Sentiment analysis on tweets using decision trees, random forests and other parallel and sequential ensemble methods

Elin Finstad & Anders Bråte
Department of Physics
University of Oslo
Norway
December 15, 2020

The vast amount of data readily available to nearly anyone means employing machine learning algorithms is easier, and can be more useful than ever. So much of our daily lives is influenced by machine learning algorithms. Some might even blame the political and social turmoil in the USA on them. This shows just how important these algorithms have become, and is exactly our reason for studying them in our paper. More specifically we study algorithms for deciding the subjective state, or mood, of a tweet, using decision trees and ensemble methods. As expected, the much too simple decision tree was outperformed by ensemble methods such as random forests, but no more than 76% accuracy was achieved at best, due to weaknesses in the methods, and errors in the dataset.

I. INTRODUCTION

Since the early days when machine learning was used to filter out spam emails, it might not have gotten any more useful since that is pretty hard to beat, but it has gotten a lot more complex. Now countless machines, programs and algorithms employ machine learning algorithms to make sense of more and more readily available datasets. Steady yet massive improvements in computer hardware technology has made machine learning even more employable, as storage capabilities increase along with computation power.

Along with the rise of computational power, we see the rise of social media. Many platforms, more popular than ever, tie people, corporations and nations together like never seen before, and the amount of data is MASSIVE. Due to this, understanding, interpreting and analyzing social media data naturally falls under the umbrella of machine learning.

So called 'Sentiment analysis' in our context is the application of machine learning and data manipulation to identify subjective states in a piece of text. More specifically the mood of the tweet, whether it is negative or positive. Being able to deduce this using machine learning has a multitude of potential uses, such as marketing and brand popularity analysis, and even larger social science applications, to simply helping costumer service agents pick the angriest and most urgent requests.

Though not usually associated with sentiment analysis, we study decision trees. This is due to their simplicity and ease of implementation, as well as how well they lend themselves to use in more complex ensemble methods. Ensemble methods such as bagging, boosting and random forests are the natural progression from simple trees. The key element we wish to explore in all this is how a weak learner, which the decision tree certainly is, can be combined to form a stronger learner using ensemble methods.

We start by looking at the simple decision tree, then we naturally progress to ensemble methods. Here we look into bagging, random forests and finally boosting methods consisting of adaptive and extreme gradient boosting.

II. THEORY

Decision trees

In its most basic sense, decision trees for categorization are series of true/false statements that divide the data into subsets. The end goal is to find the combination of questions regarding the features that is able to best divide the data, and achieve a low 'impurity'. The term purity is simply explained the degree to which the data is divided, and is calculated as the square sum of the

fraction of items with the given label.

$$Gini = 1 - \sum_{i=1}^{J} p_i^2.$$

Here, p_i is the fraction of data labeled as type i. This simple equation is called the Gini score, and is used to find how well the split with regards to a feature divides the data.
Another function that is used in the same capacity is the entropy function, borrowed from thermodynamics. This function tends to produce more balanced trees where Gini tends to isolate the most frequent class in its own branch of the tree [1]. We choose to use the Gini function, as the difference is often minimal, and Gini is slightly faster to calculate, which is relevant due to our high number of features.

Initially the Gini score of all features are calculated to find the optimal starting split (root) of the data. This process is then repeated as the method iterates its way down the tree. When a split does not yield a better Gini score the branch is ended in what is known as a leaf. This method is a greedy one, as each split and branch is picked out of the current best split, without consideration to the final result.

It is now easy to see how adding layers in this tree adds complexity meaning that the everlasting problem of overfitting is a factor, and so regularization parameters are often used. The most apparent regularization parameter is simply limiting the depth of the tree. This issue of overfitting also means that decision trees are very high variance, meaning small, sometimes unimportant features of the data are over-represented in the final model and thus reducing the general usefulness of the model.

In addition, we see that this simple method for handling data is just that, a bit simple, and so it is known as a weak classifier. This means that it can be expected to perform better than just guessing, but not much (slightly better than 50% accuracy). However decision trees do lend themselves to being used in ensemble methods for the reasons mentioned above.

Ensemble methods

Ensemble methods use several learning algorithms to increase prediction rate from what can be produced using the learning algorithms alone. The methods can be divided in two groups: sequential ensemble methods and parallel ensemble methods, based on whether the base learners are generated either sequentially or in parallel.

Bagging

Bootstrap aggregation (bagging) ensemble methods are designed to improve stability and accuracy of machine learning algorithms. This means that for an increased number of estimators, the tendency towards overfitting will stabilize, rather than increase as the complexity of the model increases. This allows for models that are complex enough to predict more nuanced features.

The bagging method uses the same training algorithm for every predictor, but train them on different subsets of the training data. When the sampling is performed with replacements it is called bagging, while it is called pasting when the sampling is performed without replacement. Since the samples can be generated independently, the data generation and the training can be done in parallel. Hence, bagging is part of the parallel ensemble methods.

This method essentially generates many decision trees which have high variance, but the average of these produce a model which has a reduced variance.

$$\hat{f}_{bag} = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^b$$

Here, we have picked out B different training sets from the dataset, and we therefore have B corresponding decision trees, which are used to find the average, that is the bagging model.

Random forest

Random forest is a type of parallel ensemble method which uses decision trees. When used for classification, the random forest method construct multiple decision trees and outputs the most frequently occurred class from all the decision trees.

In general, random forest is an improvement of bagged decision trees. The random forests are often trained with bagging on decision trees with max samples equal to the size of the training data. In addition, they introduce more randomness. When splitting the nodes, it does not search for the very best feature selection, as for the standard decision trees, but instead finds the best feature among a random subset of features. This increases the diversity of the trees, such that the variance decreases, but unfortunately at the cost of a higher bias [1].

Boosting

Boosting is a collection of the ensemble methods that can combine multiple weak learners into a strong learner. Weak learners are classifiers that only produce predictions slightly better than what can be achieved with random guessing. Boosting iteratively builds ensemble methods by training each model on the same dataset,

but with weights on the instances adjusted according to the error in the previous prediction. The idea is to force the model to focus on the instances that are hard to predict. Since the weights are updated for each iteration of the algorithm, the whole process is done sequentially, and thus boosting is part of the sequential ensemble methods. The collection of boosting methods is large, but the most popular ones are adaptive boosting (AdaBoost) and gradient boosting.

AdaBoost

The idea of AdaBoosting is to set weights to both the classifiers and the samples, such that the method is forced to concentrate on the observations that are hard to classify. The procedure starts by defining a weak classifier, which for the AdaBoost method preferably is a decision tree. Usually these decision trees have max depth 1, called decision stumps. Then the weight of each sample is initialized to $1/m$, where m is the total number of training samples. We define a classification function $G(x)$ which returns the prediction. In our case, this will be the result from the decision tree. The error rate of the training samples is then given by

$$\overline{err} = \frac{1}{n}\sum_{i=0}^{n-1} I(y_i \neq G(x_i)),$$

where y_i is the true value and n the number of training samples. Since we want to add different weights to the error rates depending on how good the prediction is, the weighted error rate of the m-th predictor is given by

$$\overline{err}_m = \frac{1}{n}\frac{\sum_{i=0}^{n-1} w_i^m I(y_i \neq G_m(x_i))}{\sum_{i=0}^{n-1} w_i^m}$$

[2]. The next step is to compute the predictor's weight α_m,

$$\alpha_m = \eta \log \frac{1-\overline{err}_m}{\overline{err}_m}.$$

Here, η is the learning rate telling how much the weight is boosted at each iteration [1]. Further, the instance weights are updated by

$$w_i^{m+1} = \begin{cases} w_i^m & \text{if } y_i = G_m(x_i) \\ w_i^m \exp(\alpha_m) & \text{if } y_i \neq G_m(x_i) \end{cases}$$

[1]. Next, all the instance weights are normalized by dividing by $\sum_{i=0}^{n-1} w_i$. Finally, a new predictor is trained using the updated weights. This finishes one iteration, and the whole process starts over again with computing the predictor's new weights, update the instance weights and train a new predictor. The process is completed when a number of max iterations is reached or when a perfect predictor is found.

To make the final prediction, the AdaBoost method computes the predictions of all the predictors and weights them using the predictor weights α_m. The final predictions is then the one with the majority of the weighted votes.

Gradient Boosting and Extreme Gradient Boosting

Much like AdaBoost, Gradient boosting is an iterative scheme where a model is updated based on the previous models result. The difference however is that while AdaBoost looks at the weights, Gradient boost looks at the residual error.

Originally, Gradient Boosting was an improvement of AdaBoost for regression problems, but the method can be use to optimize both regression, categorization and ranking problems. The term 'gradient' refers to the fact that the method used two or more derivatives of the same function. Gradient Boosting is an iterative functional gradient algorithm, meaning it minimizes a loss function by iteratively choosing a function pointing in the direction of the negative gradient.

At each iteration the residual error is calculated. This residual error is then used to produce the next decision tree. By repeating this for an ensemble of n trees, the predictions of the trees can be summed to make predictions on a new instance [1]. A learning rate can be added to scale the contribution of each tree. The method would need an ensemble of more trees to fit the training data when the learning rate is set to a low value, such as 0.1. Even though a larger collection of trees is needed, the predictions usually generalize better for lower learning rates [1].

Bag of words

Bag of words is a method to extract features from text documents. This method creates a vocabulary of all unique words in the corpus (collection of documents) and tells how frequently they occur in the text. It is called a bag of words since the order of the words is not considered. Structurally, the bag of words is usually stored in a $D \times N$ matrix, where D is the number of documents in the corpus and N is the number of tokens (unique words). This is a representation of the vocabulary. To reduce the size of this matrix it is crucial to perform a preprocessing of the data, which will be described later.

Once the vocabulary is created, the occurrence of the unique words needs to be scored. This can be done by e.g. binary representation, counting or frequency. The binary representation tells whether a word appears in the document or not. For counting, the element $x_{i,j}$ of the matrix tells how many times the token j has appeared in the document i, while when using frequency the element $x_{i,j}$ represent the frequency of token j in document i.

TF-IDF

Scoring words by frequency can cause highly frequent words to dominate the document, which can cause problems if the word contains little informational content compared to less frequent words. One solution to this problem is to rescale the frequency by how often the word appears across all the documents. This approach is called Term Frequency - Inverse Document Frequency (TF-IDF). The term frequency is calculated as before

$$TF(term) = \frac{Number\ of\ times\ term\ appears\ in\ document}{Total\ number\ of\ items\ in\ document},$$

but is now multiplied by the inverse document frequency

$$IDF(term) = \log\left(\frac{Total\ number\ of\ documents}{Number\ of\ documents\ with\ term\ in\ it}\right).$$

Hence, the final TF-IDF score is

$$TF\text{-}IDF(term) = TF(term) \cdot IDF(term).$$

Data preprocessing

Data preprocessing refers to techniques of cleaning and organizing raw data for further analysis. In our case we will use the data preprocessing to reduce the number of unique words of all the tweets, which in turns will be used as features. We also remove some special instances of characters that in our case affects the algorithm in unwanted ways. When the number of features is close to the number of training samples, the chance of overfitting increases. By reducing the number of features, the overfitting may reduce as well. In addition, if no preprocessing is done, we may end up exceeding the storage capabilities of our hardware due to the large amount of features.

An important part of preprocessing of text data for sentiment analysis is to remove so called 'stop words'. Stop words is a collection of 'fill words' that does not affect the sentiment of the data or words that are commonly used in almost all training samples independently of the sentiment. The selection of these words is a topic of discussion as removing certain words may affect the sentiment of a tweet, and should be adjusted depending on the data [3].

Another way to reduce the number of features is by removing special characters and handling emoticons. Special characters like period and comma does in very small amount affect the sentiment of the data and can by no concern be removed. Other special characters like exclamation marks may to a greater extent enhance the feeling presented, and are more up for discussion whether to be removed or not. Further, combinations of special characters in the form of emoticons may have a great impact on the sentiment of the data. The emoticons ":)" and ":(" alone states a positive and negative sentiment respectively, and can change the whole mood of a tweet. One way to reduce the number of features is to replace all positive emoticons by 'positive' and all negative emoticons by 'negative'.

The number of features can also be reduced by adding spell correction and removing different conjugations of words using a stemmer. A stemmer reduces words such as 'running', 'runner' and 'runs' to 'run', leaving us with only one version of the word. Spell correction also helps reducing the number of unique words.

III. METHOD

Data

We use pre-labeled tweets publicly available on Kaggle[1]. The original dataset contains 1.6 million tweets together with several other features, including a sentiment label, 0 when negative and 4 when positive. In our analysis we have trimmed the original data, extracting various amounts of random tweets, with the two sentiments equally distributed. Since we are only interested in the sentiment of the tweet, all other features are removed to reduce the size of the data set.

The original data is collected through Twitter API and labeled using distant supervision [4]. The labeling is done by training a model on data with emoticons. If the tweet contained ":)" it was classified as positive, and if the tweet contained ":(" it was classified as negative. The model was then tested on a test set containing tweets both with and without emoticons. Our data set is labeled by this model.

Tweets have several characteristics. They consist of at most 140 characters, causing the users to use more abbreviations to fit everything within the limited number of characters. The tweets are also often written with an informal language, hence they often contain misspellings and slang.

Data preprocessing

When preprocessing the data we start by removing duplicated tweets and retweets, to avoid some tweets being weighted more. In general retweets are marked with the abbreviation "RT", hence all tweets containing this are removed. The duplicated tweets are removed

[1] https://www.kaggle.com/kazanova/sentiment140 Accessed: 12-12-2020

using `DataFrame.drop_duplicates()` from the python package pandas [5].

We create lists of emoticons based on the sideway latin-only emoticon list from Wikipedia [6] to use in feature reduction and deletion of tweets. The emoticons are divided in positive, negative and neutral emoticons. All tweets containing some variation of ":P" are deleted, due to some issues with Twitter API at the time when the dataset was created [4]. The issue caused tweets containing ":P" to be labeled as negative, but usually ":P" does not imply any negative sentiment. Hence, these tweets are deleted from the data set.

To reduce the number of features, we replace the emoticons with their sentiment polarity based on the created emoticon lists. All positive emoticons are replaced by "positive" and all negative emoticons are replaced by "negative". All neutral emoticons are removed, since they do not contribute to the sentiment of the tweet. In addition all tweets containing both positive and negative emoticons are deleted from the dataset to avoid positive features marked as part of negative tweets or negative features marked as part of positive tweets [4].

Further, we take advantage of the special twitter model properties to reduce the number of features. Usernames are often included in the tweets, marked with "@" followed by at most 15 alphabetical characters (with the exception of underscore "_"). We replace them all with "username". We also replace all url-links with "url", which are often included by the users of Twitter. In addition, tweets are written with a casual language and thus some words are written with arbitrary repetitions of selected characters to emphasize the word. We replace such repetition of characters with only two repetitions.

We use the list `stopwords.words('english')` from the package `nltk` as base for our stop word list [7]. This list contains negations like "no", "not", "don't", "haven't", "aren't" etc. Since we are trying to classify tweets based on their sentiment, these negations are important as they can turn a negative tweet positive when removed. Hence, we remove all negations from the list. Instead, we handle the negations separately and replace them all by 'not'. This is equivalent to writing out the abbreviations and apply the updated stop word list, just less expensive.

For stemming we use the `stemmer.SnowballStemmer('english')` from the `nltk` package [7]. This is applied to a tokenized version of the tweets.

Feature extraction

To extract features to train our model we use the `feature_extraction.text.TfidfVectorizer` from the Scikit-Learn package in Python [8]. The feature extractor returns a sparse matrix with the shape $D \times N$ containing the TF-IDF score described in section II. Here, D is the number of tweets and N is the number of unique words in the collection of all tweets. The vectorizer is fitted to the collection of tweets.

Decision trees

To create decision trees we use `tree.DecisionTreeClassifier` from the Scikit Learn package in Python [8]. Here, we set the criterion for splitting to be the gini score. There are also several parameters to restrict the size of the tree, where we will use `max_depth` to limit the depth of the tree. This is done to avoid overfitting, which is often an issue with decision trees.

Ensemble methods

For all ensemble methods we use classifiers offered by the Scikit-Learn package in Python [8] through the `sklearn.ensemble` module, except for XGBoost, which is offered by the `xgboost` package [9].

First out is the bagging method for classifications `BaggingClassifier`. This classifier can be used as a bagging method with samples drawn randomly with replacement by setting the parameter `bootstrap=True`. By turning this to false `bootstrap=False`, the method changes to pasting and thus the samples are drawn without replacement. We use the bagging method.

For the random forests we continue with the Scikit Learn package, and use `RandomForestClassifier`. This method is roughly equivalent to using the `BagginClassifier` on a `DecisionTreeClassifier` with random splitting, just more convenient and optimized for decision trees.

Continuing to the ensemble method boosting, we use `AdaBoostClassifier` from the same `sklearn.ensemble` module. Scikit-Learn uses a multiclass version of AdaBoost called SAMME, short for Stagewise Additive Modeling using a Multiclass Exponential loss function [10]. When used on a binary classification problem, SAMME is equivalent to AdaBoost. Scikit-Learn can also use another variant of SAMME called SAMME.R when the predictors can estimate class probabilities. Instead of relying on predictions, this variant relies on probabilities and does in general perform better [1].

Finally, we look at gradient boosting. For this method we look at the extreme gradient boosting method, which is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable [2]. It

provides a parallel tree boosting, making it more efficient than the standard gradient boosting algorithm. In addition it is known to be more memory efficient, which is useful with our amount of features. The method is implemented by using `xgb.XGBClassifier` from the `xgboost` package in Python [9].

Accuracy score

To tell how well our model has learned we calculate an accuracy score, which gives the percentage of successful classifications. The accuracy score is given by

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}.$$

Here t_i is the guessed targets, y_i is the true value and I is the indicator function for binary classification, 1 if $t_i = y_i$ and 0 otherwise. This score is implemented using the `accuracy_score` from the module `sklearn.metrics` in the Scikit-Learn package in Python [8].

IV. RESULT

Data preprocessing

Before we can do any actual machine learning, we must do preprocessing of the data. Here we wish to see how different methods of data reduction and altering affects both the number of unique words, which defines the size of the bag of words, and how this affects the accuracy of certain methods.

Table I. The reduction in number of unique words as different reduction types are applied to the original 1.6 million tweets. Changing all the unique usernames to 'username' gives the highest feature reduction.
*All characters repeated three or more times sequentially are reduced to two
**Miscelaneous characters are all but round and square brackets, dash, hyphen and pound sign.

Reduction type	Unique words	Percent reduction
None	684358	0.00 %
Stemming	671509	1.87 %
Usernames	368743	46.1 %
Urls	622336	9.06 %
Sequential letters*	670826	1.97 %
Misc**, negations & stop words	586747	14.2 %
All	277680	59.4 %

Table II. The change in accuracy as certain data reduction methods are applied. Here, the AdaBoost algorithm with 200 decision stumps and a learning rate of 0.9 is applied on 100 000 tweets. Applying all the reduction methods increases the accuracy by 1.1 % when using AdaBoost. The difference in test and train accuracy is overall small.

	Test	Train	Change in test data from no reduction methods
All reduction methods	74.1 %	74.8 %	1.1 %
Remove all but emoticons	73.8 %	74.8 %	0.8 %
Remove all but stemming	73.3 %	74.5 %	0.3 %
No data reduction	73.0 %	74.8 %	0 %

Table III. Change in accuracy for the Decision tree method, with depth 100 and gini evaluation applied on 100 000 tweets. Applying all reduction methods reduces the difference between train and test data, in addition to increasing the test accuracy by 2.6 %.

	Train	Test	Change in test data from no reduction methods
All reduction methods	70.9 %	87.0 %	2.6 %
No reduction methods	68.3 %	89.5 %	0 %

Decision trees

We start the analysis by looking a the accuracy score for a single decision tree with varying maximum depth.

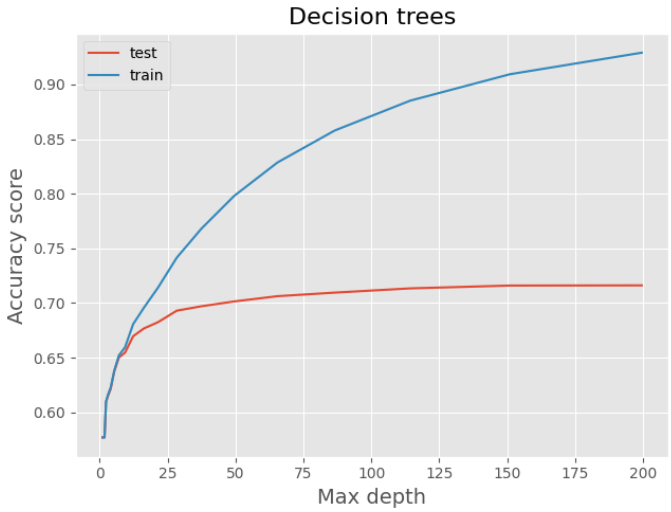

Figure 1. The accuracy score for decision trees for both train and test data for varying max depth of the decision tree. The model is trained on 200 000 preprocessed tweets. The accuracy for train data increases towards 1 while the test accuracy stabilizes at 0.71.

Bagging

In bagging, we have two parameters which are interesting to look at, the depth of the trees which follows from the tree itself, and how many we need, since bagging consists of multiple trees.

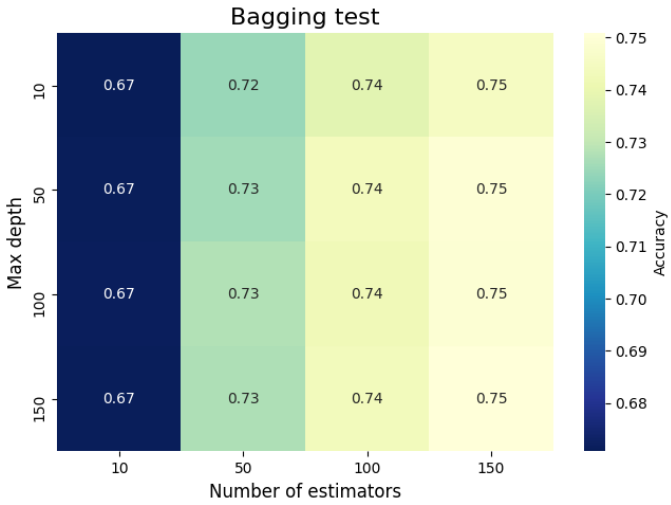

Figure 2. Accuracy score for bagging on test data, using a various amount of estimators and depth. This is applied on a dataset of 200 000 preprocessed tweets. As can be seen, the main factor deciding accuracy is the number of estimators, whilst the depth has little effect on the result.

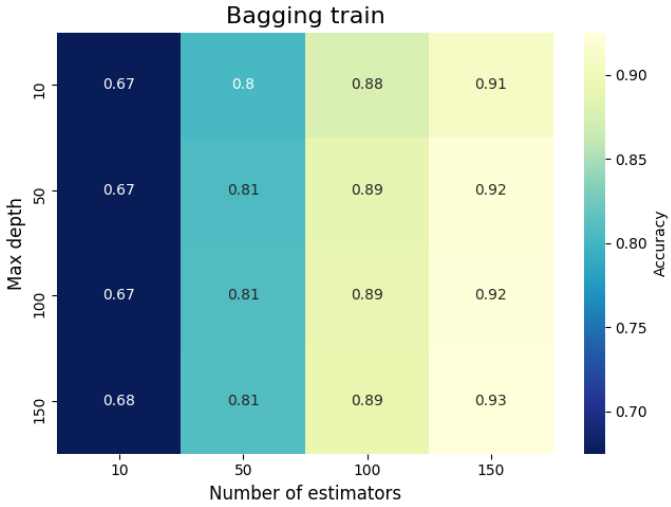

Figure 3. Accuracy score for bagging with regards to the train data of a 200 000 preprocessed tweets dataset. We see that the accuracy is largely dependent on the numbers of estimators, however the depth of each tree also affects the accuracy. In comparison to the accuracy of the test data fig. 2 we see that this is a bigger factor than on the test data.

Random forest

With random forests, both the number of estimators and the maximum depth of the decision trees are of interest. However, the depth had very little effect on the data, much like in bagging, meaning we only include the number of estimators.

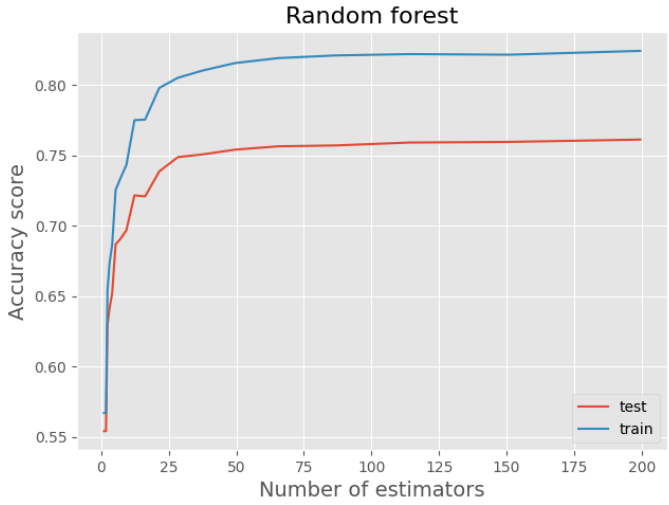

Figure 4. Accuracy score for random forests for both train and test data for varying size of the forest. When the number of estimators exceed 50 both train and test accuracy stabilizes at 0.82 and 0.76 respectively. The random forest consist of decision trees with a maximum depth of 50. The model is trained on 200 000 preprocessed tweets.

AdaBoost

The goal of AdaBoost is to take a set of weak learners and combine them into a stronger learner. As stated in the theory section II, the method is preferably used on decision trees of depth 1. We can see from fig. 1 that decision trees with depth 1 have a test accuracy score of 0.57, a clearly weak learner. Hence, we continue with decision stumps when testing the AdaBoost algorithm, and the results are shown in fig. 5.

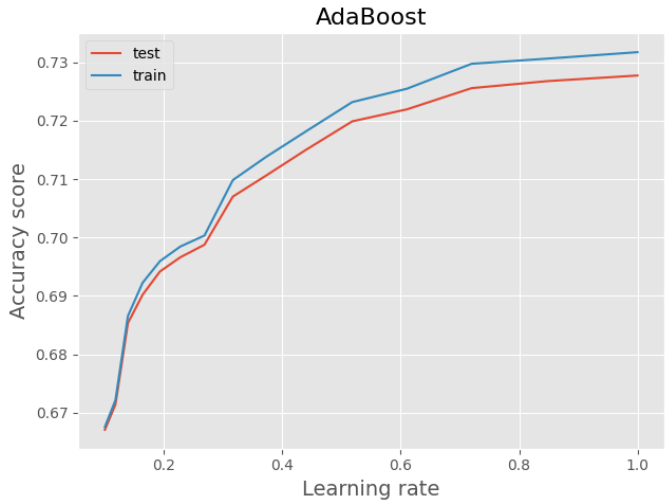

Figure 5. Accuracy score for the AdaBoost method used on decision trees with max depth 1. The accuracy score for both test and train data increases as the learning rate increases with very little overfitting. The model is trained on 200 000 preprocessed tweets.

XG Boost

XG boost is a premade package which features many different methods of optimization. For brevity and simplicity however we choose to only look at the learning rate. This is also because, as with bagging and random forest previously, the depth of the tree had a very similar effect.

Figure 6. The accuracy for XGBoost for varying learning rate. Here we use a 100 000 tweet dataset, with a tree depth of 6. As can be seen, this method behaves a bit different from the others, as it starts at a very stable test data accuracy, whilst training increases in a similar fashion to AdaBoost fig. 5.

DISCUSSION

Preprocessing

As is the crux of all machine learning, the quality of the dataset is of upmost importance. It is therefore a point of discussion to what degree the labeling of our dataset is an accurate one. As stated in section III, the dataset we used was labeled using an algorithm which looked for emoticons, and labeled the tweet based on the sentiment of the emoticons it contained. This does enable us to use a large dataset (1.6 million tweets), however the drawback is one of quantity vs quality. If the labels indeed show the polarity of the tweet, then our algorithms will be able to classify actual sentiment of a tweet, but if the labeling instead is affected more by an algorithm which does not show the true polarity of the tweet, we will rather be attempting to recreate that very labeling algorithm. In order to achieve 100 % we would just have to recreate the algorithm that looks for emoticons in order to label the tweets, which is far too simple to actually be able to analyze the sentiment of a real life tweet.

To further reduce the number of features, we tried to do spell checking using `SpellChecker()` from the `spellchecker` package. Since tweets contain a large amount of slang and abbreviations, this did more harm than good. We thus found it better to just leave the misspelled words as they are instead of making wrong corrections.

As is very common in most sentiment analysis we must reduce whole tweets to simply a list of words and their occurrence, known as a 'Bag of words'. This removes the order and often the context of a word, leaving a tweet nearly unreadable to a human. However, the idea is that this will still convey the sentiment of the tweet through key words alone. To what degree removing the order and context affects the tweet is hard to say, but our results do indicate that it is not foolproof. Despite the perceived shortcomings of this method, it does seem the easiest and most adaptive method for sentiment analysis, which is also shown in its widespread usage.

Mentioned earlier, the effect of removing certain words or features in a tweet is always a point of debate. However, as can be seen in table II the removal of diametrically opposed emoticons, as well as replacing positive and negative emoticons with the corresponding sentiment written in plain text does indeed improve the accuracy score.
From the same table we can see how stemming of words also improve accuracy, despite this often produces some strange and incorrect grammatical inflections.
All in all the total feature reduction does improve the accuracy of the methods. In addition we see in table

III how the decision tree method benefits from the data preprocessing even more, as 2.6 % improvement in the test score and a 2.5 % decrease in training accuracy shows improvement in both accuracy and reduction in overfitting, which fits with theory.

In our preprocessing we have in large part attempted to reduce so called 'corpus-specific stop words', which are words that occur so frequently as to uniformly appear in both positive and negative tweets. Indeed when looking at the `max_df` parameter of the bag of words function `TfidfVectorizer`, which removes tweets that occur in a given percentage of all tweets, we find that only one word occurs in more than 30 % of tweets. In comparison, looking at the same parameter, we see that for unprocessed data, 29 words occur in more than 30 % of tweets. It should also be noted that instances of usernames, which stands for more than 46 % of unique words are all replaced with the string 'USER', meaning this will occur in a very large amount of tweets.

From table I we see how much some features reduce the number of unique words, as they are applied to the original dataset of 1.6 million tweets. This combined with the fact that reducing certain features increases the accuracy, and reduced overfitting is a very good sign. The preprocessing of data is also surprisingly fast, requiring only a few minutes for even 1.6 million tweets on a standard laptop.

Decision trees

As mentioned in section II, decision trees are weak learners, hence we expect an accuracy score around 0.5. From fig. 1 we see that the test accuracy score stabilizes at 0.71, while the training accuracy keeps increasing towards 1. Even though the test accuracy of 0.71 is a bit higher than expected for a weak learner, this is only reached when the model is overfitted. When creating a decision tree without overfitting, the accuracy score seems to not exceed 0.65 (max depth of 10 branches), which is a more reasonable accuracy for a weak learner.

Ensemble methods

The goal by introducing ensemble methods is to improve the results given by the weak learner, which in this case is the single decision tree.

Bagging

Looking at the two heat maps figs. 2 to 3 for test data and train data respectively, we see how the accuracy is in large part affected by the number of estimators rather than the depth of each estimator (tree). The depth is however still a bigger factor in the train data, as is expected since this often increases faster and more than the test data.
As with the other methods, the test data reaches an accuracy of 75 %, however the train data is very high, as this reaches 95 %.

Random Forest

It is clear from fig. 4 that random forests reduces overfitting and increases the test accuracy compared to a single decision tree. The plot is produced with a fixed maximum depth of 50 branches. By fig. 1, the single decision tree had a test accuracy of 0.7 and a train accuracy of 0.8 at this point. Now, for random forest, the test accuracy has increased to 0.76 while the train accuracy stabilizes at 0.82. This means the difference between train and test accuracy has decreased by 4 %, and thus there is less overfitting, which is expected of a random forest. The increase in test accuracy is not extraordinary, but it is still a little better than for a single decision tree.

AdaBoost

Figure 5 shows an increase in the accuracy score from the weak learner decision stumps to the strong learner designed by the AdaBoost method. From fig. 1 we have that a single decision tree with depth 1 gives an accuracy of 0.57. When applying the AdaBoost method, we increase this accuracy up to close to 0.73 for the test data. Hence, the method has succeeded in combining multiple weak learners to a strong learner.

On the other hand, the accuracy of the AdaBoost method does not outperform a deeper decision tree. We have from fig. 1 that a single decision tree can get an accuracy up to 0.71, but at the cost of overfitting. The AdaBoost method only gets to a test accuracy of 0.73, but unlike the deep decision tree the train accuracy score is also at 0.73, and thus no overfitting.

One might speculate to whether the use of decision stumps, which have virtually no overfitting in combination to form an ensemble method is the key factor in the low overfitting which can be seen in this method.

XG Boost

In figure 6 we can see the behavior of the XG boost method as a function of increasing learning rate. This method behaves a bit differently from the others, as the test accuracy is seemingly constant, whilst train accuracy increases. Despite this, it achieves about the same accuracy as the previous methods.

One upside this method provides is the ability to parallelize the method, meaning it will run faster than sequential ensemble methods.

V. CONCLUSION

As theorized, the reduction and data preprocessing both reduced the number of unique words, allowing us to work on a large subset of the data at a time, whilst also reducing overfitting and increasing accuracy. This means that the preprocessing worked as intended.

Initially we wished to look at how combining weak learners could provide a better result than the weak learner itself, as is the basis for ensemble methods in the first place. We did indeed find decision trees to be a weak learner, as a result of 71 % was the best we could achieve, which also lead to a massive amount of overfitting. The attractive factor of decision trees, which was its simplicity, also seemed to be its weak point, as the dataset was far too complex for such a weak learner.

The result of ensemble methods were as theorized, all better, but only slightly so. For almost all the different ensemble methods we reached around 75 %, but with varying amounts of overfitting. The parallel ensemble methods all have a speed advantage, as these methods are capable of being parallelized.

Despite using a variety of methods, we fail to surpass a mediocre score of 75 %. This we believe is partly due to the fact that the data is labeled using a fairly simple algorithm. This in combination with the fact that decision trees and ensemble methods have a hard time picking up on the intricacies that such a complex dataset requires, especially after we remove many of the words and remove the order and context of them.

As we conclude that the use of bag of words, which reduced a coherent sentence to keywords is a key factor in our low accuracy score, we recommend looking into alternative methods, which better conserve the intricacies which words in context bring, such that deep learners can pick up on these more easily.
During the research for this paper we had little focus on computational efficiency, meaning we were limited to datasets of a few thousand tweets at a time. This is also something which could further increase the results, and is something to consider for future research.

Appendix A: Code for reproduction

For code used in this project see https://github.com/elinfi/FYS-STK4155/tree/master/Project3 or https://github.com/andebraa/fys-stk4155/tree/new_branch/project3. The two github repositories are very similar, and feature much of the same data.

[1] A. Geron, *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, 2017.

[2] M. Hjorth-Jensen, "Week 45: Random forests and boosting." https://compphysics.github.io/MachineLearning/doc/pub/week45/html/week45.html, 2020. Accessed: 07-12-2020.

[3] J. Nothman, H. Qin, and R. Yurchak, "Stop word lists in free open-source software packages," in *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, (Melbourne, Australia), pp. 7–12, Association for Computational Linguistics, July 2018.

[4] A. Go, R. Bhayani, and L. Huang, "Twitter sentiment classification using distant supervision," *Processing*, pp. 1–6, 2009.

[5] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020.

[6] Wikipedia, "List of emoticons — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=List%20of%20emoticons&oldid=991538808, 2020. [Online; accessed 09-December-2020].

[7] E. Loper and S. Bird, "Nltk: The natural language toolkit," *CoRR*, vol. cs.CL/0205028, 2002.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[9] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 785–794, ACM, 2016.

[10] J. Zhu, H. Zou, S. Rosset, and T. Hastie, "Multi-class adaboost," 2009.