# Machine Learning 1 project. Sentiment Mining in Tweets Regarding Coronavirus

**Introduction**

In today’s digital day and age, social networks and communication platforms are ubiquitous. Twitter is such a platform and is known for its microblogging (Walck, 2013), offering a platform to express ideas into maximum 280 characters. It welcomes more than 321 million active users and thus provides a possibility for mass internet communication (Twitter, n.d.). While this can be useful sometimes, this might also lead to a fast spread of disinformation. In order to understand how fast and widely spread possible disinformation is among the population, a systematic sentiment analysis of a vast sample of tweets could contribute to assess the public opinion, and enable to act upon it if necessary. While the aim of this paper is mainly educational in the light of the Machine Learning 1 course, it might be useful for future research.

In this project paper, we will analyze a data set retrieved from the data science community Kaggle containing 41,158 tweets regarding coronavirus (Kaggle, 2020), and describe the methodological processes of mining large number of tweets using Machine Learning techniques. According to Kaggle, the tweets have been pulled from Twitter and manually annotated. The aim of this paper is to combine different supervised learning methods for text classification and come to the best performing one. First, we preprocess the data in order to filter out a complete and relevant data set. For the purpose of computational efficiency, we perform the model selection on half of the remaining (filtered) data set and only use the entire filtered data set to assess the best performing model’s performance. For each model we built a pipeline combining each of the selected vectorizers (CountVectorizer and Tf-Idf Vectorizer) with each of the selected classifiers (Stochastic Gradient Descent Classifier, Support Vector Machine andmultinomial Naïve Bayes classifier), and they will be treated in that order. The first two classifiers support the use of *random\_state* in their parameters, but the multinomial Naïve Bayes classifier does not. For the sake of replicability, we have instantiated a *random\_state* in the pipelines containing SGDClassifier and Support Vector Machine, but not in those containing the Naïve Bayes classifier.

First, we will describe the data set in detail and then move on to how the data was preprocessed. Subsequently, this paper elaborates on which methods were used and how the experiments were performed. This section will discuss the pipelines and their performances in detail. After that, this paper illustrates the limitations that occurred in this research. Furthermore, we discuss in detail the results of all of the individual pipelines on the subset of the data set (50% as discussed above), as well as the ability of the best performing pipeline to generalize to the entire filtered data set.

**The data set**

The data set was downloaded from the data science community Kaggle (Kaggle, 2020). The original file contained two CSV-files; one for testing and one for training. The original test file contained 3,798 instances and the train file 41,158 instances. Because the size of the original test file represented less than 10% of the train file, and it was clear that we would not use the entire data set for computational efficiency, we decided to only use the train file and do all of our preprocessing steps on that data set. Moving forward, we start with one data set (CSV-file) containing 41,158 data points.

This data set has five input features and one output feature (the class labels). The input features are: “UserName”, “ScreenName”, “Location”, “TweetAt” and “OriginalTweet”. “UserName” is an integer that functions as an index. “ScreenName” is a number that represents the unique and anonymized IDs of the tweets. “TweetAt” is the date when the tweet was posted and “OriginalTweet” is the actual tweet. For this project, we will only be using the latter as the input feature, as we are only interested in the textual content. The last column, the output feature “Sentiment” is the label that we want to predict. This column contains five class labels: “Extremely Negative”, “Negative”, “Neutral”, “Positive” and “Extremely Positive”. According to Kaggle, this data set has been manually annotated. Unfortunately, it did not come with a description of the criteria used to assign a class to a certain tweet. Therefore, after having examined the data in more detail, it was unclear whether there was a well-delineated distinction between “Extremely Negative” and “Negative”, and between “Extremely Positive” and “Positive”. This is why we have merged these classes into respectively “Negative” and “Positive” in the preprocessing of the data set, leaving us with three class labels in which the tweets can be classified: “Negative”, “Neutral” and “Positive”.

**Preprocessing**

As described above, the used data set contains 41,158 instances. We load the data set as a *pandas.DataFrame* and inspect it to see whether there are any missing values. There are in fact 8,666 missing values in the “Location” column, and we subsequently drop the rows containing missing values. Even though this might not be necessary because that column will not be used as an input feature, it is useful to do so for the educational purposes of this paper. In addition to that, we merge the class labels “Extremely Negative” and “Negative”, and “Extremely Positive” and “Positive” in the “Sentiment” column as described above. After that, we take a random sample of 50% of the data set for the purpose of computational efficiency during the training and testing processes. In order to assure replicability, we instantiate a *random\_state*. We are now left with a data set containing 16,246 data points. Furthermore, we verify whether the labels are balanced; meaning that they have the same number of instances per class label in the “Sentiment” column. As this is not the case, we we set all the value counts for the three classes equal to the class with the least amount of instances (“Neutral”, with 3,093 instances), to avoid bias in our training set. By doing that, we obtain 3,093 instances in each class.

After all of these preprocessing steps, we are left with a total of 9,279 instances equally spread across the three classes “Negative”, “Neutral” and “Positive”. This is the subset on which we will perform our model selection.

In order to adequately assess a model’s ability to generalize to unseen data, it is important to split the data into a train and test set. However, there is a notion of ‘information leakage’ that is playing a noticeable role here. To be more precise; when running models multiple times, the test set can become part of the training set, as it is not the first time that the model sees it. Therefore, it is useful to split the data into a training, validation and test set. All of the pipelines will be fit on the training part an the performance of all of the individual pipelines will be assessed on the validation part. When we have found the best performing pipeline after model selection, we will retrain that pipeline on the training data and assess its performance on the test set that has been kept apart until then. This way, our model will not get too optimistic after several rounds of fitting. Using the SciKit-learn module *train\_test\_split*, we define a test part containing 20% of our data set and a training part containing 80% of our data set. Subsequently, we define a validation split containing 25% of our training part. Furthermore, we instantiate a *random\_state* for replicability purposes, *shuffle* the data so that it is no longer in any particular order (for example ordered by class label) and we *stratify* the data so that each of the train, validation and test parts contain the same amount of instances per class.

Consequently, the train input (*X\_train*) is a vector containing 5,567 training input instances (the tweets), the validation input (*X\_val*) and test input (*X\_test*) are vectors each containing 1,856 instances (the tweets). The train output (*y\_train*) is a vector containing 5,567 training output instances (the class labels), the validation output (*y\_val*) and test output (*y\_test*) are vectors each containing 1,856 instances (the class labels).

**Methods and experiments**

This paper aims to compare pipelines built by combining each of the selected vectorizers (CountVectorizer and Tf-Idf Vectorizer) with each of the selected classifiers (Stochastic Gradient Descent Classifier, Support Vector Machine and multinomial Naïve Bayes classifier). The Tf-Idf Vectorizer (Term Frequency Inverse Document Frequency) calculates word frequencies to define how important a word is in a document, while also taking into account the occurrence of that word in the other documents included in the corpus (Borcan, 2020). The CountVectorizer on the other hand only calculates token counts.

The vectorizer included in the pipeline always has *max\_features* set to 10,000 and uses the *word\_tokenize* module from NLTK as a tokenizer. On top of that, both of the vectorizers have *max\_df* set to 1.0 and *min\_df* set to 1. Even though it might be useful, we decided not to include a *stop\_word* parameter in the vectorizers, as there are several known issues with the English module for stopwords in sklearn (Sklearn documentation, n.d.). The classifier is included in the pipeline with its default parameters. In what follows, this paper will discuss every pipeline as well as its performance on the validation set in detail. After fitting the training data to the pipeline and predicting the class labels based on the validation set, we evaluate the model’s performance using *cross\_val\_score* with a boxplot, a classification report and a confusion matrix.

A short explanation on all parameters we discuss (Raschka & Mirjalili, 2019; Sklearn documentation, n.d.):

* SGDClassifier:
  + *Log* loss function (SGDClassifier): logistic regression loss function which gives a probabilistic output;
  + *Alpha* (SGDClassifier): a constant that multiplies the regularization term. (The higher, the stronger the regularization);
  + *Random\_state*: constant that assures replicability;
  + *L2 penalty*: penalty that squares the coefficients (will thus penalize higher coefficients more);
  + *Max\_iter*: the maximum number of passes over the training data.
* Support Vector Machine:
  + *C* : regularization parameter. The higher, the more “relaxed” the regularization. The smaller, the stricter the regularization;
  + *Gamma*: kernel coefficient for RBF-kernel (Radial Basis Function);
  + *Max\_iter*: maximum number of passes over the training data.
  + *Random\_state*: constant that assures replicability.
* Multinomial Naïve Bayes classifier:
  + *Alpha*: additive smoothing parameter. 0 for no smoothing, 1 for maximal smoothing.
* CountVectorizer and Tf-Idf Vectorizer:
  + *Max\_features*: top number of most occuring features (words) that will be taken into account by the vectorizer;
  + *Max*\_*df*: maximal document frequency. Default set to 1.0, meaning that a word can maximally occur in all of the documents included;
  + *Min\_df:* minimal document frequency. Default set to 1, meaning that a word must minimally occur in one document.

**Pipeline 1. CountVectorizer and Stochastic Gradient Descent Classifier**

This pipeline includes CountVectorizer with *max\_features* set to 10,000 and the NLTK word tokenizer, as well as the Stochastic Gradient Descent classifier with its default parameters, a *log* loss function and a *random\_state*. The other default parameters include *alpha* set to 0.0001, *max\_iter* set to 1000 and *l2* penalty. Judging by the classification report, this pipeline performed relatively poorly, obtaining a macro average f1-score of 0.62. The boxplot of the *cross\_val\_score* shows us that the mean macro average f1-score for the cross-validation is around 0.62. Both the classification report and the confusion matrix seem to point toward a slightly better performance on the “Neutral” class.

**Pipeline 2. CountVectorizer and Support Vector Machine**

This pipeline includes the CountVectorizer with *max\_features* set to 10,000 and the NLTK word tokenizer, as well as the Support Vector Machines classifier with its default parameters. These include *C* set to 1.0, *gamma* set to ‘scale’, an RBF-kernel (Radial Basis Function) and *max\_iter* set to -1 (no limit). The classification report indicates that this pipeline’s performance is relatively unsatisfying, reaching a macro average f1-score of 0.62. Furthermore, the *cross\_val\_score* boxplot estimates the mean macro average f1-score for the cross-validation slightly beneath 0.60. Both the classification report and the confusion matrix show a slightly better performance on the “Neutral” class.

**Pipeline 3. CountVectorizer and multinomial Naïve Bayes classifier**

This pipeline combines the CountVectorizer with *max\_features* set to 10,000 and the NLTK word tokenizer and a multinomial Naïve Bayes classifier with its default parameters, which has *alpha* set to 1.0. This pipeline reaches a macro average f1-score of 0.63 in the classification report, which indicates that it performs slightly better than the second pipeline, without outperforming the first one. The *cross\_val\_score* boxplot indicates that the mean macro average f1-score for the cross-validation is slightly above 0.60. The classification report as well as the confusion matrix show a slightly better performance on the “Negative” class.

**Pipeline 4. Tf-Idf Vectorizer and Stochastic Gradient Descent Classifier**

This pipeline includes the Tf-Idf Vectorizer with *max\_features* set to 10,000 and the NLTK word tokenizer, as well as the Stochastic Gradient Descent classifier with its default parameters, *log* loss function and a *random\_state*. The default parameters of the SGDClassifier include *alpha* set to 0.0001, *max\_iter* set to 1000 and *l2* penalty. The classification report points toward a macro average f1 score of 0.68, while the *cross\_val\_score* boxplot estimates a mean macro average f1-score for the cross-validation. It seems that there is a more equal distribution across the predictions for each of the classes in this pipeline.

**Pipeline 5. Tf-Idf Vectorizer and Support Vector Machine**

In this pipeline, we combine the Tf-Idf Vectorizer with *max\_features* set to 10,000 and the NLTK word tokenizer, as well as the Support Vector Machines classifier with its default parameters, which include *C* set to 1.0, *gamma* set to ‘scale’, an RBF-kernel (Radial Basis Function) and *max\_iter* set to -1 (no limit). Reaching a macro average f1-score of 0.66, this model is rather mediocre. The boxplot for the cross-validation confirms this by indicating a mean macro average f1-score of around 0.64. The confusion matrix indicates that the model performs slightly better on the “Neutral” class.

**Pipeline 6. Tf-Idf Vectorizer and multinomial Naïve Bayes classifier**

In this pipeline, we include the Tf-Idf Vectorizer with *max\_features* set to 10,000 and the NLTK word tokenizer, together with a multinomial Naïve Bayes classifier with its default parameters. This classifier does not support the use of *random\_state* (replicability), so it is not instantiated. Judging by the classification report, this pipeline also covers mediocre ground, reaching a macro average f1-score of 0.63. Even though this does not outperform the pipeline with CountVectorizer and multinomial Naïve Bayes classifier, the cross-validation boxplot for this pipeline indicates a mean macro average f1-score of around 0.63 (compared to 0.60 for the pipeline with CountVectorizer and NBC). The confusion matrix indicates that this pipeline slightly underperforms on the “Neutral” class.

**Pipeline 7. Final assessment of pipeline 4 on the test set**

In this part, we retrain pipeline 4 on the training data set and assess its performance onto the test set that has been kept apart until now. Pipeline 4 was the best performing model, reaching a macro average f1-score of 0.68, containing the Tf-Idf Vectorizer and the Stochastic Gradient Descent classifier. The pipeline seems to perform slightly worse on the unseen test set, reaching a macro average f1-score of 0.67, which is also confirmed by the mean cross-validation macro average f1-score in the boxplot. As was the case with pipeline 4, this pipeline also slightly performs better on the “Neutral” class.

**Limitations**

The data set that we used from Kaggle had been annotated manually (Kaggle, 2020). However, there was no description available for the criteria used to classify the tweets in their respective categories. The distinction between Extremely Positive and Positive, as well as the distinction between Extremely Negative and Negative was therefore unclear. On that account, we decided to merge those categories into respectively Positive and Negative, leaving us with three categories: Positive, Neutral and Negative.

**Results**

Pipeline 4 performed the best on our subset of the dataset, reaching a macro average f1-score of 0.68. This was the pipeline containing the Tf-Idf Vectorizer (with *max\_features* set to 10,000 and the NLTK word tokenizer) and the Stochastic Gradient Descent classifier (with *log* loss function and its other default parameters). As shown in pipeline 7, it performed slightly worse on the unseen test set, reaching a macro average f1-score of 0.67.

Pipeline 3 had the lowest performance on our subset of the data set, reaching a macro average f1-score of 0.60. This pipeline contained the CountVectorizer (with *max\_features* set to 10,000 and the NLTK word tokenizer) and the multinomial Naïve Bayes classifier with its default parameters.

**Conclusion**

In this paper, we investigated six pipelines that combined each of the selected vectorizers (CountVectorizer and Tf-Idf Vectorizer) with each of the selected classifiers (Stochastic Gradient Descent Classifier, Support Vector Machine and multinomial Naïve Bayes classifier). Even though all of the performances are comparable and within a similar range of 0.60 to 0.68 in macro average f1-score, pipeline 4 was the best performing one. This pipeline consisted of the Tf-Idf Vectorizer (with *max\_features* set to 10,000 and the NLTK word tokenizer) and the Stochastic Gradient Descent classifier (with *log* loss function and its other default parameters). When we fit this pipeline on the training set and used it to predict on the unseen test set, it reached a macro average f1-score of 0.67. The poorest performance came from pipeline 3, containing the CountVectorizer (with *max\_features* set to 10,000 and the NLTK word tokenizer) and the multinomial Naïve Bayes classifier with its default parameters, reaching a macro average f1-score of 0.60.

As stated above, there were some important limitations to this project. First of all, we merged the original five class labels into three class labels: “Positive”, “Neutral” and “Negative”. Secondly, we only used half of the data set for computational efficiency.

In conclusion, pipeline 4 was the best performing pipeline, reaching a macro average f1-score of 0.68 during model selection and 0.67 during the assessment on the test set. Pipeline 3 had the lowest performance, reaching a macro average f1-score of 0.60.

For this project, we only used the textual content in the “OriginalTweet” column to predict class labels. For future research, it might be interesting to include “Location” or “TweetAt” to see how these could potentially influence the “Sentiment” classification.

**Reference list**

Borcan, M. (2020, June 8). TF-IDF Explained And Python Sklearn Implementation - Towards Data Science. Retrieved from https://towardsdatascience.com/tf-idf-explained-and-python-sklearn-implementation-b020c5e83275

*Coronavirus tweets NLP - Text Classification*. (2020). [Data set]. Retrieved from https://www.kaggle.com/datatattle/covid-19-nlp-text-classification

1.9. Naive Bayes — scikit-learn 0.24.0 documentation. (n.d.-a). Retrieved December 23, 2020, from https://scikit-learn.org/stable/modules/naive\_bayes.html

sklearn.feature\_extraction.text.CountVectorizer — scikit-learn 0.24.0 documentation. (n.d.-b). Retrieved December 23, 2020, from https://scikit-learn.org/stable/modules/generated/sklearn.feature\_extraction.text.CountVectorizer.html

sklearn.linear\_model.SGDClassifier — scikit-learn 0.24.1 documentation. (n.d.). Retrieved December 22, 2020, from https://scikit-learn.org/stable/modules/generated/sklearn.linear\_model.SGDClassifier.html

sklearn.feature\_extraction.text.TfidfVectorizer — scikit-learn 0.24.0 documentation. (n.d.-c). Retrieved December 23, 2020, from https://scikit-learn.org/stable/modules/generated/sklearn.feature\_extraction.text.TfidfVectorizer.html

Raschka, S., & Mirjalili, V. (2019). *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2, 3rd Edition*. Birmingham, United Kingdom: Packt Publishing.

Twitter. (n.d.). In *Wikipedia*. Retrieved December 22, 2020, from https://en.wikipedia.org/wiki/Twitter#2020

Walck, E. P. (2013). Twitter: Social Communication in the Twitter Age [Review of the book *Twitter: Social Communication in the Twitter Age*, by D. Murthy]. *International Journal of Interactive Communication Systems and Technologies*, 3(2), p. 66-69. Retrieved from https://www.igi-global.com/pdf.aspx?tid%3D105658%26ptid%3D71575%26ctid%3D17%26t%3Dtwitter%3A+social+communication+in+the+twitter+age