StringToWordVector created 96 models since the first parameter had 6 different allowed inputs, the second had 4, and the third and fourth both had 2. Smart optimization algorithms have been created and can speed up the process of finding the optimum parameters setup, however as a result can end up being even less exhaustive in its trade–off for speed. In an effort to minimize sources of error no smart algorithm was deployed. Most models were manually created, save for MLP optimization for which an algorithm created the models on its own with the input defined by the user. It does not use an intelligent system but is exhaustive based on what the user allows. In other words, it created the exact same models as would have been created manually, introducing no additional factors or sources of errors.

5.4 StringToWordVector

Using word vectors increased the performance of the SMO algorithm, even when using the default settings. The method has long been favored by text analysts and the recommendations to use word vectors seems to not have been unfounded. However, there are some hurdles that should be kept in mind when using. Many of the parameters that can be modified for the method in WEKA can have a significant impact of the number of words used to create the vector space. The parameter called MinWordFreq sets a minimum number of times a word must be seen in the dataset to be allowed to use, setting the number to 1 allows all words to be used, and setting it higher removes words that are seldom used. The data used for the classifier becomes denser in terms of number of important words versus the total number of words. In the models created for the thesis this parameter tended to increase performance, but increasing it too much eventually led to decrease in performance due to the severe reduction in data it could create.

5.5 Text Complexity Features and DigInclude

There were attempts made to create the same text complexity measures for the DigInclude corpus that had been created for the SUC corpus. The service that created the features for SUC, SCREAM, was however not compatible with the DigInclude dataset. The SUC dataset contains longer but fewer texts than the DigInclude dataset which contains short but many sentences. The service was not created to handle small but large amounts of data and would throw errors when attempting to use the software with the DigInclude dataset. The team behind the SCREAM service were in the process of creating a new version to also handle this type of data. This newer version of the service was a work in progress when I was allowed entry for its use. Cross checks between the older version and the newer version was done with the use of the SUC dataset and showed a difference in the output of features. It was concluded that the new version was not correctly working and the creation of the features for the DigInclude dataset was deemed a failure.

I believe future research into this area still has academic value, how well text complexity measures behave for the classification of simple or complex language is an interesting research question. Colloquially complex language use is often referred to the text's length of sentences or the use of long words in substitution for common shorter words. If that is the case text complexity features could show good performance.

6 Conclusion

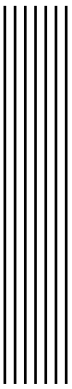
The models created on the SUC dataset attempted to classify text to one of nine categories and showed good performance overall, but optimization could improve the models even more. Using the vector space created by the StringToWordVector with the SMO classifier showed the best performance and with optimization created the best model with an average weighted f-measure of 0.753.

The text complexity features in combination with an optimized SMO classifier was evaluated to a performance of an average weighted f-measure of 0.602. Processing these features with the use of PCA decreased this performance, attempts at optimizing PCA were not able to increase the performance above using the original text complexity features.

This would indicate that for classifying text to their correct text category as defined by SUC it is the context of how words are used that perform better than the use of lexical features such as the text complexity measures from SCREAM.

All models created based on the DigInclude dataset attempted to classify the sentences to one of two classes, no model achieved good performance. More than likely this indicates that the dataset could have issues, given the small size of data that was deemed acceptable for use in training and testing the issue likely lies there.

As pointed out in Scope and Limitations, this thesis has only performed classification tasks on dataset in Swedish. However, this research question is just as relevant for other languages and combining results from multiple languages could show an indication of how all languages tend to perform and ways to treat data and their text features to maximize performance of classification tasks. Such data could have implication for future research and development of tools that span more languages than one.



Bibliography

Chang, Y.-W., Hsieh, C.-J., Chang, K.-W., Ringgaard, M., & Lin, C.-J. (2010). Training and testing low-degree polynomial data mappings via linear svm. *Journal of Machine Learning Research*, *11*, 1471–1490. United States.

Falkenjack, J. (2018). Towards a model of general text complexity for swedish. Linköping.

Falkenjack, J., Mühlenbock, K. H., & Jönsson, A. (2013). Features indicating readability in swedish text. In *9th nordic conference of computational linguistics (nodalida 2013)* (pp. 27–40). Oslo.

Falkenjack, J., Rennes, E., & Jönssons, A. (2016). Services for text simplification and analysis. Linköping.

Frank, E., Legg, S., & Inglis, S. (n.d.). Class smo. Retrieved from http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/SMO.html

Goldberg, Y. & Elhadad, M. (2008). Splitsvm: Fast, space-efficient, non-heuristic, polynomial kernel computation for nlp applications. (pp. 237–240). n.p.

Gustafson-Capková, S. & Hartmann, B. (2006). *Manual of the stockholm – umeå corpus version 2.0*. Stockholm.

Hall, M. & Schmidberger, G. (n.d.). Class principalcomponents. Retrieved from http://weka.sourceforge.net/doc.dev/weka/attributeSelection/PrincipalComponents.html

Java™ Platform, S. E. 8. A. S. (n.d.). Class stringtokenizer. Retrieved from https://docs.oracle.com/javase/8/docs/api/index.html?java/util/StringTokenizer.html

Jurafsky, D. & Martin, J. H. (2018). Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition. n.p.

Källgren, G. (1999). *Documentation of the stockholm – umeå corpus*. Stockholm.

Lilleberg, J., Zhu, Y., & Zhang, Y. (2015). Support vector machines and word2vec for text classification with semantic features. (pp. 136–140). Beijing, China. doi:10.1109/ICCI-CC.2015.7259377

Mikolov, T., Chen, K., Corrade, G. S., & Dean, J. A. (2013). *Computing numeric representations of words in a high-dimensional space*. US9037464B1. U.S patent.

Mühlenbock, K. H. (2013). *I see what you mean. assessing readability for specific target groups*. Göteborg: Göreborgs Universitetet.

Platt, J. (1999). *Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods*. Redmond: Microsoft Research.

Rennes, E. & Jönsson, A. (2016). Towards a corpus of easy to read authority web texts. Linköping.

Rifkin, R. M. (2002). *Everything old is new again: A fresh look at historical approaches in machine learning*. Cambridge: Massachuisetts Institute of Technology.

Tibell, R. (2015). Training a multilayer perceptron to predict the final selling price of an apartment in co-operative housing society sold in stockholm city with features stemming from open data. Diploma thesis, KTH Royal Institue of Technology.

Trigg, L., Inglis, S., Paynter, G., & Kibriya, H. M. (n.d.). Class stringtowordvector. Retrieved from http://weka.sourceforge.net/doc.dev/weka/filters/unsupervised/attribute/StringToWordVector.html

Ware, M. (n.d.). Class mlp. Retrieved from http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/MultilayerPerceptron.html

Witten, I. H., Frank, E., Hall, M. A., & Pal, C. J. (2017). *Data mining: Practical machine learning tools and techniques*. Cambridge: Morgan Kaufmann.