Sentiment Analysis Multitool, SAM

Bachelor Project
Course Code: BIBAPRO1PE

Kongsbak, Mads Guldborg Kjeldgaard (mkon@itu.dk)
Christensen, Steffan Eybye (sech@itu.dk)
Puvis de Chavannes, Lucas Høyberg (lupu@itu.dk)
Jensen, Peter Due (pedj@itu.dk)

Supervised by
Strømberg-Derczynski, Leon (leod@itu.dk)

May 15, 2019

Abstract

The authors of this thesis seek to implement a general purpose program that uses different approaches to programmatically predict the sentiment of sentences in comments on Danish social media - more specifically political comment sections - in what is called Natural Language Processing. Furthermore, to support this approach a dataset consisting of a large quantity of comments needed to be collected as well as a comprehensive amount of annotating needed to be done, such that it would be possible to evaluate the accuracy of the implementations. During the process the authors found that the dataset was weighted towards containing more negative sentiment rather than positive, and that this trend especially was prominent on Facebook. The authors believe that there may have been a flaw in the methodology that could have negatively affected the results. The thesis concludes that with significantly more data, the accuracy should increase consequently also diversifying the scope of the dataset such that it is no longer too specific.

Contents

1 Introduction **1**

2 Methodology **2**
2.1 Social Media . . . 2
2.2 News Media . . . 2
2.3 Articles . . . 3
2.4 Labelling our data . . . 3
2.5 Krippendorff's alpha . . . 4
2.6 Streamlining our annotations . . . 4
2.7 Cross validating our dataset . . . 5

3 The Data **6**

4 The Architecture **9**
4.1 SOLID-principles . . . 9
4.2 Single Responsibility Principle . . . 9
4.3 Open/Closed Principle . . . 9
4.4 Liskov Substitution Principle . . . 10
4.5 Interface Segregation Principle . . . 10
4.6 Dependency Inversion Principle . . . 10
4.7 The Pipe . . . 10
4.8 Settings . . . 11
4.9 Python . . . 12
4.10 Parallelism . . . 13
4.11 Refresher . . . 14

5 Classifiers **15**
5.1 Primitive Classifiers . . . 15
5.2 Lexical Analysis . . . 15
5.3 MML . . . 21
5.4 The SVM . . . 21
5.5 Classification using a Neural Network . . . 29

6 Analysis **31**

7 Discussion **33**
7.1 Time constraints . . . 33
7.2 Negatives and Neutrals . . . 33
7.3 Accuracy and Diversity . . . 34

8 Conclusion **37**

References **38**

A Appendixes **42**
A.1 Link to repositories . . . 42
A.2 MML 100 runs . . . 42
A.3 SVM 100 runs . . . 44
A.4 Multi-SVM 100 runs . . . 46

A.5 Neural Network 100 run statistics 48
A.6 SVM 100 sentences test 100 runs 48
A.7 Program to test AFINN 50
A.8 Baseline Classifier test results 52
A.9 Random Classifier test results 52
A.10 AFINN test results 53
A.11 Lexical Analysis Results 54
A.12 Baseline Classifier F1 Score Calculation 55
A.13 Random Classifier F1 Score Calculation 56
A.14 AFINN Classifier F1 Score Calculation 57
A.15 Lexical Analysis F1 Score Calculation 58

1 Introduction

In this project we aim to create a general purpose tool for sentiment analysis in the software genre called Natural Language Processing. More specifically, this tool should be specialized in predicting sentiment in sentences in comments on Danish political articles on social media - which is called Sentiment Analysis[1].

To constitute a general purpose program there are several functionalities that we find important to include. First, the user should have several customizable options for running the program. This could include having direct impact on how the program behaves in specific circumstances or on a more abstract level - change what algorithm is being used. Second, the program should - together with a set of easy to understand instructions - be usable for a non-technical individual. Last, the program should be able to reliably predict sentiment in sentences in comments on all sorts of political articles.

In the project we were to implement two or more different types of algorithms that are able to predict sentiment in sentences. An algorithm that is capable of taking input data and map it into a category is called a classifier[2]. There are three types of classifiers relevant to this paper: The rule based/lexical classifier, the support vector machine (hereafter SVM) and neural network (hereafter NN). The three different types of classification will be thoroughly described later. The important note here is that two out of the three approaches are using supervised machine learning techniques[3]. Explained simply, supervised machine learning uses an arbitrary machine learning algorithm to fit a function to a dataset so the algorithm ”learns” the characteristics of the data, and afterwards will be able to predict which class an arbitrary input belongs to. The ”supervised” part means that each data entry we feed the algorithm is accompanied by a ”label” of what class the entry belongs to. In our case we want the algorithms to be able to tell whether a sentence is positive, neutral or negative. Therefore we need to collect a vast amount of data, so that we can enhance the ability of the algorithm to do qualified guesses. The more and the better data we have, the better results we should get. This also leads to the question: how do we determine if results are good or bad? To do this sort of distinction it is important to collect a large amount of data and train the algorithm to what we believe is the right answer. When the algorithm then makes a qualified guess, we can determine whether that guess was correct according to our labelling.

How the program is constructed is also an important factor. This is not directly related to the functionality, but it is however very important in relation to how scalable the program is with an increasingly larger dataset. Therefore we will also go into detail with how the program was constructed, what decisions were made to support the usability of the program and what the result of these changes were.

[1]https://monkeylearn.com/sentiment-analysis/
[2]https://monkeylearn.com/sentiment-analysis/
[3]https://www.ibm.com/downloads/cas/GB8ZMQZ3 - page 15

2 Methodology

In this section we will explain how we approached gathering and annotating our data.

To be able to do sentiment analysis in sentences in comments on political articles we had to select which social media, which news media and which articles that we were to collect from. One of the things we kept in mind when selecting the previous mentioned data sources was that we wanted to create a diverse dataset. Therefore, exposure/reach was a very important value to consider. We also wanted to avoid echo-chambers [1], because they would skew the dataset towards being less representative for general purpose use. We only collected the root comments of a post, which is defined by a comment that are directly replying to the post itself and not replies to such a reply. This was done in order to ensure the comment were related to the article, rather than previous comments. We also created 'tags' for each of the articles, which basically consists of the main topics that the articles are relevant to. This was done because *if* we wanted to compare comments between different topics, this was an easy way to sort the dataset. It would also enable us to keep track of how diverse our dataset was.

2.1 Social Media

To get the most diverse dataset it was clear that we had to utilize the most used social media in Denmark. It therefore was quite obvious that we were to collect data from Facebook[4] because Denmark has one of the highest Facebook users per capita. Facebook also has a rich environment of news media to collect from, which again gives the ability to create an even more diverse dataset.

The next social media that we chose to use was Reddit[5]. Reddit is a forum consisting of subforums in which Denmark has an unofficial national Reddit — which is called a subreddit. The Danish subreddit has 114 349 subscribers[2] and is known to provide more diverse discussions on posts than Facebook, since it was built for that purpose[3].

The last social media that we chose to use was Twitter[6]. Twitter is a social media where there is a fairly short character limit on the tweets(posts), but many political actors are active on twitter, and it was therefore also deemed to be a valuable source of data.

2.2 News Media

The first major news media data source was TV2's Facebook-page. Said media has the highest number of likes, sitting at a total of 594 062 likes[4], amongst all the Danish news Facebook pages. This means that essentially 10% of the Danish population are following the media, which through likes, shares, reactions and comments will yield a much larger reach. Below is a list of other news media pages we gathered data from, and the number of likes each page had.

- B.T.: 398 238 [5]

[4]https://www.facebook.com/
[5]https://www.reddit.com/
[6]https://twitter.com/

- DR Nyheder: 298 238 [6]
- Politiken: 275 050 [7]
- Dagbladet Information: 151 117 [8]

Late in the process our dataset lacked positive sentiment, therefore we included comments from Fucking Flink's Facebook-page. The name translates to "fucking nice" and is essentially a positive-minded echo-chamber with 174190 likes[9]. Even though Fucking Flink is to be considered an echo-chamber, it was a very efficient way to collect many positive comments, which was exactly what we needed to ensure a diverse dataset.

2.3 Articles

When we chose our political articles we implemented two rules:

1. It had to be directly politically relevant
2. It had to be at least 48 hours old

We kept from implementing more rules as we were afraid of creating a biased dataset. How one defined directly political was also left out to be a subjective judgement, because what someone would find political relevant — others might not. It being politically relevant is essential for this project, since we want our machine learning algorithms to learn the language used in that context. The reason behind that a post had to be 48 hours old is we deemed it necessary to ensure that we had a broad range of people who have commented on them.

2.4 Labelling our data

The process of labelling the data is critical when creating a dataset for supervised machine learning. Labelling is the process of analysing the data and giving it a label, so that behind-the-scenes the algorithms can try to distinguish them. How we chose to label the data will therefore directly affect the results we get. In our case we did fine-grained polarity, which meant instead of just doing 3 types of classification:

- 1 — Positive
- 0 — Neutral
- -1 — Negative

We instead went with five types of classification.

- 2 — Very Positive
- 1 — Positive
- 0 — Neutral
- -1 — Negative
- -2 — Very Negative

We believed this would give us a more versatile model, and that it would make it easier for the machine learning algorithms to detect positive/negative sentiment.

2.5 Krippendorff's alpha

Krippendorff's alpha is a statistical method for calculating the overall agreement between a given number of annotators, independently annotating a dataset [10]. We used an online tool named "ReCal" [11] to calculate our alpha values. Since every member of the group independently annotated the words in the lexicon used for the rule-based classifier, the alpha value can be used to indicate the reliability of the lexicon. We reached an alpha value of 0.922 on our annotation of our lexicon. This indicates that we had a high level of agreement and that the ratings are correct. In the beginning of the project, we had the assumption that we would have a high level of agreement. As advised by our supervisor we decided to annotate the first 100 sentences and compare our annotations to ensure that the assumption was correct. Since we decided only to use the general polarity of the sentences, we normalized negative ratings to -1 and the positive ratings to 1. When we then again calculated the alpha value on the initial annotations, we reached an alpha of 0.547. This means we were not much in agreement.

2.6 Streamlining our annotations

To streamline our labelling of the sentences we followed an idea by our supervisor to make some guidelines for how we would annotate our data. We found support to this idea by Saif M. Mohammed who says: *"Clear and simple instructions are crucial for obtaining high-quality annotations. This is true even for seemingly simple annotation tasks, such as sentiment annotation, where one is to label instances as positive, negative, or neutral"* [12]. These guidelines were created by selecting a subsection of our collected data, which in total was 100 sentences, which we then evaluated collectively. Once we all agreed upon a rating for a given sentence, we created a specific guideline to support this rating. Specifically we ended up looking for tone, words, agreement, emojis and other special characters such as question marks, exclamation marks etc. Below are the guidelines we created. Note that they might overlap and it was left up to the annotator to determine what rules applied:
2:

- Positive action with positive intensifier.
- Direct positive sentiment towards topics, people or objects.
- Many tokens that indicate positivity e.g <3 <3 <3 <3 <3.

1:

- Turns something negative into something positive.
- Something positive is being referred to.
- Positive tone/message.
- Wishes something positive without being eagerly formulated.
- Subtle expression of positive attitude towards individuals/person/objects.
- Few tokens that indicate positivity.
- A sentence that is near impossible to use negatively (yet often positively).
- They express agreement.

0:

- A sentence that can only be understood in context is inconclusive.
- That which is not immediate positive nor negative.
- Sentences that are not inherently positive or negative.
- Impartial expressions.
- Objective descriptions.

-1:

- They express disagreement.
- Tokens that indicate negativity e.g :(.
- Turns something positive into something negative.
- Something negative is being referred to.
- Negative tone/message.
- Wishes something negative without said being eagerly formulated.
- Subtle expression of negative attitude towards individuals/person/objects.
- A sentence that is near impossible to use positively (yet often negatively).

-2:

- Strong negative intensifiers or cursing.
- Direct/clear negative sentiment towards topics, people or objects.
- Many negative tokens e.g " :(:(:(:(:(".

Towards the end of the project and after having used the guidelines, we again calculated an alpha value on 100 new, unique sentences[7]. This time we reached an alpha of 0.651, which clearly indicates that the guidelines increased our agreement on our annotation. While we agreed more often, we were still not being consistent, which we will explain below.

2.7 Cross validating our dataset

An important decision we made later in the process was to cross validate all the ratings given by the other group members. We did this in pairs to make sure we had a discussion on the ratings we were unsure of and because there were indications that we had not followed the guidelines correctly. We did not initially think this was necessary, but since we found examples of sentences that were clearly violating our guidelines, it became evident that this was a necessity. An example of a sentence that a group member had labelled wrong throughout his labelling was "I agree", which was rated as neutral, but according to our rules this should be positive sentiment. We ended up revaluating around 5% of the total ratings, which also gave us a slight improvement of accuracy of $\sim 1\%$ points. This also made us question the correctness of the fine-grained polarity, but since we were purely focusing on the polarity, we did not seek to validate this.

[7]Link to these sentences: https://github.com/lucaspuvis/SAM/blob/master/SAM/Data/100_wild_sentences.csv

3 The Data

The dataset consists of 3345 root comments from Reddit and Facebook, and there's an uncounted number of root tweets, since these were automatically split from comments into sentences by the script that extracted them. The 3345 comments was collected from 239 different articles that were spread on 92 different tags/topics. All together this yielded us 9008 labelled sentences of which 51% are from Facebook, 27.6% are from Reddit and 21.4% are from Twitter:

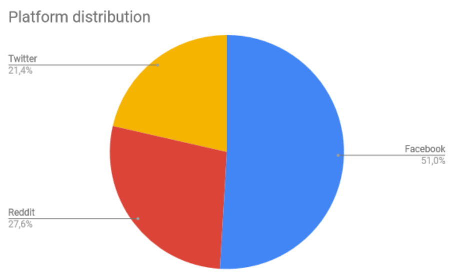


Figure 1: Distribution of sentences on social media

Of the 9008 sentences there are 1489 positive, 4057 neutral and 3462 negatives, which are 16%, 45% and 39% of the whole dataset respectively, and gives us around 22% more negatives than positives. A figure of the statistics can be seen on figure 2. This was no surprise to us since it's commonly known that there's a higher chance that people with negative sentiment will express their opinions [13].

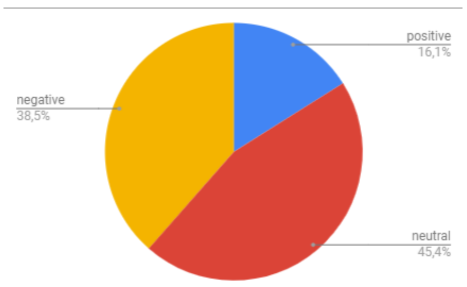


Figure 2: Distribution of our classes.

We discovered that there was a deviation of sentiment across the social media platforms, as seen on figure 3.

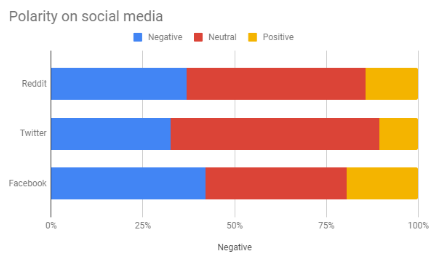


Figure 3: Comparison of sentiment across media.

It's quite clear that Twitter is the most neutral of the three platforms while Reddit is not far behind; however, the surprise is that Facebook actually has more negative sentiment than neutral. It's a surprise since we assumed that the neutral sentiment would be the largest classification on all of the platforms. There could be many reasons for this, where some individually or together may be:

- Media focuses on negative news, because they get more exposure, which in return spawns more negative sentiment.
- Facebook is the much larger platform and the average person just seek to express their opinion, and those who seek to discuss it go to another platform such as Reddit.
- Political subjects raises negative sentiment in general, and since the exposure on Facebook is far greater than the two other platforms, the general populations sentiment/opinion is clearer.

An initial hypothesis was also that Reddit - known for being better at discussions than Facebook - should have a higher average sentence count than Facebook. To no surprise we found that Facebook only had an average sentence per comment of 1.88 whilst Reddit had 68% more with 2.74 sentences per comment. We believe this is to be expected since well-minded discussions often provide well-formulated and longer arguments when people are discussing their views rather than just expressing them. An observation that supports this claim is that Facebook and Twitter, when seen as a whole text-document, only has a Lix-number of 25.07 whilst Reddit has 29.18. A lix-number indicates how hard a text is to read by for example accounting for long words(7 letters or more) [14]. Another observation that supports this claim is that Reddit had 54 different articles with 54 different topics, while Facebook with 185 different articles only had 38 different topics.

We discovered that our ten most frequent words were noticeably different from those of the written language in different aspects: Looking at table 1, which contains our 10 most frequent words and four frequency lists from sproget.dk[15], we see that the word "er" is not even present in the frequency list of Lemma 5000. It is less commonly used in the dictionary, in newspapers and in novels. We do think it relates to the nature of the social media platform, but we do not have concise evidence to support this claim. Our rationale is that the word "er" is often used to express a subjective opinion, and the comment section is a place of expressing subjective opinions, which consequently means the frequency list will be different from that of the common written language. This hypothesis is somewhat supported by the fact that the lix-number of our dataset combined is around 26.1, which gives a slight indication of the comments being more akin to spoken language than written language.

Table 1: Danish word frequency lists.

	Our Dataset	Dictionary[15]	Newspapers[15]	Novels[15]	Lemma 5000[15]
1	er	og	i	og	i
2	det	i	og	det	være
3	at	at	at	i	og
4	og	det	af	at	en
5	i	er	er	han	den
6	ikke	en	en	var	på
7	en	på	til	jeg	til
8	der	til	det	en	det
9	de	med	for	ikke	at
10	til	af	der	til	af

4 The Architecture

Initially we decided that we wanted to code our program in C#[8]. This was not an objective, analytical decision, but rather a sentimental question of preference and familiarity with the language. Our decision was also affected by the fact that C# is a very popular language in the enterprise environment[16], which we also wanted to accommodate with the tool being general purpose.

Quite quickly it became evident that the machine learning frameworks available for C# were lacklustre. There was a lack of documentation, which gave us issues with trying to set up the algorithms, and diversity, which gave us issues with testing the different approaches. We therefore decided to do machine learning with Python, since it has a rich, well-documented and diverse landscape of algorithms. This now meant we were building a C#-program that will be utilizing Python. How we did this will be described later in section 4.9.

4.1 SOLID-principles

We wanted to keep our code in line with the SOLID-principles [17] as much as possible. The SOLID-principles are a set of guidelines that can help developers write better, cleaner and more maintainable code. SOLID is an acronym of the following principles:

- Single responsibility
- Open-Closed
- Liskov substitution
- Interface segregation
- Dependency inversion

These principles are described in the upcoming sections.

4.2 Single Responsibility Principle

This principle dictates that a class, module or function should be responsible for one single part of the functionality of a program. For example, if we want to load data, train a model and save the model (described in section 5), we need to write at least three functions. One for loading data, one for training the model, and one for saving the model. Of course, a function can be invoked by one of the others without violating the principle, but one function is not allowed to handle everything. [17]

4.3 Open/Closed Principle

This principle states that software should be open for extensions but should be closed for modification. This means that it should be possible to extend the behaviour of such an entity, without modifying its source code.

[8]https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

4.4 Liskov Substitution Principle

This principle extends upon the Open-closed principle, and adds the constraint that if a type S is a subtype of type T, then objects of type T can be replaced with objects of type S without changing the properties of the program. This means that everywhere the program expects some instance of T, we can use S instead, without the user noticing. We use the Liskov substitution principle when predicting sentiment of different sentences. All our classifiers have the same interface and output, and therefore we can use the classifier specified by the user at runtime.

4.5 Interface Segregation Principle

This principle states that no client should be forced to depend on methods it does not use, and therefore it is better to create many small and specific interfaces, rather than larger ones. Following this principle helps lower coupling[9] and therefore makes it easier to refactor or change the behaviour of the program.

4.6 Dependency Inversion Principle

This principle states that it is better to depend on abstractions rather than concretions. Following this principle allows for easy switching between different modules implementing the same interface. We use this principle almost everywhere in the program. When we calculate the sentiment in a set of sentences, we use an implementation of the IEvaluator interface. This allowed us to implement multiple evaluators, and switch between them at runtime, rather than having to use the same one all the time.

4.7 The Pipe

We went with the pipe design pattern because this would enable us to create abstractions that easily adhered to the Liskov Substitution principle and the Single Responsibility principle. This allows us to run any evaluator with any given file using only one line of code, as shown below. This proved to be very useful, as switching between evaluators became very simple with this implementation.

```
ThreadedSentenceLevelPipe(CSVReader.ReadCommentsYield, filename, evalr).ToList();
```

The pipe pattern resembles the idea of an assembly line. For the programmer this means having clear, distinctive computations/stages that needs to be done in a sequence. The pattern is recognized to be especially effective when: [18]

- The number of calculations is large compared to the number of stages.
- It is possible to dedicate a processor to each element, or at least each stage, of the pipeline.

This fits our task that has clear stages that we need to put our data through: Load the data, pre-process it, evaluate it, analyse and output it again. The idea

[9]In software engineering, coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules.

is also to create an abstraction that makes it easy for anyone to execute this sequence. In our case we implemented the pipe in such a way that we can easily swap a stage with another that does the same job e.g two different classifiers.

Seen on figure 4 is a UML diagram of how our pipe interacts with the program. For simplicity — importing and exporting has been omitted, but those are of course also a part of the process.

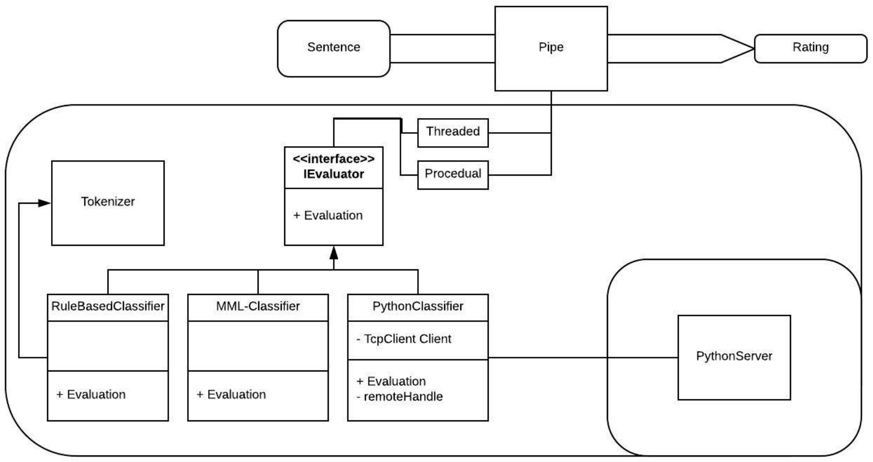


Figure 4: UML Diagram of how the pipe works

4.8 Settings

One of the goals of the program is to be general purpose. This also implies a level of customizability for the user, such that they can change the behaviour of the program to suit their needs. Most of the changes are limited to the Tokenizer, which is currently only used by our own rule based classifier described in section 5.2. There are two different ways of altering the behaviour of the program:

1. By passing different arguments to the program at runtime.
2. Or by changing the settings within the settings.json-file.

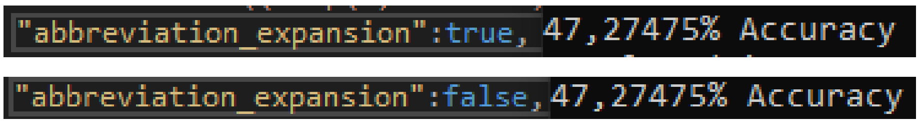
Not all our settings will be listed, but for the above two categories we have:

1. Do predictions on input with the user specified classifier or take the input, shuffle it and return a document containing 10% of the data and another with the rest. This is directly linked to training the different machine learning algorithms and testing them.

```
SAM svm -r /Data/MyDataToLabel.csv
SAM data -r /Data/MyDataToSplit.csv
```

2. • Multi Threaded or procedural — Please refer to Section 4.10 about parallelism.

- Expand abbreviations — Our rule based classifier can expand abbreviations, such that the lexicon will be able to match on the words. If the user wishes to parse a full comment to the program(multiple sentences), an abbreviation will inevitably make the program split a sentence at the punctuation, since the program can't tell the difference between an abbreviation or the end of a sentence. Ultimately, this will not make a noticeable difference(if any) for the accuracy of our rule based classifier right now; however, since the tokenizer has this functionality, it may improve the accuracy of a future custom classifier as seen on figure 5.
- AbbreviationFinder Regex — The user is able to specify their own regular expression (regex) for finding abbreviations with the rule based classifier.
- Compare — The program can either label a dataset or load in a labelled dataset and determine the accuracy of the program.
- WriteNonMatchedTokens — This setting outputs a frequency list (in absolute numbers) of all the tokens that were not matched by the rule based classifier. This was a very useful tool when we were expanding our lexicon.



```
"abbreviation_expansion":true, 47,27475% Accuracy
"abbreviation_expansion":false, 47,27475% Accuracy
```

Figure 5: Comparison of Abbreviation Expansion

4.9 Python

Initially we embedded Python in our C#-program using an Adapter pattern. The adapter was created in such a way that it was interacting directly with a process that had a python terminal running with the Adapter pattern. [19] This very quickly ran into some unforeseen and very critical issues: There was a character limit when passing arguments to the Python-terminal, which meant that we had to split up the sentences dynamically to fit within the length. This was mostly a nuisance, but still doable to program around. We could not keep the process running to continuously evaluate the arguments that we wanted to pass. We had to open a new process, load the model again and then rate the limited amount of sentences mentioned above, which resulted in an unfeasible running time. We do not actually know how long it would have taken to rate all our sentences, because we stopped after 10 minutes and had only evaluated 378 sentences. Since we had 5500 sentences at the time of this experiment, it would have taken approximately two and a half hours to evaluate all sentences.

Instead, we created a Python server, which all our Python-related classifiers connect to when they want to do a request. This also now meant that in regard to the Open/Closed principle, anyone can easily add a different type of remote classifier that they want to interact with. It also satisfied the Liskov substitution principle, because the classifiers can be used interchangeably. This construct does have some limitations, which will be mentioned in the parallelism section.

For the classifiers themselves, the way we decided to build the python part of our program is very simple: Each classifier has its own script that handles training and evaluation (graphs and statistics) of the classifier, and then saves it using the dump function of the joblib package. The neural network saved the model using a slightly different method, namely the built in saving function in Keras. This essentially saves our trained classifier on disk with a small file size, allowing it to be loaded whenever needed. The idea behind this is threefold:

- First and foremost, we do not have to train a classifier each time we want to use it. The only time we would want to do that is to change settings, or we have a new dataset.
- We can have multiple instances of the same classifier with different settings.
- Lastly, we can close the program without losing our classifier — it will persist on the drive where the program is located.

The classifier is then saved with one of the following filenames:

- classifiername_pipeline.joblib
- classifiername_model.h5

As an example, the SVM will be saved as svm_pipeline.joblib. The reason it is called pipeline is because we use sklearns pipeline object. It allows us to create a list of transformers (like our TfidfVectorizer) and a final estimator (like our LinearSVC). Then we can call the transform and fit functions on the pipeline, just as we would on the individual transformers and estimators. Since it is not only the final estimator we wish to save, because the processing of the data is just as important, it allows us to save and load a single, compact file, instead of one for each transformer and the final estimator.

The saved file is used later in a script that handles phrases as input from the user. This script checks which classifier the user wishes to utilize for predicting the sentiment on a certain phrase (except if it's the rule based classifier, this is handled by the C# part of the program), and then loads the corresponding pipeline. Then the script gives the phrase to the pipeline that has been loaded in, which then predicts the sentiment of the phrase, and sends it back to the C# part of the program, which finally hands it back to the user.

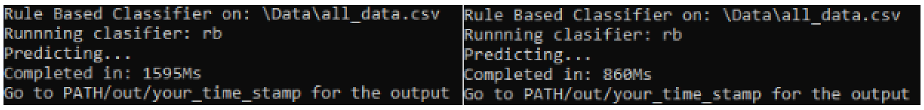
4.10 Parallelism

From the Pipe-pattern section it was noted that we have two different ways of running the pipe. One is doing it procedurally — handling one sentence at a time — and the other is a multi-threaded approach, which means handling multiple sentences at a time. In our program there are advantages and disadvantages to both approaches:

	Procedural	Multi-Threaded
Advantage	Maintain order of data	Greatly decreased running time
Disadvantage	Running time is slow	Unordered data in output

Table 2: Parallelism Overview

On figure 6, a running time comparison of the two is shown. This design decision reduced the running time of the program by 47%.

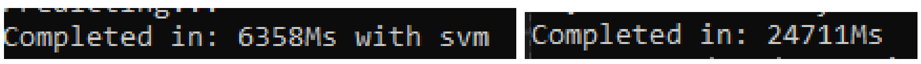


```
Rule Based Classifier on: \Data\all_data.csv
Runnning clasifier: rb
Predicting...
Completed in: 1595Ms
Go to PATH/out/your_time_stamp for the output
```

```
Rule Based Classifier on: \Data\all_data.csv
Runnning clasifier: rb
Predicting...
Completed in: 860Ms
Go to PATH/out/your_time_stamp for the output
```

Figure 6: Comparison of runtimes of the program.

The multi-threaded approach is currently limited to all the classifiers that are part of the C# program. Due to the difficulties of using Python together with C# we chose to force procedural on python classifiers, because several factors made it unfavourable to make it so: Firstly, it's very resource demanding to do operations that talks in between programs or the operating system. Running time itself increases when the program has to interact directly with the operating system. Secondly, it would be unavoidable to create a bottleneck and ultimately increase the running time drastically. This occurs because we're running the python server locally. We can't run many threads on the server because by limiting the thread supply and hence increasing the load on the CPU, we're bottlenecking the C#-program. Furthermore, we can't run multiple threads in the C#-program because the server then can't handle all the incoming requests concurrently without being bottlenecked of same previous mentioned reason. Figure 7 shows how allowing a multithreaded approach currently will yield a far slower running time.



```
Completed in: 6358Ms with svm
```

```
Completed in: 24711Ms
```

Figure 7: Comparison of runtimes of the server.

4.11 Refresher

In this section we introduced why we chose a C# program that utilizes a Python server. We introduced the theory SOLID that affects our design decisions as well as some settings that are available to a user. Lastly we introduced our pipe-pattern together with our parallelism.

5 Classifiers

5.1 Primitive Classifiers

For statistical purposes we have implemented two primitive classifiers, one called Baseline Classifier and one called Random Classifier. These are implemented to ensure that our more advanced classifiers are better at determining sentiment than randomly guessing or simply choosing the most frequent sentiment.

5.1.1 Baseline Classifier

The purpose of the baseline classifier is to guess for the sentiment which appears most frequently in the dataset, which in our case is neutral. Since the baseline classifier statically guesses for the sentiment which appears most frequently, the accuracy will always be equivalent to the amount of that specific class within the dataset. In our case, this means we hit 45.37% with the results seen in figure 3.

	Predicted		
Actual	Negative	Neutral	Positive
Negative	0	3472	0
Neutral	0	4087	0
Positive	0	1449	0

Table 3: Results of the Baseline Classifier.

5.1.2 Random Classifier

The purpose of the random classifier is to guess randomly for a sentiment. The random classifier will guess randomly between the three options, and averages to around 33.33% accuracy, since it has 1/3 chance of correctly guessing the sentiment. Given in figure 4 are the results of running the random classifier on our dataset, which in this case resulted in an accuracy of 33.09%.

	Predicted		
Actual	Negative	Neutral	Positive
Negative	1167	1164	1141
Neutral	1375	1361	1351
Positive	530	466	453

Table 4: Results of the Random Classifier.

5.2 Lexical Analysis

For the naive lexical analysis of sentiment we have created a classifier called Rulebased Classifier. Our rule based classifier uses a Lexicon of words to determine the sentiment of a given sentence. We have created a dictionary of words most commonly used in positive or negative sentiment sentences, and associated these words with a numerical value, ranging from -2 to +2, indicating the sentiment of said word.

The classifier takes a sentence as a string, and then runs that string through our Tokenizer. In the end, a list of Tokens will be created, which contain the word and the sentiment of said word, if a sentiment is found in our lexicon. An example of this process can be seen in table 5. The classifier will then run through the list of Tokens, accumulating their total score on the way, determining a decimal value. If this value is greater than zero, our rule based classifier will determine the sentiment as positive. If the value is less than zero, our classifier will determine the sentiment as negative. If the value is zero, the classifier will determine the sentiment as neutral.

Regular Sentence:	Det var godt.			
Tokenized Sentence:	["det", 0]	["var", 0]	["godt",2]	[".",0]

Table 5: Tokenization of a sentence.

For the lexicon we found an already-made lexicon online created by Finn Årup Nielsen [20] (hereafter AFINN). This lexicon provided a good baseline, but did not fit with our rating system, as it contained ratings higher than 2 and smaller than -2. We went through the entire lexicon and re-evaluated all words to make sure we agreed with the sentiment given by AFINN, and also added missing conjugations to words present in the lexicon. We've also added 688 new words we have found through our data, specifically words used quite often in political contexts which also are used in either a predominantly negative or positive context. In unison with this, we also added specific emojis which are used predominantly in a negative or positive context. There are not a lot of examples of emojis with this quality, but a few examples are the angry emoji, the thumbs up emoji, and the thumbs down emoji. After re-evaluating all the sentiment and adding new words to the lexicon, we went from 3552 words to 4240 words. 2831 of these words are rated to be negative, and 1409 of these words are rated to be positive. This means that our lexicon contains approximately 66.6% negative words and 33.3% positive words.

Using the unedited lexicon created by AFINN we get an accuracy of 52.76% with the results seen in table 6, and using our lexicon we get 55.58% accuracy with the results seen in table 7. This is in total an accuracy increase of 2.8% points, and as can be seen on the stats, we now correctly guess more negative and positive sentences, but lose out on a few neutral sentences.

	Predicted		
Actual	Negative	Neutral	Positive
Negative	1307	1569	596
Neutral	579	2710	798
Positive	99	614	736

Table 6: Results using AFINN's lexicon

	Predicted		
Actual	Negative	Neutral	Positive
Negative	1616	1311	545
Neutral	637	2639	811
Positive	113	584	752

Table 7: Results using our new lexicon

We quickly noticed that there were certain scenarios that our rule based classifier could not handle with a simple lexicon. Intensifying words such as 'meget' were not accounted for, and words such as 'ikke' was not negating the sentiment of following tokens. We also noticed that a lot of Danish sayings do not contain any words which contain sentiment by themselves, but could drastically change the sentiment of an entire sentence when used in unison. We also noticed that a lot of sentences ended with three or more periods if they disagreed or showed displeasure towards the article, and a lot of sentences contained a mixture of question marks and exclamation marks if they were outraged.

It would not be possible to handle these scenarios using the current strategy of using a lexicon of commonly used negative and positive words. It's not possible to give a period sentiment, as it would skew the results of all other sentences using punctuation properly. Neither would it be possible to do for exclamation- and question marks, as same logic would apply. Sayings mostly contained words with no sentiment, and therefore it would be inaccurate to give them sentiment individually, and negations also did not have sentiment themselves, but only modified sentiment of other tokens. For intensifiers, it would also not be possible to give the word by itself sentiment, as intensifiers can intensify both negative- and positive sentiment tokens. This meant that we had to come up with a way to handle these edge cases, since our current approach was insufficient.

These ideas are implemented in the form of *Modifiers*. A modifier contains both a *modValue*, which is the value used to change the sentiment of certain tokens, and a *lookupValue*, which indicates how many tokens you look up and change the sentiment of. There are two modifiers which require additional parameters, such as *Repeating*, which requires the character to check for and the amount of repeated characters required to modify the sentiment, and *Vending*, which requires the list of words in said phrase. In total, five different modifiers were implemented, all handling the specific scenarios described previously.

5.2.1 Negations

Negations are Modifiers that negates the sentiment of subsequent tokens. Negations are used to make sure that the subsequent tokens change sentiment from positive to negative or vice versa when encountering a word which negates sentiment. When the program encounters a negation, it will look ahead depending on the lookup value of this given negation, and negate all words it encounters, as long as they also have sentiment. An example of such a word is 'ikke'. If you write "Det var godt", the overall sentiment is positive, but if you write "Det var ikke godt", the overall sentiment becomes negative. This example can be seen in table 8. With negations implemented our classifier can now handle such occurrences. Negations have been implemented to stop if a sentence ends, to make sure that the negation does not incorrectly change the sentiment of tokens which are not in the same sentence, should it for example contain parentheses or a quote.

0	0	2	
Det	var	godt	
0	0	$\rightarrow$	2·(-1)
Det	var	**ikke**	godt

Table 8: Negation example with sentiment.

5.2.2 Multipliers

Multipliers are Modifiers which intensify the sentiment of subsequent tokens. This is used mainly for intensifiers to make sure a higher sentiment is returned. When using multipliers and encountering an intensifying word, it will now look ahead and intensify the sentiment of following tokens, which means tokens with sentiment will be valued higher if they're being intensified by an intensifier. This ensures that intensifying words will correctly change the sentiment, as intensifying words cannot themselves be given a sentiment value due to their ambiguous nature. An example of such a word is 'meget'. If you write "Det var godt", the overall sentiment is positive, but if you write "Det var meget godt", the sentiment of the latter sentence is more positive than the former. With multipliers implemented, it makes sure that our classifier can handle such occurrences. This example can be seen in table 9. Multipliers have been implemented to stop if a sentence ends, to make sure that the negation does not incorrectly change the sentiment of tokens which are not in the same sentence, should it for example contain parentheses or a quote.

0	0	2	
Det	var	godt	
0	0	$\rightarrow$	2·1.5
Det	var	**meget**	godt

Table 9: Multiplier example with sentiment.

5.2.3 Phrases

Phrases are used to account for a collection of words which individually do not have sentiment, but when used together in a certain order portray sentiment. These have been implemented so it's both possible to look ahead but also look behind when checking for phrases. This is because certain phrases start with words which are very common in the Danish language, and this reduces computation time, since we avoid checking for a phrase unnecessarily many times. In this case, we check for the least common of the two words and simply look backwards for the remaining words to make sure the entire phrase is there. If we find such a phrase, we will correctly change the sentiment of the last token in the phrase. An example of this is the saying 'hovedet under armen', which means to do something without thoroughly thinking it through, and is normally used in negative connotations only. An example showcasing a sentence with this saying is seen in table 10. The three individual words which make up this saying do not have sentiment themselves, but when combined in this order the sentiment becomes negative.

0	0	0	0	0
Du	har	hovedet	under	armen
0	0	$\rightarrow$	$\rightarrow$	-2
Du	har	**hovedet**	under	armen

Table 10: Phrase example with sentiment.

5.2.4 Special

The special modifier is used to account for a large string of exclamation marks or question marks used in unison. When met with either an exclamation mark or a question mark, we build a string containing all the following exclamation- and question marks of the sentence. When this string has been built, we check it against a regular expression to see if it matches. If it matches, we will change the sentiment of the token to -2 and nullify all trailing identical tokens to make sure they do not contain sentiment. Question marks and exclamation marks should not contain sentiment, but the nullification is done as a safety measure to make sure they definitely do not.

5.2.5 Repeating

Repeating modifiers are used for any repeating characters, such as question marks, exclamation marks, or periods. We noticed that using repeating characters can express certain sentiment, for example using three or more periods in a row can express disappointment or negativity towards a given subject. An example of this is the sentence ”Imens kan DF, SD, Afd, Front national osv jo takke for ubetalelig reklame.....”. If we ignore the trailing periods, the sentence is positive, but in reality the sentence is negative or sarcastic. Therefore, we’ve implemented a way to check for repeating characters and then modifying the sentiment if a character is repeated a certain amount of times.

5.2.6 Results of Modifiers

The modifiers are stored similarly to our regular lexicon, but contains more information given the nature of modifiers. This modifier lexicon is loaded the same way as our regular lexicon, and our current modifier lexicon only contains 49 entries, with several more ideas not implemented in the lexicon.

The implementation of Modifiers has slightly changed the way the rule based classifier behaves. It still splits the sentence into individual tokens and gives them sentiment, but before counting the sentiment it will check the word contained in that token if there is a modifier for it. If there is, the modifier will be executed, which in turn can change the sentiment of the current token, any future tokens or both. After the tokens have potentially been modified from executing this Modifier, the sentiment will be counted as normal. Since it is possible for there to be more than one Modifier on a given word, we make sure to execute all modifiers associated with the word before continuing the summation of sentiment.

As showcased during section 5.2.3 there's the phrase 'hovedet under armen'. We also have 'syg i hovedet' as a modifier, which also triggers on the word 'hovedet', which is an example of how one word can have multiple modifiers.

Before Modifiers were implemented the accuracy of our implementation was 55.58%. When using the bare-bones modifier lexicon, the accuracy jumps to 56.27%, which can be seen in table 11. This is in total an increase in accuracy of 0.69% points. The lexicon used to test the accuracy of modifiers only contains 49 entries as previously mentioned, but still manages to make it so our rule based classifier correctly guesses the sentiment of more sentences. This is a good indication that the modifiers are effective. There is a 104 sentence jump when it comes to correctly guessed negative sentences, but there's a decrease in positive sentences by 3, and neutral sentences by 39.

	Predicted		
Actual	Negative	Neutral	Positive
Negative	1720	1246	506
Neutral	685	2600	802
Positive	123	577	749

Table 11: Results using Modifiers

A good example sentence to showcase the effectiveness of modifiers from our dataset is 'det er jo ikke for sjov, at det britiske folk stemte sig ud af EU's formynderskab, der har taget overhånd.'. Before the implementation of Modifiers our rule based classifier determined the sentiment of this sentence to be neutral, but with a Negation Modifier in place for the word 'ikke', the sentiment is correctly determined to be negative, since 'sjov' has positive sentiment and is evened out by 'formynderskab' having negative sentiment.

We are satisfied with the accuracy of the Lexical Analysis. While the addition of a lexicon containing Modifiers proved to improve accuracy, it contained very few entries and therefore could not have a significant impact. After reading through some unmatched tokens as mentioned in section 4.8, it becomes clear that increasing the entries in the lexicon would yield better results. In our pipeline we have 31 more sayings which could be included in our modifier lexicon, but were not included due to time constraints.

We also discovered a *wordnet* containing Danish words specifically developed to be used in conjunction with the development of software for processing of natural language by the name of *DanNet.*[21] We're certain that this wordnet could be used in conjunction with developing the lexicon for the Lexical Analysis part of our classifiers. If we saw a word being used frequently in the dataset, we could check the wordnet to see which words were similar or closely related to it.

5.2.7 Recap

In this section we introduced our lexical classifier. We described how we created a better version of a pre-existing lexicon and how we increased our accuracy by implementing Modifiers.

5.3 MML

In the early stages of the project we tried implementing a classifier using the ML.NET package from Microsoft in a classifier we henceforth will call MML (Microsoft Machine Learning). Using this would allow us to create the entire program using C#, which would save us the trouble of finding a way to make C# and Python work together. Although this would be positive for the project, there were some difficulties in working with the framework.

We implemented a classifier through the framework and got it working quite fast, but we did not achieve the results we were expecting. The documentation is lacklustre at best, and therefore it is very difficult to make any changes or find meaningful information which could aid in improving the classifier. Furthermore, we had a lot of errors when trying to extend the classifier with new features when trying to improve its accuracy, which were very hard to fix given the previous lack of documentation. It should be noted that the version of ML.NET that were available to us at the time was a pre-release (v0.11[22]). We collectively decided that the best action moving forward would be to channel our attention towards the SVM implemented in Python. This is partly because we had a fair amount of experience using these libraries already, and because it would then be possible to extend our SVM with new features to hopefully improve the accuracy. We did go back to testing MML towards the end of the project, and surprisingly achieved results that were close to our best SVM. An example of training the classifier on our data and testing it with our test data, we achieved an accuracy of 63.54% with the results shown in table 12.

	Predicted		
Actual	Negative	Neutral	Positive
Negative	166	78	31
Neutral	138	248	64
Positive	19	12	182

Table 12: Results using MML classifier.

5.4 The SVM

5.4.1 Preliminary decisions & notes

As described in chapter 4, we began our project with the idea of using C# for every part of the program. When we stumbled into the problems described in aforementioned chapter, we switched to using Python. Specifically, we chose to use the scikit-learn library (hereafter sklearn). This library includes a vast array of classifiers of ways to preprocess data for said classifiers. For the data preprocessing, we went with sklearns TfidfVectorizer[23].

A problem with using a SVM is that it's inherently meant for binary problems (eg. is it true or false, is it a car or not etc.). Since we have five classes if we utilize our fine-grained annotations, or at the minimum three classes if we remap two's and minus two's to one's and minus one's, it's still one class too many for it to be a binary problem. An important method to mention before that is that you can handle this as a one-versus-rest problem, meaning that you train a classifier to see if a given input belongs to a specific class or not. Sklearns SVM implementation handles this by training n_classes one-versus-rest classifiers, meaning that it will eg. end up with a positive-or-not, neutral-or-not and negative-or-not classifier, and predict over them until it finds the correct class. An issue with this is that we have no transparency. Two classifiers could claim a phrase belongs to them, i.e. one claims positive and another negative, and we do not know how it handles such issues. We will come back to this problem later.

5.4.2 Data preprocessing

For shrinking our classes from 5 different to simply negative, neutral and positive, we check if the number is either 0, negative or positive. If it's either of the two latter, we just label them -1 or 1 respectively. The reasoning behind this is that we don't have enough data to train a classifier with both positive (1) and very positive (2) classes. By going from fine-grained labels (5 different classes) to only 3 classes, we improve our accuracy by an average of 15%. It is important to note that using 5 is only worse in the sense that our classifier has a harder time distinguishing the 5 classes from one another, when looking at the polarity of the predictions — if they were positive, negative or neutral — it remains the same as when squishing the classes.

As a very first step, we lowercase all letters in our sentences. Without this, the same word would be seen as a different word depending on how uppercase or lowercase letters are used. One could argue that a full capital version of certain words could hold a different sentiment than one without, but it would also see words in the start of sentences as different words, just because their first letter is in uppercase. This would greatly increase the number of words we have not yet seen before and therefore increase the difficulty of classifying them.

At this point in the processing, we attempted stemming the words. Stemming is removing the suffix from a word, leaving you with only the root. It would take words like "haltede" and remove the suffix, leaving us with "halte". The issue with this is that some words change drastically depending on what tense it is in. The word "spurgte" cannot be converted to "spørge", just by removing the suffix. Stemming words could potentially remove the problem of not having seen a specific tense of a word. If our vocabulary includes "love", but not "loved", predicting on a sentence that uses "loved" could give wrong results. But if it is stemmed, then it would simply remove the suffix "d", and end up with "love" which it has already seen. However, with our dataset, it didn't do a measurable difference. This could come down to our vocabulary being very large, the different tenses of the word not looking alike or spelling errors (of which there are many).

We have made a list of stop words, which are essentially words without any significant sentiment that we remove from the sentences. It can also be words that are used very frequently. Example of such words are: "er", "du" and "at", which also coincides within our top 10 most frequent words as seen in 1 . The idea is that none of them carry any information that we need, and therefore are not important to us. Yet if we do not remove them, when it has been converted to a vector, it would have to end up with a representation of either positive, negative or neutral, which could distort our results.

After having applied this cleaning to every single sentence, the result is then converted into a sparse matrix representation of the token counts. This means that the number of a said token is increased by one every time it is seen. We also take a look at n-grams, so that our classifier may better understand words in context to one another. This means that the sentence “Hej med dig”, if we have an n-gram range of 1 to 2, would be split into the following tokens “hej”, “med”, “dig” but also “hej med” and “med dig”. That way, it can distinguish how words are used in certain contexts, and how their sentiment may change depending upon that. What we found to be the perfect range for our dataset is 1 to 3, meaning that it creates unigrams, bigrams and trigrams. Anything less is not enough to convey the context, and anything more decreased our performance while also taking longer to train.

Something different we found to give good results is to tokenize letters instead of words, and use a much larger n-gram range. So instead of looking at a word in context to its surrounding words, we look at letters in context to its surrounding letters within a word. What we end up with is tokens of individual letters, and tokens of letters of length 1 to n. We chose to only look at single words, meaning that if we try to make a trigram of the word “og”, it will only generate the tokens “o”, “g”, and “og”.

The difference between these two are that tokenizing by words will give the classifier a better understanding of which words are used in context to the words around it, whereas tokenizing by characters in a word might help with spelling mistakes.

What then happens is that it the sparse matrix is transformed to a normalized tf or tf-idf representation. As the documentation for sklearns TfidfTransformer states: *“The goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus.”* [23]
The TFIDF formula is as given:

- **TF**: Term frequency, measures how frequently a term occurs in a document. Since documents differ in length, it is possible a term would appear more often in longer documents than shorter ones. Therefore, the term frequency is often divided by the document length as a sort of normalization *TF(t) = (Number of times a term appears) / (Total number of terms in document)*

- **IDF**: Inverse Document Frequency, it measures how important a term is. Without this, all terms are seen as equally important. However, certain terms, such as the ones in our list of stop words, are of little importance but may often appear — with that in mind we need to weigh down frequent terms, but scale up the rarer ones. *IDF(t) = log e(Total number of documents / number of documents with term t in it)*

- Use idf: We can choose not to use the inverse document frequency

- Smooth idf: Add one to every document frequency. This means that it appears as if an extra document was seen containing every term in the collection exactly once. According to sklearn documentation, it prevents zero divisions.

- Sublinear tf: Instead of using the normal tf, replace it with 1 + log(tf) for sublinear scaling.

Choosing which of these parameters to use will be discussed further on. What we end up with after this is our data in a higher dimensional space.

For the classifier itself, we started with sklearns SVC**svc** (C-Support Vector Classification), and the first thing we needed to ask ourselves was which kernel to use. To figure this out, we tried with a bunch of different kernels, namely the radial basis function (RBF[24]) kernel, a polynomial kernel[25], and a linear kernel[26]. Which one to use will heavily depend on how our data is represented in its high dimensional space. After having tried out these different kernels, we found that the linear kernel gave us the best results, with a 10-15% accuracy increase over the other two. To speed up training, we then switched to sklearns LinearSVC, which is built with a linear kernel in mind.

To better choose which to use, we used sklearns form of hyperparameter tuning: GridSearchCV. It takes a list of whatever parameters relevant to the classifiers and transformers (As our TfidfTransformer) that we want to try out, and it tries out every single possible combination of them, while cross validating with the stratified k-fold method. This lets us tailor the parameters to our data at the expense of training time, while also letting our classifier see all the data in the set.

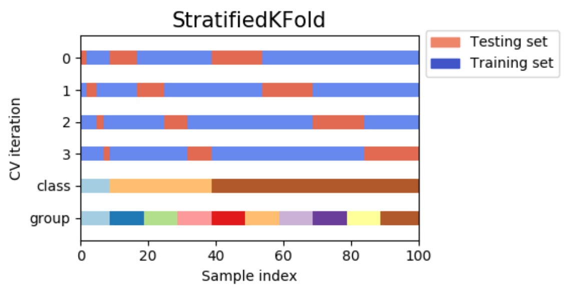


Figure 8: Image showcasing Stratified K-Fold.[27]

The k-fold cross validation is a way to split the data into test and training parts. It splits the data into k samples, and then uses one fold for testing while using the rest for training. It does this until every fold has been used for testing, meaning that the classifier will have seen all the data at the end of the cross validation. Stratified sampling essentially means that each fold will have an equal percentage of the different classes, as can be seen in figure 8. Each testing set from each iteration contains the same proportional amount of each class as the rest. For our data, since we have a disproportion of neutral and negative to positive, it ensures that each fold contains positive sentences. Essentially, we can see how our classifier performs on each class, instead of having the possibility of having all the positives in the training set, and none in the test set.

5.4.3 Custom multiclass SVM

As described in chapter 5.4.1, there is a lack of transparency on the internal picking of a prediction when using our standard SVM. We do not know how it acts on conflicting predictions between the different classifiers on the same phrase. That was the motivation for us to create a script where we have full control over these decisions. The core idea behind it is that we have two classifiers: one checks if it is negative or not, and the other checks if it is positive or not. If it is neither positive nor negative, then it's neutral. The data preprocessing will be the same as for the ordinary SVM, with the only difference being how the data is labelled.

The negative-or-rest classifier is trained on the same dataset as the positive-or-not classifier, but the labels are altered for the two. For the former, the labels are either -1 for negative, 0 for non-negative. For the latter, it's 1 for positive, 0 for non-positive. This means that for the former, both positive and neutral are labelled with a 1. The reason for this is the following: When predicting, both classifiers will return their prediction — either -1 or 0 for the negative-or-rest, or 1 and 0 for the positive-or-rest — which will be added together. If the result of adding these two together is -1 (negative-or-rest predicted -1, positive-or-rest predicted 0, -1 + 0 equalling -1), then it's negative, and vice-versa if the result is 1. Here is a full list of the results and what they mean:

- -1: Negative-or-rest has predicted negative, positive-or-rest has predicted non-positive. This means that it predicts negative.
- 1: Negative-or-rest has predicted non-negative, positive-or-rest has predicted positive. This means that it predicts positive.
- 0: This can mean one of two things: either both claimed the phrase does not belong to their primary category, and thus have both returned 0 (0 + 0 = 0), OR it means that both claim it belongs to their respective primary class (-1 + 1 = 0). In this case, we simply claim that it's neutral.

So with simple arithmetic, we have created the transparency and control that we wish to have over the classifiers. What more is that we have done it with one classifier less than standard sklearn LinearSVC (recall that it trains the same amount of classifiers as there are classes).

5.4.4 Kernel choice

The very first thing we wished to figure out is how our data would look like as a 2D representation. The thought behind this was that we would be better suited to figure out which kernel to use if we could observe how our data behaves when it has been transformed into a higher dimension space. To this end, we used t-distributed stochastic neighbour embedding (TSNE[28]) to get a better look. As for the parameters, we used the default ones, except decompose_by, which we set to 25.

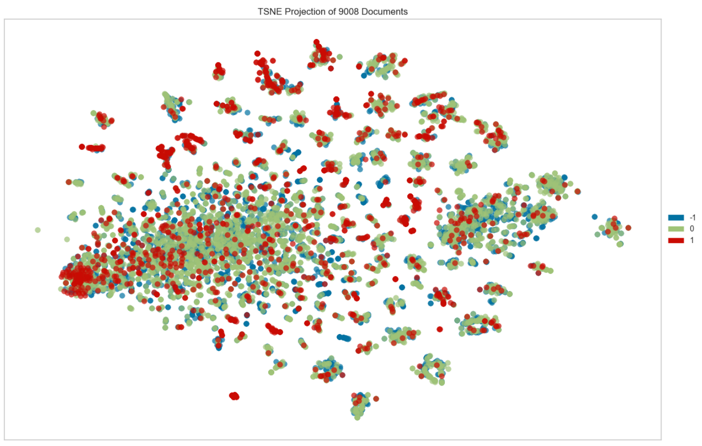


Figure 9: Projection of our data.

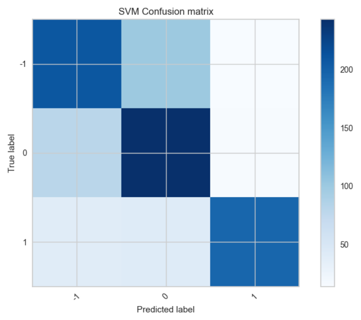
What figure 9 shows is a projection of our data when using a word tokenizer with an n-gram range of 1-3. The number of dimensions this data exists in normally is 78000, which is the number of features we have in this case. As can be seen from this graph, the data overlaps a lot. This could either mean that our problem is non-linear, meaning that we cannot separate the different classes with a linear kernel, or it could also be that the way we perform dimensionality reduction doesn't properly represent the data when it has been reduced to a 2D representation. While it doesn't look like the problem is linearly separable, we tried different kernels to see which ones work the best. We tested a radial basis function (RBF) kernel, a linear kernel and a polynomial kernel. What then comes as a surprise, is that even though the illustration of the data makes it seem non-linear, the linear kernel outperforms every other kernel. Here are the results of the different kernels

- Linear kernel: 65% accuracy
- RBF kernel: 45% accuracy
- Polynomial kernel: 40% accuracy

From these results, we decided to go with a linear kernel and sklearns linear SVM implementation (as described earlier)¸ which has the bonus of a quicker training time.

5.4.5 Classifier confusion

After having trained our classifier, we decided to create a confusion matrix to test our classifiers accuracy and performance over the different classes.



(a) Image of the confusion matrix.

	Predicted		
Actual	Negative	Neutral	Positive
Negative	209	100	14
Neutral	80	243	15
Positive	40	42	195

(b) Table of the confusion matrix.

Figure 10: Confusion matrix of our classifier.

The primary issue with our classifier is that it has trouble distinguishing between neutral and negative sentences.

5.4.6 Tokenizing words or characters

As described earlier, we can either tokenize the given phrases on a word-by-word basis or on a character-by-character basis. To test this we've run both methods 100 times and written down the results. The results from word tokenizing can be seen on table 13.

Accuracy	word	
Minimum	Average	Maximum
56.38%	60.63%	64.15%

Table 13: Results with word tokenizing.

When utilizing character tokenisation, we get the results as seen on table 14.

Accuracy	char_wb	
Minimum	Average	Maximum
60.49%	66.12%	69.03%

Table 14: Results using character tokenization.

Tokenizing with characters gives us an average increase of 6% points over tokenizing with words.

5.4.7 Names and topics

An observation we made during testing is that after training, names and topics that were discussed in one topic could skew the prediction of testing sets that contain the same names and topics. As an example, the classifier thinks that

”Pia Kjærsgaard” is positive, and the name has such a force behind it that almost every sentence with that name in it that we tested with would be classified as positive, even though they contained some rather nasty words. So if we wanted to predict the sentiment of a phrase where the subject is Pia Kjærsgaard, like the sentence ”Pia Kjærsgaard er forfærdelig”, even though the sentiment of the phrase towards her is negative, her name has been used primarily in a positive context, and it would predict wrongly. Other names and topics could be Esben Lunde Larsen or vaccines.

By updating our list of stop words to contain topics and names, the lower bound of our confidence interval increased from 60.5% to 62.62% as can be seen on table 16. That is, our lowest score increased by 2% points. At the same time, our highest score increased by almost 1% points. The average score decreased slightly.

Accuracy	char_wb	
Minimum	Average	Maximum
62.62%	66%	69.9%

Table 15: Results with updated stop words.

The better confidence interval and average accuracy is only one of the reasons we've chosen to go with tokenizing characters instead of words. Another reason is that we believe it handles light spelling mistakes better than word tokenizing. Where word tokenizing only takes into account the words already seen, character tokenizing will be able to correct slight spelling mistakes on some words by comparing parts of the word to something it has already seen. As an example, both methods of tokenizing will result in the classifiers to correctly predict the phrase *forfærdeligt træls* as being negative. If we feed both of them with the misspelled phrase *forfærdligt træl*, the classifier with a word tokenizer wrongly predicts it to be neutral, whereas the character tokenizer correctly predicts it to be negative. Our assumption is that since the character tokenizer has already seen tokens such as *forfærd* from *forfærdeligt*, the tokens help the classifier understand the relation of the tokens to the words they originate from.

5.4.8 Custom multiclass SVM accuracy

When trying out our own multiclass SVM built using sklearns library, we achieve results that resembles the ones before very much. This one uses both the updated stop words and a character tokenizer. The differences lie in an average and maximum that both are slightly lower by 0,41% points and 0,31% points respectively, but a minimum that is higher by 1,31% points.

Accuracy	char_wb	
Minimum	Average	Maximum
63.93%	65.59%	69.59%

Table 16: Results with updated stop words.

What we've discovered by looking at the accuracy results of the cross validations from the two different classifiers that constitute this multiclass SVM (negative-or-rest & positive-or-rest), we see a general pattern.

- The positive-or-rest classifier generally scores between 87-88% accuracy.
- The negative-or-rest classifier generally scores between 64-68% accuracy.

As can be seen, differentiating between positives and non-positives is easy with our dataset, whereas it is slightly harder differentiating between negatives and non-negatives.

5.4.9 Refresher

In this section we described how we preprocess the data before we vectorize it. We described the different approaches we had to the SVM and why we use the linear kernel with n-grams of 1 to 10 with characters and word bounds.

5.5 Classification using a Neural Network

We implemented our neural network using the Keras[29] library. This allowed us to build and train a neural network quickly while not losing functionality.

One of the most important parts of our neural network is the LSTM (Long Short Term Memory) layer. This is where the tokens in the sentences are put in relation to one another in order to be able to predict the sentiment of the sentences. LSTM is a type of recurrent neural network[10], with the added feature that it learns which information to keep and what to forget. This essentially allows the network to look at multiple parts of the input at once. This makes LSTM networks very powerful when applied to domains like speech recognition, handwriting detection and natural language processing.

5.5.1 Preprocessing

Before a sentence is ready to be sent through the neural network, we need to do some preprocessing. This consists of the sentence being tokenized (converted to a sequence of tokens) and normalization of the token sequences. These steps are described in the following sections:

5.5.2 Data loading

Our dataset is loaded into the program the same way as everywhere else. However, since our neural network outputs predictions between 0 and 1, we need to make sure our labels are within this range. We do this by changing our negative labels to 0, our neutral labels to 0.5 and our positive labels to 1.

5.5.3 Tokenization

Since our neural network cannot read strings of text we need to find another way to represent our data. We do this by converting our sentences into sequences of integers using a tokenizer. This reads the sentence and maps every word to

[10]A type of neural network with connections to previous states. This allows the network to work on multiple parts of the same input data

a value if the tokenizer has seen the word when we train the neural network. If it finds a word it has not seen before, a token corresponding to the word "UNKNOWN" is added instead. The tokenizer also removes words that are outside the 10000 most used words in our dataset. This ensures that we ignore words that occur so rarely that they cannot be used to predict sentiment, and also reduces the training time as the number of words the network has to learn is reduced. The tokenizer used by our classifier contains mappings for the 10000 most used words in our dataset.

5.5.4 Sentence normalization

After the sentences have been tokenized, we need to normalize their lengths. We decided to use the length of the longest sentence in the dataset, since we wanted to avoid ignoring potentially important data. Normalizing the length of the sentences allows us to train the network using multiple sentences per iteration, rather than one at a time. This greatly reduces the training time of the network.

5.5.5 Network structure

Our network consists of the following three main layers. The embedding layer, which accepts our preprocessed sentences and converts them into feature vectors for later use in the network, an LSTM layer which does the predictions, and a dense layer, which completes the prediction, and returns the predicted sentiment of the sentence.

The network outputs a floating-point value between 0 and 1, where we assume 1 is positive, 0 is negative and 0.5 is neutral. Since the rest of our program uses the labels -1, 0 and 1, we need to convert the predictions to the correct labels. We do this by converting all predictions below 0.33 to -1, all above 0.66 to 1 and predictions between 0.33 and 0.66 to 0.

5.5.6 Training

We train our neural network using the labelled sentences in our dataset. After the sentences have been preprocessed, they are ready to be used for training the network. The network is trained over a number of iterations called epochs. For every epoch, the training data is split into smaller sets called batches. These batches are then sent through the network, which attempts to learn from data within them. When all the batches have been through the network, a new epoch begins and the batches are sent through again. After every epoch, a set of validation data is sent through the network and the accuracy of the predictions are evaluated. In order to prevent the network from overfitting on our training data, we allowed the network to stop training if there were no improvement in accuracy for the previous 5 epochs when predicting on the validation data. Additionally, to ensure the final network is as accurate as possible without overfitting, we only save the model with the best accuracy on our testing dataset.

When training the neural network, we reach an accuracy around 55% on our testing dataset. However, when using the network at any other time, we only hit an accuracy of 33%. We have not been able to find the exact reason, but we believe the reason for the low accuracy is most likely due to us finding it difficult to configure the neural network.

6 Analysis

Shown in table 17 is a collection of all the results from running our classifiers on the dataset. We also added the result of AFINN and his program in this table. For fairness, we ran his program in the exact same way as we ran our rule based classifier, so that the results could easily be compared.

The results of these classifiers have been gathered by training and running the classifier 100 times and taking the mean average of the accuracy results. This does not apply to the baseline-, the random-, AFINN's- and the rule based classifiers, as their results are completely static due to their nature of obtaining sentiment. Furthermore, as they use the complete dataset for obtaining accuracy, their results have been noted with a star. Otherwise, the classifier will have used 90% ($\sim$ 8100 phrases) of the entire dataset as training data, and 10% ($\sim$ 900 phrases) as testing data.

	Accuracy	F1 Score
Baseline	45.37%*	0.2081
Random	33.13%	0.3158
AFINN	52.89%*	0.4995
Lexical	56.27%*	0.5372
MML	63.01%	0.6345
SVM	67.12%	0.6689
Multi SVM	65.86%	0.6501
NN	35.84%	0.3000

Table 17: Table over results of all classifiers.

It's evident from this table that neither of our baseline classifiers (Baseline and Random) beat any of the other classifiers in accuracy. This proves that the other six classification approaches managed to beat the baseline classifiers.

As can be seen from the table, our lexical analysis did better than what AFINN had produced. This is partly because we have added more words which we've seen often used in our dataset, but most likely it's because we've based our lexicon upon AFINN's. When comparing our program to AFINN's when using AFINN's dataset, his program actually produced slightly higher accuracy, as seen in section 5.2 in table 6. Our classifier reached an accuracy of 52.76% whilst his reached an accuracy of 52.89%, which is in total an increase of .13% points. It's also possible to enable the support for emoticons when using AFINN to classify. Since we also have some emojis in our lexicon for our rule based classifier, it would only be fair to test his emoticon support too. Enabling this setting increased the accuracy by .04% points, which means that our lexical analysis approach still performs better.

An interesting result amongst these are the fact that the LSTM neural network did not manage to beat either of the naive approaches to sentiment analysis. Both AFINN and the lexical analysis managed to beat its accuracy, as well as our baseline classifier. As mentioned in section 5.5.6, we did have some difficulties getting the neural network to provide good and consistent results. The fact that it doesn't manage to beat either of the naive approaches does indicate that there are some major issues with how we handled the neural network, although we do not have a definitive reason to explain why we did not achieve higher accuracy.

When looking at our machine learning algorithms, all the variations did better than all the naive approaches including our primitive classifiers. The best performing of our SVMs was the regular SVM, and thereafter is the Multiclass SVM. Our MML implementation came in third, and our LSTM implementation last. The Multi SVM, although a nice idea, was not as good as our regular SVM in practice.

Alongside accuracy, we have also supplied the F1 scores of each classifier. F1 score is a weighted average of Precision and Recall, being the proportion of true positives on all positive predictions and the proportion of true positives on all actual positive elements. It works well in uneven class distributions, which is very helpful in our case of having few positives. Our F1 scores coincide with the accuracy fairly well, aside from our Baseline and our Random classifier. Again, as it was with accuracy, our SVM has the highest F1 score, followed by the Multi SVM, and on third place MML.

All in all it's clear to see that our SVM reigns supreme when it comes to accuracy and F1 score. Trailing slowly behind it is the Multi SVM, followed by the MML approach.

7 Discussion

7.1 Time constraints

In our project we were trying to explore as many valid options for classifiers as it was feasible possible. We ended up exploring more than 10 different algorithms to predict sentiment. While we did not achieve great results with most of them, this is not an indication of the algorithms not performing well to our task, as the reason could purely be how we allocated our time. Initially, we tested the base configurations of many of the different technologies and went with the ones we achieved the best results with — the SVM variations. As described in section 5.3 about MML, we quickly abandoned the framework, but as clearly seen in section 6, the framework ended up achieving the third best result only with use of the base configuration and the entirety of out dataset, and had we spend enough time on it, we possibly could have improved it further.

7.2 Negatives and Neutrals

Throughout the project it was quite clear to us that we had issues distinguishing negatives from neutrals and vice-versa. Firstly, if we purely tried to classify positives and not positives, we would achieve around 87% accuracy, which clearly indicates that we had no real issue with predicting the sentences with positive sentiment. While this can be because positive sentences in general are more unique, we can definitely say that the problem with inaccuracies was not due to classifying positives.

We have two ideas as to why we had issues with differentiating the classes: One of the reasons might be that even though we became more reliable in annotating the data after establishing guide lines, it became evident in our cross-validation of the data that most of the discrepancies were neutral/negatives. An example of this confusion becomes quite clear in the following sentence: "De holdt mig mange gange i fængsel". This sentence was annotated as -1, but the sentence is not inherently negative og positive, because it is only stating a fact. Therefore, it was changed to 0.

The other reason might be that negative and neutral sentences are inherently much a like, and in many cases what really differentiates them are the tone, which an algorithm will find very difficult to determine without a large quantity of data. An example of this is a sentence like: "hvorfor så ikke denne dag der skulle kun stemmes og babyen kan helt sikkert ikke røbe noget af resultaterne" where to a human it does not really make sense hence we feel it is neutral. We can't predict the sentiment and thus find it neutral, but the classifiers predict it to be -1. As also seen below in figure 11, our SVM classifier had mostly difficulties differentiating these two classes.

With more data, it might be possible for the SVM to better tell them apart, which brings us to the next section.

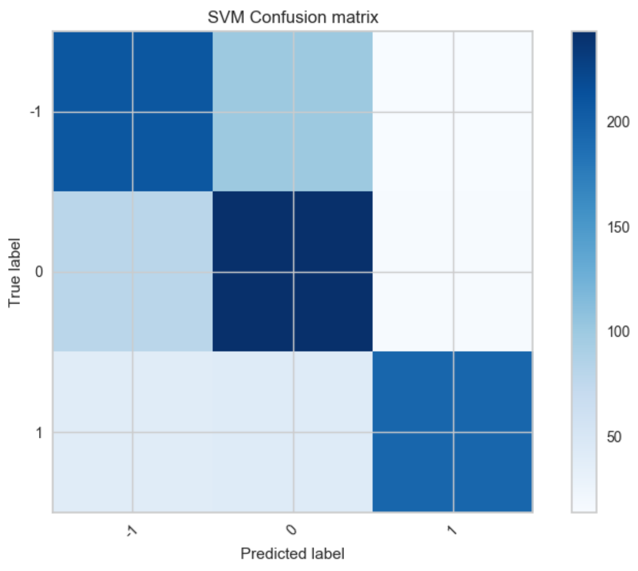
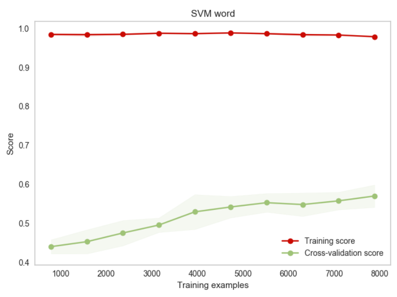


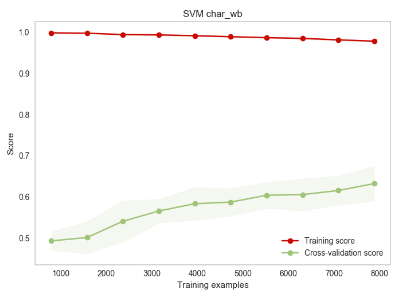
Figure 11: Confusion Matrix on our SVM.

7.3 Accuracy and Diversity

One of the reasons we believe that we are not achieving a higher accuracy is that we did not fulfil our goal of creating a diverse dataset; in fact, we unfortunately might have created a much too narrow scope. We believe this to be a flaw in our methodology in which we did not focus on diversifying the articles we collected data from. Instead, we were focused on finding articles with many comments in the belief that there would be a lot more data to collect. A prime example of this is that one of our group members collected 2500 sentences from only 18 articles. This means that 27.75% of our dataset originates from 7.53% of the different articles, which we believe gives us too narrow of a scope. Another reason we assume to be a causation for a lower accuracy can be seen from the following graphs.



(a) Learning curve with word.



(b) Learning curve with char_wb.

Figure 12: Graphs of both learning curves for our SVM.

The graphs in figure 12a and 12b shows how the average accuracy of our cross validation iterations increase with the amount of data samples we give it. As sklearns user guide to learning curves states [30]: *If the training score is much greater than the validation score for the maximum number of training samples, adding more training samples will most likely increase generalization.*. Since our training score is largely stable and our cross validation score is increasing, it could imply that we do not have enough data, and with that, not enough features.

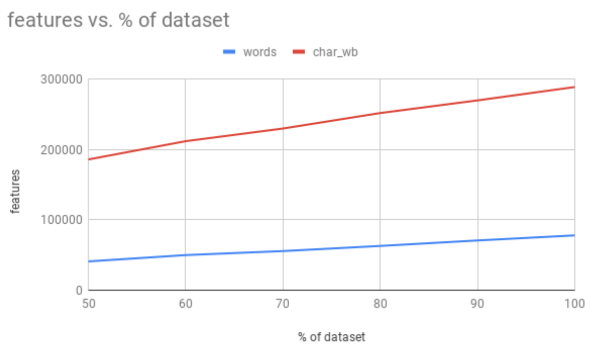


Figure 13: Feature space showcase of our SVM

As seen in figure 13, we can see that both when tokenizing words and characters, our feature space continues to increase linearly with the amount of training samples we have. The problem with increasing the feature space is the *curse of dimensionality*. The curse of dimensionality says that the more features or dimensions we have, the more data we need to generalize accurately grows exponentially. That is, the more dimensions we have, the longer it will take to train, and the more memory it will use. An important note in the difference between the feature spaces are also that the char_wb uses 1-10 n-grams and the words only use 1 to 3 n-grams.

7.3.1 Testing our hypothesis

In summary, we have some indications that our dataset is not entirely representative for comments on Danish political articles on social media. To disprove this hypothesis we performed an experiment where we collected 100 completely random sentences, unseen and from a very broad spectrum of media/articles[11]. We then trained our SVM 100 times where we randomly selected 90% as training data, and tested with the 100 test sentences. We then trained our SVM with the entire dataset and tested it with our 100 sentences. Firstly, we did the version with 90% because all the other versions of our SVM, MML and NN had 10% of the dataset as test data. Secondly, we did a 100% version to test how our SVM would perform in a real world scenario, but it would also be able to provide us with indications of whether our hypothesis about the lack of data was correct or not.

[11]Link to these sentences: https://github.com/lucaspuvis/SAM/blob/master/SAM/Data/100_wild_sentences.csv

As seen in figure 18, our accuracy mean clearly falls with 3% point in the 90% version when we use sentences that were touching political topics we had not previously seen before. This is an indication of the hypothesis about a flaw in the methodology being correct and thus we cannot falsify this hypothesis from the experiment. This is easily solved by increasing the amount of data and making sure in the future that the data is more diverse. What we can see with the 100% version is that the accuracy increases on the 100 samples, which again is an indication of the hypothesis regarding the lack of data has merit. Additionally, as seen in 7.3, increasing the amount of data is still indirectly increasing the diversity of the dataset.

	Accuracy	F1 Score
90%	63.7255%	0.6340
100%	62.4709%	0.6218

Table 18: Results of 100 unique sentences

8 Conclusion

In summary, we have explained our methodology and introduced guide lines on how to annotate sentences. We have collected and annotated over 9000 sentences from the comment section of political articles on Danish social media. We have introduced several implementations that predicts sentiment and the most promising being our SVM. A discovery was that Facebook largely consisted of negative sentences, and that it seems negative and neutral sentences are very much alike and therefore harder to distinguish. We have also revealed a minor flaw in our methodology that can be countered with an increasingly larger and diverse dataset, which we also have discovered would make us able to achieve a higher accuracy. Looking forward, we believe we should increase our dataset and continuously try different variations of the SVM that we did not get to introduce in this paper.

References

[1] Oxford Dictionary. (May 7, 2019). Echo chamber, [Online]. Available: https://en.oxforddictionaries.com/definition/echo_chamber.

[2] Reddit. (May 14, 2019). Denmark, [Online]. Available: https://old.reddit.com/r/Denmark/.

[3] Reddit, inc. (May 14, 2019). Reddit content policy, [Online]. Available: https://www.redditinc.com/policies/content-policy.

[4] Facebook. (May 7, 2019). Tv2 nyhederne, [Online]. Available: https://www.facebook.com/tv2nyhederne/.

[5] ——, (May 7, 2019). B.t., [Online]. Available: https://www.facebook.com/ditbt/.

[6] ——, (May 7, 2019). Dr nyheder, [Online]. Available: https://www.facebook.com/DRNyheder/.

[7] ——, (May 7, 2019). Politiken, [Online]. Available: https://www.facebook.com/politiken/.

[8] ——, (May 7, 2019). Dagbladet information, [Online]. Available: https://www.facebook.com/dagbladetinformation/.

[9] ——, (May 7, 2019). Fucking flink, [Online]. Available: https://www.facebook.com/FFflink/.

[10] K. Krippendorff, "Computing krippendorff's alpha-reliability", 2011. [Online]. Available: http://repository.upenn.edu/asc_papers/43.

[11] Deen Frelon. (May 7, 2019). Recal, [Online]. Available: http://dfreelon.org/recal/recal-oir.php.

[12] S. M. Mohammad, "A practical guide to sentiment annotation: Challenges and solutions", 2016. [Online]. Available: https://www.aclweb.org/anthology/W16-0429?fbclid=IwAR1m54yZMwQ1iOEpwOgMsKWqCR-SYQWOZ2MpwRtuatY1XSs9VVqoLOIv6yc.

[13] Marketing Charts. (May 6, 2019). Bad customer service interactions more likely to be shared than good ones, [Online]. Available: https://www.marketingcharts.com/digital-28628.

[14] Haubergs. (May 7, 2019). Lix calculator, [Online]. Available: http://haubergs.com/rix.

[15] sproget.dk. (May 14, 2019). De mest almindelige ord i dansk, [Online]. Available: https://sproget.dk/temaer/ord-og-bogstaver/hvad-er-de-mest-almindelige-ord-pa-dansk.

[16] Stack Overflow. (May 6, 2019). Developer survey results, [Online]. Available: https://insights.stackoverflow.com/survey/2018.

[17] R. C. Martin. (May 6, 2019). The principles of ood, [Online]. Available: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

[18] University of Florida. (May 7, 2019). Pipelineprocessing, [Online]. Available: https://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/AlgorithmStructure/Pipeline.htm.

[19] OODesign. (May 7, 2019). Adapter pattern, [Online]. Available: `https://www.oodesign.com/adapter-pattern.html`.

[20] F. Å. Nielsen, "A new anew: Evaluation of a word list for sentiment analysis in microblogs", 2011. [Online]. Available: `http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/6006/pdf/imm6006.pdf`.

[21] Center for Sprogteknologi. (May 7, 2019). Dannet, [Online]. Available: `https://cst.ku.dk/projekter/dannet/`.

[22] (May 13, 2019). Ml.net v0.11, [Online]. Available: `https://github.com/dotnet/machinelearning/releases/tag/v0.11.0`.

[23] scikit learn. (May 6, 2019). Tfidftransformer, [Online]. Available: `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html#sklearn.feature_extraction.text.TfidfTransformer`.

[24] Scikit-learn developers. (May 13, 2019). Sklearn rbf kernel documentation, [Online]. Available: `https://scikit-learn.org/stable/modules/metrics.html#rbf-kernel`.

[25] ——, (May 13, 2019). Sklearn polynomial kernel documentation, [Online]. Available: `https://scikit-learn.org/stable/modules/metrics.html#polynomial-kernel`.

[26] ——, (May 13, 2019). Sklearn polynomial kernel documentation, [Online]. Available: `https://scikit-learn.org/stable/modules/metrics.html#linear-kernel`.

[27] scikit-learn developers. (May 7, 2019). Stratified kfold explanation, [Online]. Available: `https://scikit-learn.org/stable/auto_examples/model_selection/plot_cv_indices.html`.

[28] (May 13, 2019). Sklearn rbf documentation, [Online]. Available: `https://www.scikit-yb.org/en/latest/api/text/tsne.html`.

[29] Chollet, François and others. (May 7, 2019). Keras, [Online]. Available: `https://keras.io/`.

[30] scikit-learn developers. (May 7, 2019). Sklearn learning curves, [Online]. Available: `https://scikit-learn.org/stable/modules/learning_curve.html#learning-curve`.

List of Figures

1 Distribution of sentences on social media 6
2 Distribution of our classes. 6
3 Comparison of sentiment across media. 7
4 UML Diagram of how the pipe works 11
5 Comparison of Abbreviation Expansion 12
6 Comparison of runtimes of the program. 14
7 Comparison of runtimes of the server. 14
8 Image showcasing Stratified K-Fold.[27] 24
9 Projection of our data. 26
10 Confusion matrix of our classifier. 27
11 Confusion Matrix on our SVM. 34
12 Graphs of both learning curves for our SVM. 34
13 Feature space showcase of our SVM 35
14 Results from running the Baseline Classifier 52
15 Results from running the Random Classifier 52
16 Results running Afinn without emoticons 53
17 Results running Afinn with emoticons 53
18 Results from running Rule Based with Afinns lexicon 54
19 Results from running Rule Based with our lexicon 54
20 Final results from running the Rule Based classifier. 55

List of Tables

1 Danish word frequency lists. 8
2 Parallelism Overview 13
3 Results of the Baseline Classifier. 15
4 Results of the Random Classifier. 15
5 Tokenization of a sentence. 16
6 Results using AFINN's lexicon 16
7 Results using our new lexicon 17
8 Negation example with sentiment. 18
9 Multiplier example with sentiment. 18
10 Phrase example with sentiment. 19
11 Results using Modifiers 20
12 Results using MML classifier. 21
13 Results with word tokenizing. 27
14 Results using character tokenization. 27
15 Results with updated stop words. 28
16 Results with updated stop words. 28
17 Table over results of all classifiers. 31
18 Results of 100 unique sentences 36
23 Table of Confusion for Negatives 55
24 Table of Confusion for Neutrals 55
25 Table of Confusion for Positives 56
26 Table of Confusion for Negatives 56
27 Table of Confusion for Neutrals 56
28 Table of Confusion for Positives 56

29 Table of Confusion for Negatives 57
30 Table of Confusion for Neutrals 57
31 Table of Confusion for Positives 57
32 Table of Confusion for Negatives 58
33 Table of Confusion for Neutrals 58
34 Table of Confusion for Positives 58

A Appendixes

A.1 Link to repositories

Link to public repository containing the data: https://github.com/steffan267/Sentiment-Analysis-on-Danish-Social-Media
Link to public repository containing the code: https://github.com/lucaspuvis/SAM

A.2 MML 100 runs

Accuracy	F Score
63.3262	0.6378
62.2601	0.6264
62.4733	0.6301
64.0725	0.6475
62.58	0.6289
62.8998	0.6348
63.0064	0.6334
63.2196	0.6313
63.0064	0.6387
63.2196	0.6343
62.7932	0.6262
62.58	0.6298
63.4328	0.6394
62.0469	0.6256
62.1535	0.6232
63.3262	0.64
63.7527	0.6432
62.6866	0.6312
62.6866	0.6324
63.5394	0.6375
64.0725	0.6483
63.2196	0.6364
62.7932	0.6285
63.113	0.6359
64.0725	0.6501
63.3262	0.6417
62.7932	0.6328
63.0064	0.635
62.7932	0.6346
62.7932	0.6341
62.7932	0.6338
62.6866	0.6313
62.0469	0.6218
63.0064	0.6369
62.3667	0.6242
64.0725	0.649
62.58	0.6284
63.8593	0.6484

63.0064	0.6348
63.0064	0.6361
62.8998	0.6324
63.2196	0.6389
63.3262	0.639
63.0064	0.6344
63.5394	0.641
63.4328	0.6394
62.1535	0.6232
63.3262	0.6374
61.7271	0.6167
62.6866	0.6315
63.2196	0.6391
63.9659	0.6467
62.7932	0.6309
63.113	0.6356
62.8998	0.6356
62.2601	0.6235
64.2857	0.6499
62.1535	0.6245
63.0064	0.6361
63.3262	0.6373
62.7932	0.6324
63.6461	0.6425
62.58	0.6263
63.2196	0.6384
63.113	0.6355
63.113	0.6375
63.4328	0.6406
63.113	0.6273
63.8593	0.6481
62.8998	0.6338
62.8998	0.6376
63.0064	0.6349
63.2196	0.6388
62.7932	0.6318
63.5394	0.6388
63.3262	0.638
62.4733	0.6271
63.2196	0.6349
62.3667	0.6267
63.2196	0.6365
62.7932	0.6328
62.3667	0.6252
62.6866	0.6283
62.6866	0.6314
63.7527	0.6414
62.6866	0.6298
63.4328	0.6405
63.2196	0.6338

62.3667	0.6245
62.2601	0.6247
63.2196	0.6388
63.5394	0.6404
62.1535	0.6233
63.3262	0.6375
63.3262	0.6388
61.4073	0.616
62.7932	0.6288
63.7527	0.643
63.3262	0.6376
63.5394	0.6425

A.3 SVM 100 runs

Accuracy	F score
0.6614872364	0.6607561635
0.7014428413	0.6989490006
0.6725860155	0.6707005234
0.6803551609	0.6787190021
0.6559378468	0.6539430024
0.6759156493	0.6747068014
0.6736958935	0.6689893172
0.6847946726	0.682894799
0.6692563818	0.6675820089
0.6437291898	0.6404474573
0.6681465039	0.66634559
0.6648168701	0.6624841994
0.6736958935	0.6708605108
0.6770255272	0.6766812327
0.6736958935	0.6722842943
0.6970033296	0.6950535414
0.6725860155	0.6699560831
0.6581576027	0.6563592618
0.6947835738	0.6929331127
0.6559378468	0.6542231833
0.6526082131	0.65179038
0.6459489456	0.6421890391
0.6914539401	0.6885237951
0.6526082131	0.6512851487
0.6359600444	0.6354335417
0.6836847947	0.6830044185
0.6736958935	0.6722200841
0.71809101	0.7145994375
0.6614872364	0.6588441067
0.6692563818	0.6650176106
0.6725860155	0.6714628206
0.679245283	0.6774510322

0.6836847947	0.6808324088
0.6814650388	0.6785443161
0.6770255272	0.6750754131
0.6859045505	0.6817348951
0.6814650388	0.6799306984
0.6548279689	0.6514893047
0.6625971143	0.6618256116
0.6770255272	0.6740927585
0.6825749168	0.6798909686
0.6681465039	0.6644050903
0.6892341842	0.6872227449
0.6870144284	0.6843296218
0.6570477248	0.655129305
0.6503884573	0.6467043591
0.6725860155	0.6702516239
0.653718091	0.651854435
0.6759156493	0.6756731465
0.6625971143	0.6605535518
0.6936736959	0.6932369996
0.6759156493	0.6732919871
0.667036626	0.6662660437
0.653718091	0.6510576175
0.6625971143	0.6593601624
0.6648168701	0.6602964588
0.6725860155	0.6700104873
0.6759156493	0.6749567408
0.6637069922	0.6615740731
0.6692563818	0.6662562323
0.6570477248	0.6518892789
0.6526082131	0.6512022654
0.6692563818	0.664564707
0.6703662597	0.6665316435
0.6825749168	0.6810277207
0.679245283	0.673462919
0.6759156493	0.6726995632
0.6370699223	0.632652987
0.667036626	0.6643564219
0.6759156493	0.675539004
0.6725860155	0.6694089259
0.6648168701	0.6641824816
0.692563818	0.689007986
0.6714761376	0.6698467746
0.6825749168	0.6814571806
0.6759156493	0.674894931
0.667036626	0.6635209361
0.6614872364	0.6596798755
0.6581576027	0.655618738
0.6736958935	0.6692205861
0.6770255272	0.6757349091
0.6559378468	0.6542174273

0.6681465039	0.6646113195
0.6803551609	0.6796644528
0.6814650388	0.6790306517
0.6703662597	0.6688617232
0.6681465039	0.6658305454
0.6703662597	0.6675377292
0.679245283	0.6774183974
0.6625971143	0.6614678715
0.6603773585	0.658890624
0.6714761376	0.670656634
0.6870144284	0.6849414402
0.6659267481	0.6622733707
0.6914539401	0.6900582734
0.6725860155	0.6713604928
0.6714761376	0.6676406968
0.6770255272	0.6759003822
0.6570477248	0.656885219
0.667036626	0.6665822384

A.4 Multi-SVM 100 runs

Accuracy	F Score
63.70699223	0.6748860769
65.70477248	0.659496311
65.92674806	0.6482979363
67.81354051	0.6629957131
64.48390677	0.6576925097
68.70144284	0.6518792252
67.48057714	0.6393606451
66.59267481	0.6518214775
66.81465039	0.675120665
66.59267481	0.655870916
66.03773585	0.6586077362
64.92785794	0.6666113009
64.92785794	0.6476503919
67.59156493	0.6598987734
68.25749168	0.6586614275
65.70477248	0.6080750537
66.37069922	0.6669069326
66.03773585	0.652520744
65.59378468	0.6538546222
64.37291898	0.6143272694
67.70255272	0.6507987854
68.47946726	0.6605617589
64.92785794	0.6550158249
67.81354051	0.6526536353
63.70699223	0.6287721264
65.81576027	0.6625476953

66.92563818	0.6497782061
66.14872364	0.6666437093
68.92341842	0.6413255519
65.14983352	0.6485723173
63.04106548	0.6432161573
67.03662597	0.6370196875
67.25860155	0.6450969976
66.92563818	0.6470759105
65.59378468	0.6723792713
63.59600444	0.6278675341
66.7036626	0.6486921022
69.36736959	0.6292793483
65.70477248	0.6372193682
66.7036626	0.6519493957
64.0399556	0.6607681597
65.26082131	0.6689500955
65.26082131	0.6501356946
65.14983352	0.6703771512
63.15205327	0.6583119956
62.70810211	0.6463559317
63.81798002	0.5980634315
63.92896781	0.6427964749
64.70588235	0.6609634077
66.48168701	0.6363848885
66.03773585	0.6503859935
65.26082131	0.6456190176
64.92785794	0.6559244687
66.03773585	0.6555477834
63.70699223	0.6525027235
66.37069922	0.6403619787
67.03662597	0.6483438445
66.25971143	0.6547495643
68.92341842	0.6133648944
64.70588235	0.6586917501
68.25749168	0.6634015292
64.0399556	0.656917385
67.70255272	0.6559252885
66.48168701	0.5947780706
64.1509434	0.6215324449
65.14983352	0.663205715
68.70144284	0.6734474606
63.59600444	0.6678062141
68.81243063	0.655346803
67.03662597	0.6485897413
65.48279689	0.6725344287
66.25971143	0.6794749107
65.70477248	0.6715653206
62.93007769	0.6461675436
65.03884573	0.6270675944
65.70477248	0.6575493132

65.14983352	0.6450570801
65.92674806	0.6517417542
67.14761376	0.6303446509
67.48057714	0.6171391303
66.03773585	0.6555305358
66.14872364	0.643980532
62.8190899	0.6516650578
65.92674806	0.6664912566
63.92896781	0.6613817071
64.81687014	0.6541969147
65.48279689	0.6444341032
64.92785794	0.6433560397
69.2563818	0.630832549
65.92674806	0.6342884039
66.25971143	0.6846959293
65.59378468	0.6529909674
63.04106548	0.6646972375
66.48168701	0.6279043637
65.70477248	0.6588165248
67.36958935	0.6323385635
65.70477248	0.6330608703
65.3718091	0.6350909971
65.48279689	0.68090234
63.81798002	0.6651437154

A.5 Neural Network 100 run statistics

100 runs	
avg acc:	35.84%
max acc:	37.74%
min acc:	34.65%
avg f:	0.3
max f:	0.37
min f:	0.26

A.6 SVM 100 sentences test 100 runs

Accuracy	F1 Score
0.637254902	0.6325355776
0.6176470588	0.6136892071
0.6078431373	0.6030758644
0.6274509804	0.6259528687
0.6078431373	0.6087395696
0.6078431373	0.6022441111
0.6666666667	0.6636857952
0.6274509804	0.6297114366
0.6470588235	0.6463856647

0.6274509804	0.6252756599
0.637254902	0.6333333333
0.6274509804	0.6287100793
0.5980392157	0.5889931332
0.6470588235	0.6453824497
0.6176470588	0.6140358104
0.5980392157	0.59533571
0.637254902	0.633802378
0.6470588235	0.643086879
0.6274509804	0.6261249396
0.637254902	0.6302827142
0.6568627451	0.6540866719
0.637254902	0.6334467691
0.6078431373	0.6060705922
0.637254902	0.6342997557
0.6176470588	0.6128342246
0.6176470588	0.6137254902
0.637254902	0.6345648214
0.6274509804	0.6262718763
0.637254902	0.6339342865
0.6176470588	0.6164650003
0.6176470588	0.6121845261
0.6176470588	0.6173136269
0.6666666667	0.66213126
0.5980392157	0.5964222885
0.6568627451	0.6490118164
0.6470588235	0.6433749257
0.6176470588	0.615718896
0.6470588235	0.6444807275
0.637254902	0.6335316273
0.6274509804	0.6247429918
0.637254902	0.6338978015
0.637254902	0.633802378
0.6274509804	0.6252756599
0.637254902	0.635762693
0.6274509804	0.6288876866
0.6274509804	0.6252756599
0.6176470588	0.6107287923
0.6078431373	0.6034547152
0.6470588235	0.644529463
0.637254902	0.6325355776
0.637254902	0.6367827399
0.637254902	0.6382759534
0.6078431373	0.6060705922
0.6078431373	0.6021515231
0.6176470588	0.6131721351
0.5980392157	0.5968519498
0.6176470588	0.6167260844
0.6176470588	0.6145384868
0.6470588235	0.6456973955

0.6274509804	0.6261249396
0.6176470588	0.6128342246
0.6764705882	0.6753094753
0.6176470588	0.6193144947
0.6176470588	0.6084669698
0.6274509804	0.6238562092
0.6176470588	0.6118982028
0.5980392157	0.5974920881
0.6078431373	0.6069896371
0.6274509804	0.6244509451
0.5882352941	0.5827880807
0.637254902	0.6367479113
0.6176470588	0.6123461405
0.6274509804	0.6250876972
0.6176470588	0.6176668647
0.5882352941	0.5832966796
0.6274509804	0.6244206774
0.6176470588	0.6128342246
0.6078431373	0.6045278791
0.6274509804	0.6231367767
0.5980392157	0.5929451455
0.637254902	0.635762693
0.637254902	0.6348324238
0.6470588235	0.645260242
0.637254902	0.6308779992
0.637254902	0.6353029708
0.637254902	0.6336880782
0.6176470588	0.6156273561
0.5882352941	0.5846405229
0.637254902	0.6328942748
0.6274509804	0.6244206774
0.5882352941	0.5864725725
0.6274509804	0.6231367767
0.6078431373	0.6028627646
0.5980392157	0.5988503965
0.5980392157	0.5980043759
0.6176470588	0.6146167558
0.6274509804	0.6238562092
0.6274509804	0.6235333594
0.6176470588	0.6146167558
0.5980392157	0.5949814975

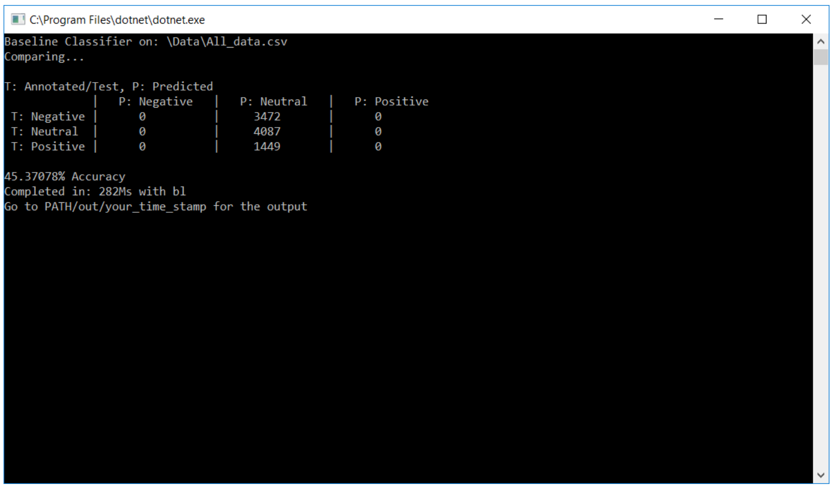
A.7 Program to test AFINN

```python
import csv
from afinn import Afinn

afinn = Afinn(language='da')

with open("all_data.csv", 'r', encoding="UTF8") as csvfile:
        csvreader = csv.reader(csvfile)
        afinn = Afinn(language="da")
        List = []
        positive_guessed_positive = 0
        positive_guessed_neutral = 0
        positive_guessed_negative = 0
        neutral_guessed_positive = 0
        neutral_guessed_neutral = 0
        neutral_guessed_negative = 0
        negative_guessed_positive = 0
        negative_guessed_neutral = 0
        negative_guessed_negative = 0
        count = 0
        linecount = 0
        for row in csvreader:
                score = afinn.score(row[1])
                List.append((row[0], score))
                linecount = linecount + 1
                for pair in List:
                        rating = int(pair[0])
                        if rating == 0:
                                if pair[1] == 0:
                                        count = count + 1
                                        neutral_guessed_neutral =
                                        ↪  neutral_guessed_neutral + 1
                                if pair[1] > 0:
                                        neutral_guessed_positive =
                                        ↪  neutral_guessed_positive + 1
                                if pair[1] < 0:
                                        neutral_guessed_negative =
                                        ↪  neutral_guessed_negative + 1
                        if rating > 0:
                                if pair[1] > 0:
                                        count = count + 1
                                        positive_guessed_positive =
                                        ↪  positive_guessed_positive + 1
                                if pair[1] < 0:
                                        positive_guessed_negative =
                                        ↪  positive_guessed_negative + 1
                                if pair[1] == 0:
                                        positive_guessed_neutral =
                                        ↪  positive_guessed_neutral + 1
                        if rating < 0:
                                if pair[1] < 0:
                                        count = count + 1
                                        negative_guessed_negative =
                                        ↪  negative_guessed_negative + 1
                                if pair[1] > 0:
                                        negative_guessed_positive =
                                        ↪  negative_guessed_positive + 1
                                if pair[1] == 0:
                                        negative_guessed_neutral =
                                        ↪  negative_guessed_neutral + 1
print((count/linecount)*100)
print("Negative guessed Negative: " + str(negative_guessed_negative))
print("Negative guessed Neutral: " + str(negative_guessed_neutral))
print("Negative guessed Positive: " + str(negative_guessed_positive))
print("Neutral guessed Negative: " + str(neutral_guessed_negative))
print("Neutral guessed Neutral: " + str(neutral_guessed_neutral))
print("Neutral guessed Positive: " + str(neutral_guessed_positive))
print("Positive guessed Negative: " + str(positive_guessed_negative))
print("Positive guessed Neutral: " + str(positive_guessed_neutral))
print("Positive guessed Positive: " + str(positive_guessed_positive))
```

A.8 Baseline Classifier test results



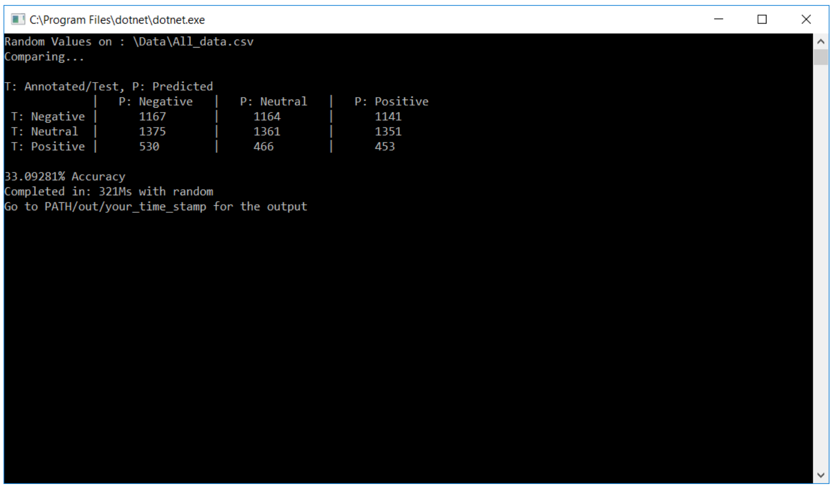
```
Baseline Classifier on: \Data\All_data.csv
Comparing...

T: Annotated/Test, P: Predicted
             |   P: Negative   |   P: Neutral   |   P: Positive
 T: Negative |      0          |     3472       |      0
 T: Neutral  |      0          |     4087       |      0
 T: Positive |      0          |     1449       |      0

45.37078% Accuracy
Completed in: 282Ms with bl
Go to PATH/out/your_time_stamp for the output
```

Figure 14: Results from running the Baseline Classifier

A.9 Random Classifier test results



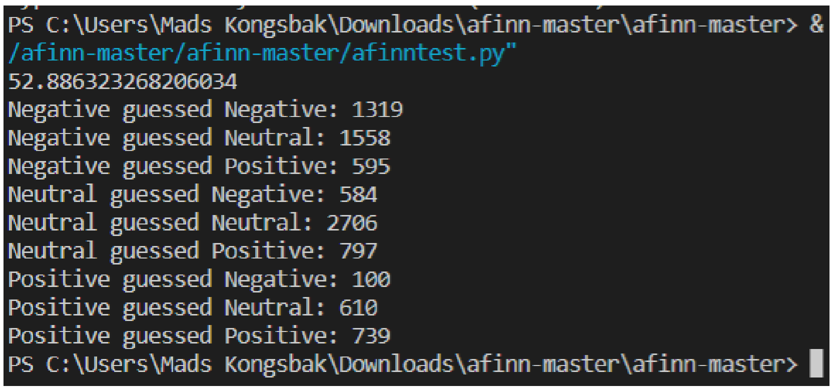
```
Random Values on : \Data\All_data.csv
Comparing...

T: Annotated/Test, P: Predicted
             |   P: Negative   |   P: Neutral   |   P: Positive
 T: Negative |      1167       |     1164       |      1141
 T: Neutral  |      1375       |     1361       |      1351
 T: Positive |      530        |     466        |      453

33.09281% Accuracy
Completed in: 321Ms with random
Go to PATH/out/your_time_stamp for the output
```

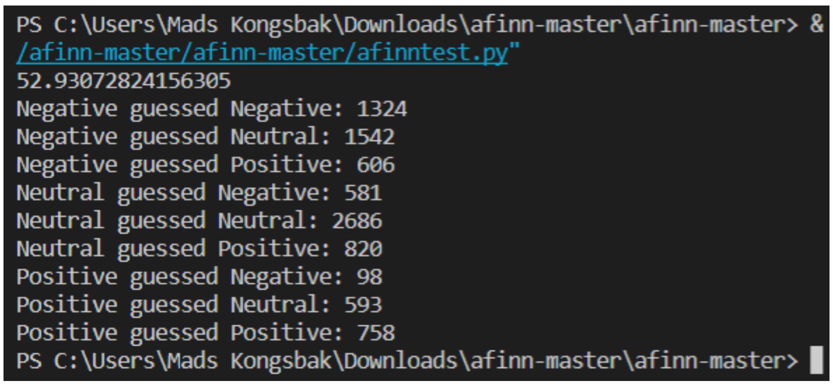
Figure 15: Results from running the Random Classifier

A.10 AFINN test results



```
PS C:\Users\Mads Kongsbak\Downloads\afinn-master\afinn-master> &
/afinn-master/afinn-master/afinntest.py"
52.886323268206034
Negative guessed Negative: 1319
Negative guessed Neutral: 1558
Negative guessed Positive: 595
Neutral guessed Negative: 584
Neutral guessed Neutral: 2706
Neutral guessed Positive: 797
Positive guessed Negative: 100
Positive guessed Neutral: 610
Positive guessed Positive: 739
PS C:\Users\Mads Kongsbak\Downloads\afinn-master\afinn-master>
```

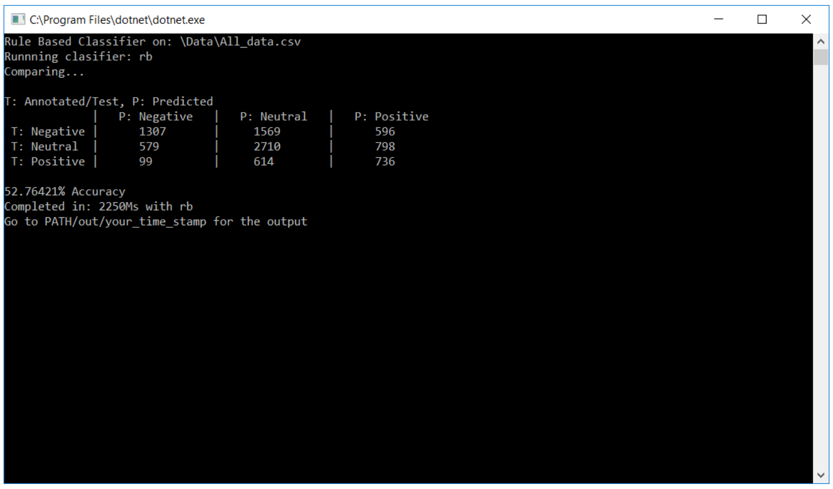
Figure 16: Results running Afinn without emoticons



```
PS C:\Users\Mads Kongsbak\Downloads\afinn-master\afinn-master> &
/afinn-master/afinn-master/afinntest.py"
52.93072824156305
Negative guessed Negative: 1324
Negative guessed Neutral: 1542
Negative guessed Positive: 606
Neutral guessed Negative: 581
Neutral guessed Neutral: 2686
Neutral guessed Positive: 820
Positive guessed Negative: 98
Positive guessed Neutral: 593
Positive guessed Positive: 758
PS C:\Users\Mads Kongsbak\Downloads\afinn-master\afinn-master>
```

Figure 17: Results running Afinn with emoticons

A.11 Lexical Analysis Results

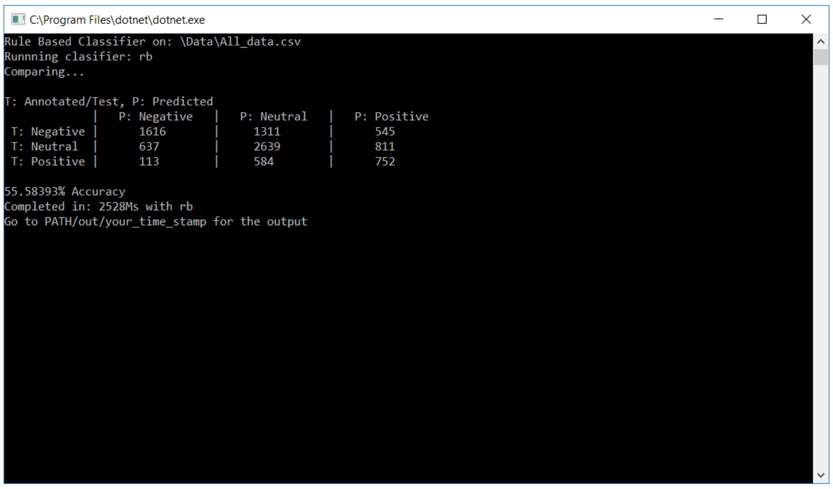


```
C:\Program Files\dotnet\dotnet.exe
Rule Based Classifier on: \Data\All_data.csv
Runnning clasifier: rb
Comparing...

T: Annotated/Test, P: Predicted
             |   P: Negative   |   P: Neutral   |   P: Positive
 T: Negative |      1307       |     1569       |      596
 T: Neutral  |      579        |     2710       |      798
 T: Positive |      99         |     614        |      736

52.76421% Accuracy
Completed in: 2250Ms with rb
Go to PATH/out/your_time_stamp for the output
```

Figure 18: Results from running Rule Based with Afinns lexicon

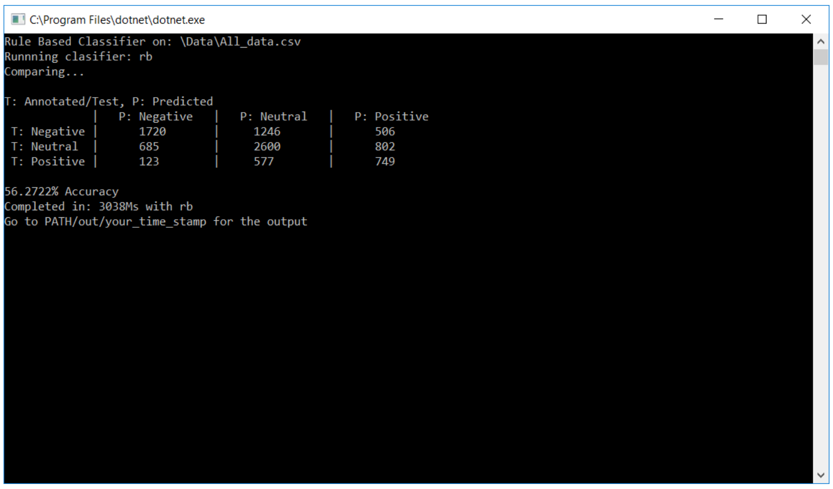


```
C:\Program Files\dotnet\dotnet.exe
Rule Based Classifier on: \Data\All_data.csv
Runnning clasifier: rb
Comparing...

T: Annotated/Test, P: Predicted
             |   P: Negative   |   P: Neutral   |   P: Positive
 T: Negative |      1616       |     1311       |      545
 T: Neutral  |      637        |     2639       |      811
 T: Positive |      113        |     584        |      752

55.58393% Accuracy
Completed in: 2528Ms with rb
Go to PATH/out/your_time_stamp for the output
```

Figure 19: Results from running Rule Based with our lexicon



```
C:\Program Files\dotnet\dotnet.exe
Rule Based Classifier on: \Data\All_data.csv
Runnning clasifier: rb
Comparing...

T: Annotated/Test, P: Predicted
             |   P: Negative   |   P: Neutral   |   P: Positive
 T: Negative |      1720       |     1246       |      506
 T: Neutral  |      685        |     2600       |      802
 T: Positive |      123        |     577        |      749

56.2722% Accuracy
Completed in: 3038Ms with rb
Go to PATH/out/your_time_stamp for the output
```

Figure 20: Final results from running the Rule Based classifier.

A.12 Baseline Classifier F1 Score Calculation

	Predicted	
Actual	Negative	Other
Negative	0 True Positives	3472 False Negatives
Other	0 False Positives	5536 True Negatives

Table 23: Table of Confusion for Negatives

$$F1 = \frac{2 \cdot 0}{(2 \cdot 0) + 3472 + 0} = 0$$

	Predicted	
Actual	Neutral	Other
Neutral	4087 True Positives	0 False Negatives
Other	4927 False Positives	0 True Negatives

Table 24: Table of Confusion for Neutrals

$$F1 = \frac{2 \cdot 4087}{(2 \cdot 4087) + 4927 + 0} = 0.62420771286$$

	Predicted	
Actual	Positive	Other
Positive	0 True Positives	1449 False Negatives
Other	0 False Positives	7559 True Negatives

Table 25: Table of Confusion for Positives

$$F1 = \frac{2 \cdot 0}{(2 \cdot 0) + 1449 + 0} = 0$$

$$F1Average = \frac{0.62420771286 + 0 + 0}{3} = 0.20806923762$$

A.13 Random Classifier F1 Score Calculation

	Predicted	
Actual	Negative	Other
Negative	1167 True Positives	2305 False Negatives
Other	1905 False Positives	3631 True Negatives

Table 26: Table of Confusion for Negatives

$$F1 = \frac{2 \cdot 1167}{(2 \cdot 1167) + 2305 + 1905} = 0.35666259168$$

	Predicted	
Actual	Neutral	Other
Neutral	1361 True Positives	2726 False Negatives
Other	1630 False Positives	3291 True Negatives

Table 27: Table of Confusion for Neutrals

$$F1 = \frac{2 \cdot 1361}{(2 \cdot 1361) + 1630 + 2726} = 0.38457191297$$

	Predicted	
Actual	Positive	Other
Positive	453 True Positives	2492 False Negatives
Other	996 False Positives	5067 True Negatives

Table 28: Table of Confusion for Positives

$$F1 = \frac{2 \cdot 453}{(2 \cdot 453) + 996 + 2492} = 0.20619025944$$

$$F1Average = \frac{0.35666259168 + 0.38457191297 + 0.20619025944}{3} = 0.31580825469$$

A.14 AFINN Classifier F1 Score Calculation

	Predicted	
Actual	Negative	Other
Negative	1319 True Positives	2153 False Negatives
Other	684 False Positives	4852 True Negatives

Table 29: Table of Confusion for Negatives

$$F1 = \frac{2 \cdot 1319}{(2 \cdot 1319) + 684 + 2153} = 0.48182648401$$

	Predicted	
Actual	Neutral	Other
Neutral	2706 True Positives	1381 False Negatives
Other	2168 False Positives	2753 True Negatives

Table 30: Table of Confusion for Neutrals

$$F1 = \frac{2 \cdot 2706}{(2 \cdot 2706) + 2168 + 1381} = 0.60395045195$$

	Predicted	
Actual	Positive	Other
Positive	739 True Positives	710 False Negatives
Other	1392 False Positives	6167 True Negatives

Table 31: Table of Confusion for Positives

$$F1 = \frac{2 \cdot 739}{(2 \cdot 739) + 710 + 1392} = 0.41284916201$$

$$F1Average = \frac{0.48182648401 + 0.60395045195 + 0.41284916201}{3} = 0.49954203265$$

A.15 Lexical Analysis F1 Score Calculation

	Predicted	
Actual	Negative	Other
Negative	1720 True Positives	1752 False Negatives
Other	808 False Positives	4728 True Negatives

Table 32: Table of Confusion for Negatives

$$F1 = \frac{2 \cdot 1720}{(2 \cdot 1720) + 808 + 1752} = 0.57333333333$$

	Predicted	
Actual	Neutral	Other
Neutral	2600 True Positives	1487 False Negatives
Other	1823 False Positives	3098 True Negatives

Table 33: Table of Confusion for Neutrals

$$F1 = \frac{2 \cdot 2600}{(2 \cdot 2600) + 1823 + 1487} = 0.61104582843$$

	Predicted	
Actual	Positive	Other
Positive	749 True Positives	700 False Negatives
Other	1308 False Positives	6251 True Negatives

Table 34: Table of Confusion for Positives

$$F1 = \frac{2 \cdot 749}{(2 \cdot 749) + 700 + 1308} = 0.42726754135$$

$$F1Average = \frac{0.61104582843 + 0.57333333333 + 0.42726754135}{3} = 0.5372155677$$