Documentation - ML Pipeline - Team One

Introduction

As part of the Seminar "Machine learning in practice" taught by Lucas Bechberger in the fall of 2021 at the Institute for Cognitive Science, University of Osnabrück, we implement an exemplary machine learning pipeline from preprocessing to the deployment of the application, testing, and trying out different preprocessing, feature extraction, dimensionality reduction, classification and evaluation methods.

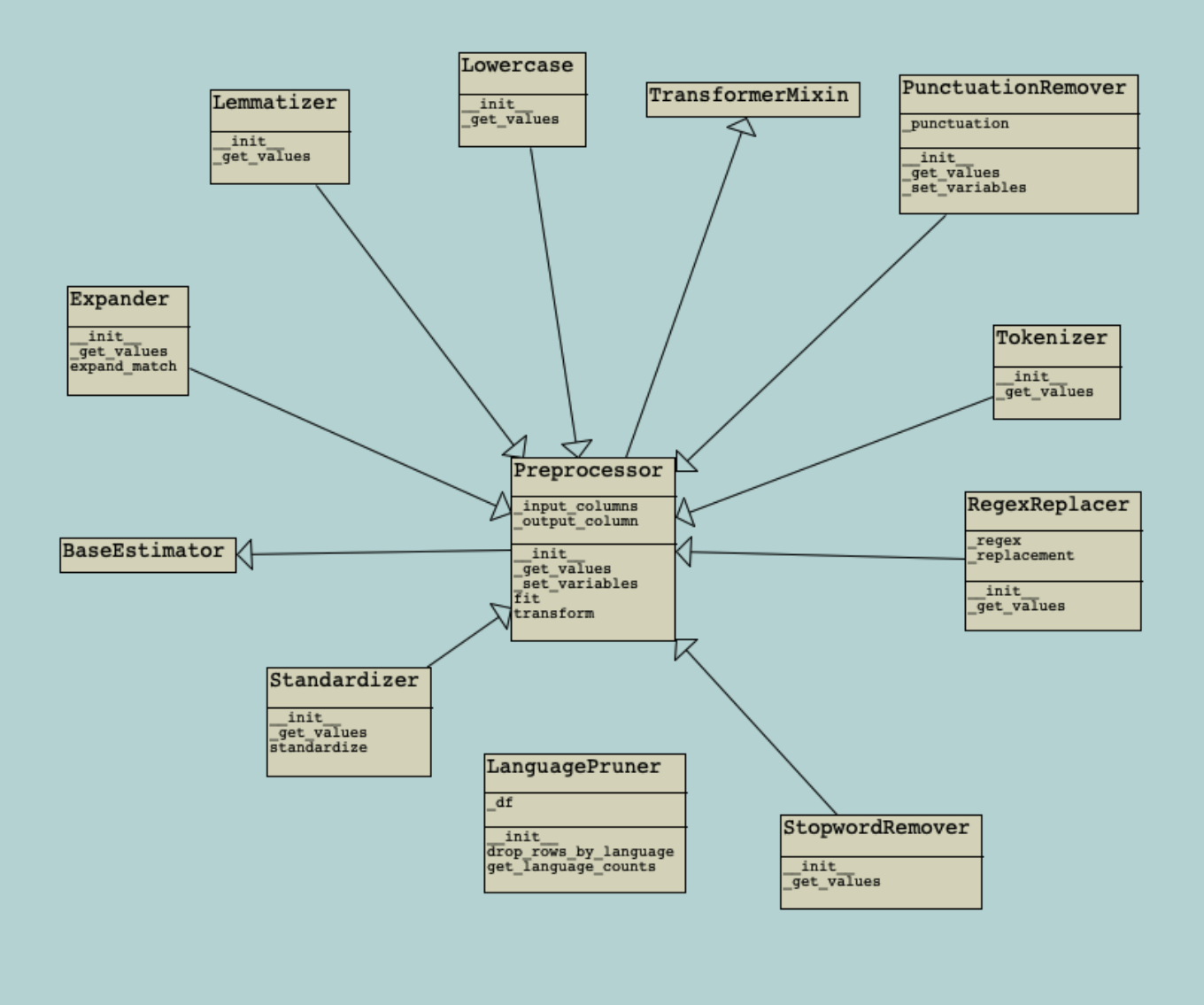
Research Hypothesis

The goal of our example project is to predict, given a tweet, whether it will go viral, i.e., receive many likes and retweets. The virality criterion is defined by the sum of likes and retweets, where the threshold is specifiable by the user, but defaults to 50.

Data Collection

As a data source, we use the "Data Science Tweets 2010-2021" data set (version 3) by Ruchi Bhatia from Kaggle, which contains tweets resulting from the searches of "data science", "data analysis" and "data visualization" authored between 2010 and 2021.

Preprocessing



1. Remove Non-English Tweets

Returns a dataset containing only tweets marked as being of the target language.

Goal

Assuming the tweet itself, not just its metadata is supposed to be a feature, language consistency is important. Removing tweets in a language other than the target language, (here: English), reduces the complexity of pre-processing steps, as these can now be performed on useable tweets only.

Implementation Process

The Dataset contains a column *language*, tagging each tweet with its (presumed) language. The LanguagePruner Class drops rows from the dataset with a *language* tag other than the target language. In addition, we implemented a helper function within that class, which retrieves the tweet count of each language in the dataset and stores the output in `data/preprocessing/language_counts.csv`. This output

determines the ratio of languages in the set and can help decide whether there is enough available data in the target language.

Discussion

While discarding data in a foreign language speeds up the preprocessing step by only focusing on useable data, some issues may arise:

- There are a few inconsistencies between the actual language of a tweet and the tag in the *language* column. The tag might be retrieved from a user's language setting, i.e. Italian, while the same user has incidentally tweeted in English. So removing non-target language tweets may result in some foreign language tweets being overlooked by `LanguagePruner`, while some target-language tweets are removed as well.
- If there are not enough tweets in the target language, removal of foreign-language tweets results in too little available data. In this case, textual features, such as TF-IDF, can't be used and the removal of foreign-language tweets should be skipped, while metadata features should be prioritized.

2. URL removal

Returns a column containing the tweet's content without URLs.

Goal

If a tweet in our dataset contains URLs, they have to be removed so they do not influence feature extraction later on which is based on the English language. Especially features like named-entity-recognition and sentiment-analysis are more efficient if the tweet does not contain URLs.

Implementation process

We used a regular expression in run_preprocessing.py to filter for URLs in the tweet and remove them. This uses the class RegexReplacer in regex_replacer.py.

Discussion

According to what was argued earlier, removing URLs for the sake of better feature extraction seems perfectly reasonable. But is the feature extraction better? If we strictly remove all URLs, we potentially lose important information: Does the tweet link to a site? If yes, which site is it, and is this a significant contribution to the virality of the tweet? Luckily, the dataset already provides a "URL" column containing the exact URLs to which a tweet refers. If we want to implement features that screen URLs or their contents we are still able to do so!

3. Lowercase

Goal

We lowercase all tweet texts to be able to reliably match different capitalizations of the same word in downstream preprocessors as well as the classifier.

Sometimes, however, capitalization is used to distinguish different concepts:

```
"We saw a worsening in US relations to China." -> "we saw a worsening in
us relations to china."
```

Implementation Process

We implemented a Lowercase class that uses python's lowercase functionality.

Discussion

This step is generally useful for natural language processing and does not usually result in the loss of task-specific features, e.g. for acronyms that spell out an existing normal word.

4. Expand Contractions

Expands contracted words into their long-forms.

Goal

Contractions add redundancy, as they technically are separate tokens, even though their components overlap with other tokens. They also usually don't carry much semantic value, since their components are usually stop words. Expanding them to their respective long-form removes this redundancy and assists the stopword removal process that occurs at a later point in the pipeline.

Example ("" denote tokens)

- *isn't --> is not --> "is" "not"* instead of *isn't --> "isn't"*

Implementation Process

The contractions are expanded by the Expander, while the contraction mapping can be found here. The Implementation stems from towardsdatascience.com and uses a list mapping contractions to their respective long forms.

Discussion

Adding this preprocessing step is not necessarily crucial to the preprocessing and one might argue that removing it may speed up the pipeline. However, it is a simple way to minimize the vocabulary of our dataset by avoiding unnecessary duplicate tokens and ensuring the fidelity of our model to semantics. To clarify, tokens with the same semantics should be classified as one item in a vocabulary, no matter if they are contracted or not.

5. Punctuation

Removes all kinds of punctuation from the tweet

Goal

We want to lay the focus on word semantics in our classification and match words at a different position within a sentence. Punctuation of any form can be disturbing. Hashtag characters are also removed, conveniently allowing us to treat hashtags as normal words.

Implementation Process

We implemented PunctuationRemover which uses a string replacement function to remove punctuation.

Discussion

Removing punctuation may result in loss of context in cases where punctuation is used to convey emotional content (as with exclamation marks) or vital sentence meaning (as with question marks and commas). However, we believe that punctuation in the context of tweets only marginally influences meaning as many tweeters omit punctuation anyway and the character limit of 240 generally leads to less complex grammatical structure.

6. Standardize Spelling Variations

Standardizes UK spelling variations to their US equivalent.

Goal

Spelling variations arise due to misspellings or location-based differences. Different spellings for the same word add redundancy to our features, as they are counted as different vocabulary, even though their semantics are the same. Changing variations of words, in our case location-based differences, to a standard spelling ensures that semantic information for the words is kept and that they can be further dealt with as the same word.

Implementation Process

The tweets are standardized by the Standardizer class, while the spellings mapping can be found here. The Implementation is in line with the implementation for the expansion of contractions, as seen above, and uses as

mapping of UK spellings to their respective US spellings. We treat the US spellings as the standard vocabulary and change any UK variations to the standard US spelling.

Discussion

Adding this preprocessing step is not necessarily crucial to the preprocessing and one might argue that removing it may speed up the pipeline. However, it is a simple way to minimize the vocabulary of our dataset by avoiding unnecessary duplicate tokens and ensuring the fidelity of our model to semantics. To clarify, tokens with the same semantics should be classified as one item in a vocabulary, no matter if they are contracted or not.

7. Tokenizer

Splits the tweet string into word tokens.

Goal

To ease feature extraction from tweet texts we split them at word boundaries. We split the tweet texts into lists by word boundaries to be able to count them and run statistical feature extraction more easily.

Implementation Process

The tweet texts are tokenized using NLTK (Natural Langauge Toolkit) in the Tokenizer class.

Discussion

This step is generally necessary to process natural language and aids in the classification of the tweets.

8. Remove numbers

Removes any numerical values

Goal

Number expressions have a high variance without much meaningful semantic difference. To improve classification, we decided to replace number expressions in tweets with a generic token.

Implementation Process

Numbers are replaced using a regular expression in the RegexReplacer class.

Discussion

Replacing numbers with a generic token has the advantage of removing unnecessary noise from the dataset to aid in classification, assuming that individual number expressions are irrelevant to the task. Since the dataset specifically encompasses tweets related to data science, there is a chance that tweeters will use numbers more

frequently and that numbers have a higher significance to the tweet message, but we believe that the specific value of the number expression does not influence virality.

9. Lemmatization

Reduces a word to its underlying lemma

Goal

Our goal was to generalize the form of words as far as possible while retaining their meaning. This enhances comparability between tweets and is a useful precondition for stopword removal.

Implementation process

We created the class Lemmatizer which accesses the WordNetLemmatizer from nltk and used part-of-speech tags to replace the words in the tweet with their lemmas effectively.

Discussion

We also thought about using Stemming instead of Lemmatization. NLTK stemming is more straightforward to implement and probably has a slightly better runtime. However we still quickly decided on the lemmatizer because its accuracy is significantly better.

10. Stopword removal

Remove very common words from the tweet

Goal

Our goal was to get rid of very common English words which can not be used for meaningful features. If we left them in, they would probably not contribute to the quality of our classifier. In addition, this decreases runtime because it makes our dataset smaller and is the perfect preparation for our TF-IDF feature.

Implementation process

We created the class StopwordRemover which accesses a corpus from NLTK containing English stopwords. Every word contained in this corpus and some additional meaningless symbols (specified manually by us) are filtered out.

Discussion

A significant downside of this preprocessing step is that it can negatively influence our sentiment analyzer by distorting context - depending on which words are filtered and which are not.

Example

Original tweet:

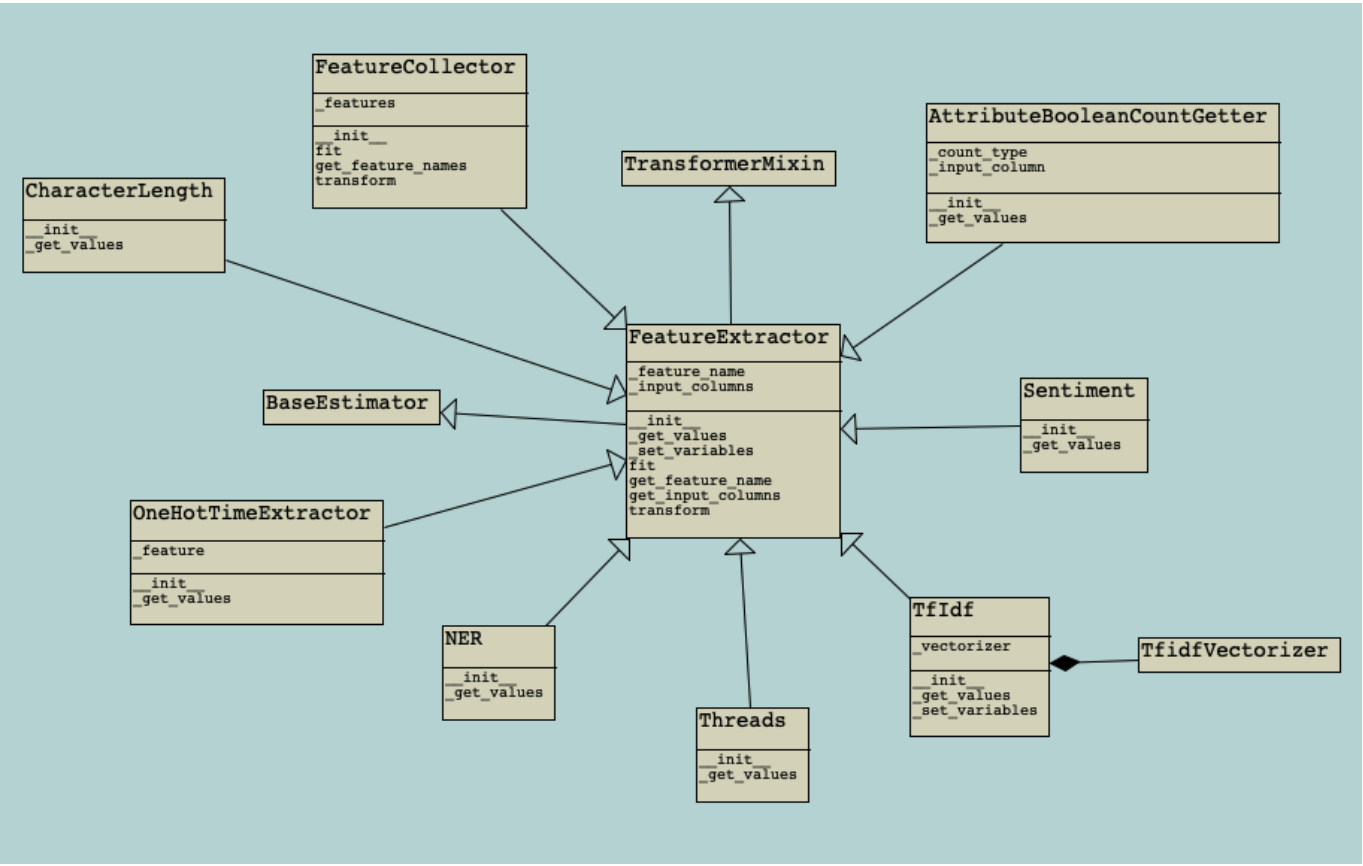
```
"I really don't love Barack Obama."
```

Tweet after stopword removal:

```
"really love Barack Obama."
```

While being aware of this issue, we still left the preprocessing step in our pipeline because of the reasons mentioned in the "Goal"-section.

Feature Extraction



Character Length

The class CharacterLength counts the number of characters used in a given tweet. Perhaps a tweet's virality somewhat depends on its length.

DateTime Transformations

Extracts *weekday*, *month*, *time of day* and *season* features from the creation time of a tweet.

Goal

In the given dataset, the columns *date* and *time* contain information regarding the time the tweet was created. What they lack, are meaningful categorical subsets of time. The time of tweet creation may be a valuable feature for our model, but exact dates and seconds probably don't carry too much relevant information. Instead, we have transformed the data to include the month, weekday, season, and time of day that the tweet was created. These features might hold more information as to the chance of a tweet becoming viral and are categorical, thus can be used by any classifier.

Example:

Let's say there are 2 viral tweets in our dataset. The first was created on Sat. Dec. 5 2020 at 18:45:22 and the other on Sun. Jan 10 2021, at 21:37:02. While it could be the case that tweets posted at exactly 18:45:22 or on the 10th of January of a year have a higher likelihood of going viral, we are assuming that these data points more likely indicate that tweets posted in winter, during the evening, or on the weekends may have a higher chance of going viral.

Implementation Process

The class OneHotTimeExtractor receives a feature name as an input and retrieves the relevant column (either *date* or *time* from the dataset, converts it into Datetime format and transforms the times into categorical data in the following ways:

- **Weekday**: Using `datetime.weekday()` function, the feature is returned as an int between $0-6$
- **Month**: The month of the posting is retrieved and the feature is returned as an int between $1-12$
- **Season**: According to the month, the feature is returned as an int between $0-3$ denoting one of four seasons
- **Daytime**: According to the hour of the posting, the feature is returned as an int between $0-3$ denoting the time of day.

Finally, the categorical features are converted into one-hot vectors using sklearn's OneHotEncoder to avoid the model learning an ordinal relation between the integer categories.

Discussion

Transforming the *date* and *time* attributes into categorical one-hot vectors untangles the metadata of the time of post into more meaningful units.

Retrieve Attribute Counts or 'exists' Boolean

Counts the amount items used for a given attribute or returns a $0/1$ boolean feature, whether the tweet fulfills or uses a certain attribute.

Goal

A tweet's virality may be influenced by the amount of an attribute it contains or simply by if the tweet has a certain attribute. The attributes or columns in question here are *photos*, *mentions*, *hashtags*, *reply_to*, *urls*, *video*, and *retweet*. We assume that at least a few of those attributes influence the likelihood of a tweet going viral, by either simply being included or not, how many of the attributes were included or if the tweet is of a certain type, as would be the case for the *video* and *retweet* columns.

Implementation Process

The class BooleanCounter receives one of the above-mentioned columns as input, as well as the desired feature type *count* or *boolean*. As the columns contain stringified lists, the `BooleanCounter` interprets them as the list datatype and determines either the length of the list or whether or not the list is empty. In the case that the input column is either *video* or *retweet*, where the goal is merely to output a boolean feature on if the tweet is of the type or not, `BooleanCounter` standardizes the *True/False* and $0.0/1.0$ formats of the columns into a $0/1$ boolean feature.

Discussion

Using counts or boolean values is a simple way to represent the above-mentioned attributes without conflating their information value. We assume that the amount of hashtags used in a tweet is a better indicator of the likelihood of virality than the exact hashtags that were used, as we have no additional information on the popularity or reach of a certain hashtag, only that the hashtag was used. If we had access to more information about the above attributes, such as what an attached photo depicts, current events, etc., then there is an alternative feature option to the pure count or 'exists'- boolean of an attribute. However, with our dataset and prediction goal, a *count* or *boolean* feature is likely to carry the most information in regards to our hypothesis.

Sentiment Analysis

Extracts *negativity*, *positivity*, and *neutrality* scores for the tweet texts.

Goal

Our hypothesis is that emotionality and subjectivity of tweet content influences virality. We, therefore, chose to employ a sentiment analyzer to extract positive and negative sentiment from the tweets.

Implementation Process

We use the VADER sentiment analyzer to extract positive, negative, and neutral sentiment scores that range from 0 to 1, as well as a compound value that ranges from -1 to 1. This analyzer is used in the Sentiment

feature extractor class, which thus adds four output dimensions to the overall feature vector.

Discussion

The sentiment is often cited as one of the driving forces of content in social networks. We thus believe that it also plays a role in predicting tweet virality. However, the method of sentiment analysis used by the VADER project does not take into account sentence-level semantics but merely word-level semantics. Specifically, it uses precalculated scores for the words it finds in the tweet to calculate average sentiment scores for the whole text. This is a rather naive approach, but we believe it to be a worthwhile tradeoff between added value and performance.

TF-IDF

Extracts "novelty" scores for the 200 most frequent words (across all tweets).

Goal

To provide the classifier with words that are relevant for classification, we use a TF-IDF approach, which calculates the term frequency divided by the inverse document frequency.

Implementation Process

We use the TfIdfVectorizer from scikit-learn to calculate TF-IDF scores for the top 200 words that appear in the dataset after removing all stopwords.

Discussion

By novel words in tweets, we allow the classifier to infer the influence of specific words on tweet virality. Removing stop words in the preprocessing step helps weed out fill and function words from closed classes that do not carry much semantic relevance. However, restricting our TF-IDF vocabulary to 200 words may limit classification performance as it is not clear that only popular words (and combinations thereof) influence virality.

Thread detection

Extract a boolean feature for whether tweets are part of a "thread".

Goal

Tweeters can group multiple tweets using an informal mechanism called threads. To help classify the virality of tweets we detect whether a tweet is part of a thread.

Implementation Process

We use a simple Regular Expression to match the thread emoji, as well as number expressions like $1/or1/4$ at the beginning or end of the tweet.

Discussion

We considered matching the word 'thread' as well but decided against it since many tweets that merely respond to threads also mention the word thread. Threads may either affect virality positively because tweeters can post more content in one go, or negatively. After all, the audience is not patient enough to read the whole thread. One way or another, we believe threads are an important attribute in characterizing a tweet and may thus be equally important in influencing virality.

Named Entity Recognition

Counts the occurrences of SpaCy named entities

Goal

A critical factor of a tweet's virality could be the number of referrals to a person, organization, country, etc. within the tweet. Therefore we wanted to feed our classifier this information.

Implementation Process

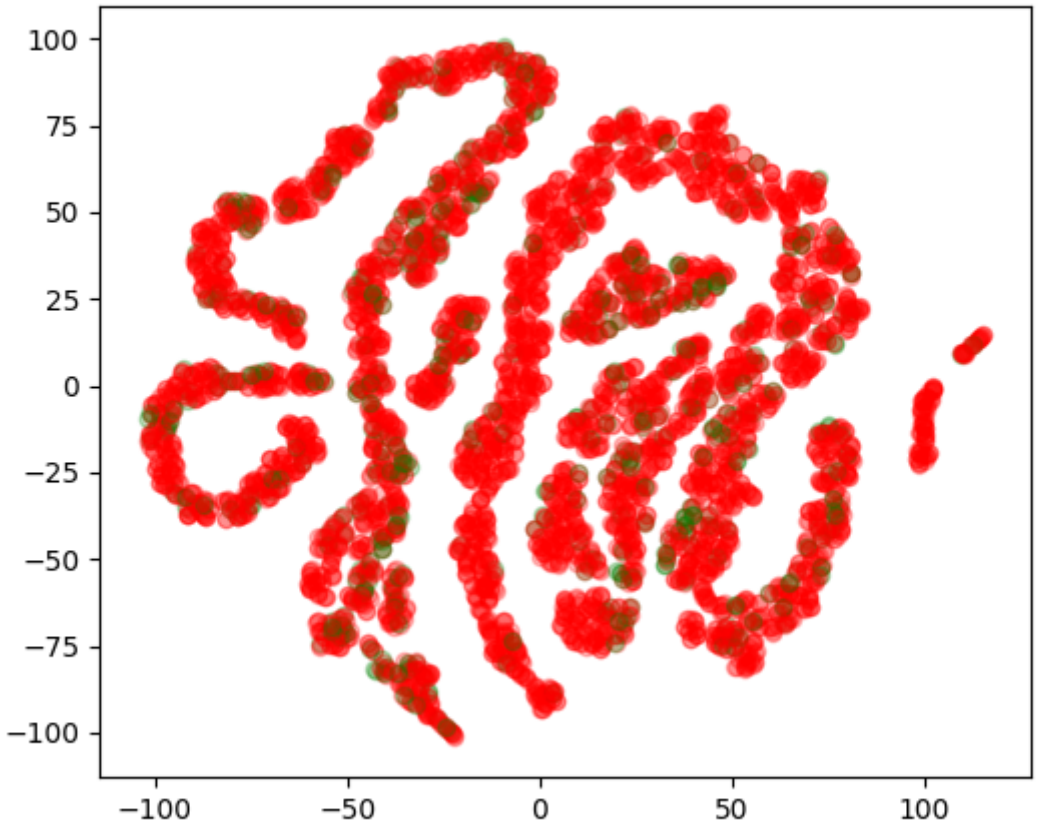
We implemented the class NER which computes the entity type of each word in a tweet using SpaCy's Entity Recognizer and a trained English pipeline. In a preliminary implementation, we used NLTK for this but it recognized entities much worse so we switched to SpaCy. After this, the entities are counted, the counts are appended in an array and all the arrays are fed to our classifier.

Discussion

During testing, we noticed a major shortcoming of this feature: Most of the time it only recognizes persons and money as entities. This means that the feature is unfortunately not as useful as we initially expected. We decided to use it nevertheless because the recognition of just these entities can also help with classification.

Dimensionality Reduction

After feature extraction, we experimentally visualized our feature space using t-distributed stochastic neighbor embedding (t-SNE), which stochastically embeds the complete feature space in a two-dimensional space for visualization purposes (positive samples are green, negative ones are red). The result is not very encouraging in terms of being able to learn a classification from the feature space, however, t-SNE mainly accounts for variance and does not take into account the informativeness of the individual dimensions concerning the task. Unsurprisingly, our attempts to use dimensionality reduction techniques, which also mostly take into account variance, to improve the classifier were mostly futile.



Mutual Information (MI)

Mututal Information measures the amount of information obtained about one variable when considering another.

Principal Component Analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower-dimensional space.

Goal

We wanted to remove correlated features that do not help our classification while minimizing the risk of overfitting.

Implementation Process

We used the PCA class to project the features onto k many dimensions, while k can be specified in the CLI by the user. Additionally, we iterated through the components of this projected feature space. If --verbose is set we print useful user information indicating the composition of the feature space in percent.

Discussion

A possible shortcoming of PCA could be unstandardized data. Luckily we take care of that in our preprocessing (See standardize.py). In addition we have to be very careful about our selection of k such that we don't lose too much information. We figured that k=20 would lead to the best results.

Truncated Singular Value Decomposition (TSVD)

TSVD as implemented by SciKit learn, while ultimately realizing principal component analysis as well, might be a faster algorithm than the one implemented by the PCA class, especially for large sparse data sets. In our case, it affected neither runtime nor classification performance.

Machine Learning Model

The machine learning model is the central piece of our pipeline that has to learn a concept from the features we extracted. We chose to implement and compare the following models.

Majority Vote Classifier

The Majority Vote Classifier classifies an instance based on the majority class in the training set.

Example

If a dataset has a class distribution of $y_1 = .6$, $y_2 = .3$ and $y_3 = .1$, the classifier will classify any and all instances as belonging to class y_1.

Discussion

Compared to the Label-Frequency Classifier, the Majority Vote Classifier performs much better, reaching around 10 more accuracy on both the training and validation set. However, both F_1 and Cohen's Kappa are 0, not a useful baseline for these metrics. Still, due to the higher accuracy, the Majority Vote Classifier makes for a better baseline to compare our model to.

Evaluation Results:

Metric	Training	Validation
Acc	0.8765	0.8768
Kappa	0.0000	0.0000
F_1	0.0000	0.0000
Balanced	0.5000	0.5000

Label-Frequency Classifier

Classifies an instance according to the distribution of labels in the training set.

Goal

The label-frequency classifier serves as an alternative baseline to the majority vote classifier.

Implementation Process

We use sklearn's DummyClassifier on the 'stratified' strategy in run_classifier.py.

Discussion

Compared to the Majority Vote Classifier, the Label-Frequency Classifier performs poorly, reaching around 10 less accuracy on both the training and validation set. And, while the F_1 is > 0, which would at least add some sort of baseline for this metric, Cohen's Kappa is negative, showing disagreement. Thus, the Majority Vote Classifier makes for a better baseline to compare our model to.

Evaluation Results:

Metric	Training	Validation
Acc	0.7853	0.7856
Kappa	0.0005	-0.0053
F_1	0.1228	0.1166
Balanced	0.5001	0.5028

SVM Classifier

Support Vector Machines have historically been very successful in binary classification tasks and were usually perceived as the silver bullet until the dawn of Neural Networks.

Goal

We ran our data set through an SVM classifier as we believed it would perform best, together with the Multi-Layer Perceptron.

Implementation Process

We employed SciKit learn's LinearSVC implementation in run_classifier.py.

Discussion

Results after hyperparameter optimization reveal that training using this classifier went comparatively well and did neither overfit nor underfit. Nonetheless, a balanced accuracy of 0.7 is still rather underwhelming.

Metric	Training	Validation
Acc	0.7181	0.7178
Kappa	0.2042	0.2003
F_1	0.3200	0.3165
Balanced	0.7120	0.7071

K Nearest Neighbors Classifier

Classifies an instance based on the majority label of the k data points closest in the feature space.

Goal

We chose KNN as an additional classifier out of interest, after it having been mentioned in prior courses.

Implementation Process

We use sklearn's KNeighbors Classifier in run_classifier.py. Between the different algorithms (*Auto, Ball Tree, KD Tree, and Brute*) to choose from, *Auto* achieved the overall best results on the training and validation set. We tested both *uniform* and *distance* weights, with different k s, and settled on *distance* weights with $k = 10$, which yielded the best results.

Extract from Hyperparameter Optimization Evaluation Results*:

Algorithm	Metric	Training	Validation
BallTree	Acc	0.8060	0.7982
	Kappa	0.1544	0.1128
	F_1	0.2657	0.2284
KD Tree	Acc	0.8064	0.7998
	Kappa	0.1523	0.1151
	F_1	0.2653	0.2298
Auto uni, 5	Acc	0.8824	0.8684
	Kappa	0.1623	0.0587
	F_1	0.1940	0.0939
Auto d, 10	Acc	0.9338	0.8553
	Kappa	0.6077	0.0434
	F_1	0.6394	0.0980

* performed on a subset of the dataset for efficiency

Discussion

Metric	Training	Validation
Acc	0.9928	0.9020
Balanced Acc	0.9661	0.5702
F_1	0.9605	0.2367
Kappa	0.9565	0.1963

KNN yields different results, depending on if sparse or dense features are used. Thus, adding LaPlace Smoothening and/or using OneHotEncoder's 'dense' parameter might have improved the performance. Overall, KNN performs very poorly in comparison to the baseline classifiers and is highly overfitted to the training set, with the final hyperparameter configuration only achieving 57 balanced accuracy, $.23$ F_1 score, and a mere $.20$ Cohen's Kappa on the validation set.

Multi-Layered Perceptron Classifier

Optimizes the log-loss function using the stochastic gradient descent or L-BFGS.

Goal

Due to the enhanced sophistication of MLP compared to KNN or SVM, we thought the quality of classification to be significantly better.

Implementation Process

We used the class MLPClassifier which takes multiple hyperparameters. These can be specified in the CLI. If none are specified, the parameters that performed best during testing are selected. We optimized the hyperparameters using grid_search.sh and mlflow. Tested hyperparameters include hidden_layer_sizes, activation, solver, and max_fun. The last did not make an impact on classification quality at all.

Discussion

To our surprise, MLP performed slightly worse than SVM. Especially the balanced accuracy is significantly lower (Δ 0.1390 for the validation set) than the balanced accuracy of SVM.

Best Results from Hyperparameter Optimization:

Metric	Training	Validation
Acc	0.8910	0.8936
Kappa	0.1757	0.1951
F_1	0.2273	0.2455
Balanced	0.5681	0.5756

We specified the hyperparameters leading to this result as default values for classification with MLP in run_classifier.py.

Evaluation

Evaluation metrics inform decisions about the quality of a classifier. We chose to implement the following metrics for this task.

Accuracy

Accuracy is a very basic measure of binary classification performance, which is not very reliable in the face of imbalanced data (a naive always-false classifier would yield an accuracy of $.9$ on our dataset). We added this metric anyway for sake of completeness and out of interest for how the different metrics differ in measuring our results.

Balanced Accuracy

Balanced accuracy, while similar to normal accuracy, yields an informative metric even for imbalanced data. It results from taking the mean of true positive rate and true negative rate. As we are dealing with a heavily imbalanced data set balanced accuracy is a useful metric for assessing model performance.

Informedness

Informedness, also known as *Youden's J statistic* or *Youden's index*, measures the probability of an informed decision (as opposed to a random guess) and is not skewed by imbalanced data such as the current data set.

Cohen's Kappa

Cohen's Kappa measures inter-rater reliability between the ground truth and the classifier in a range from -1 to 1, where 1 means complete agreement. However, it is overly conservative when applied to imbalanced data. In

our case of a 90 to 10 distribution, it would be impossible to achieve a maximum value of Kappa, so it has to be taken with a grain of salt.

F_1 Score

The F_1 *score* combines precision and recall scores into a singular value between 0 and 1 according to the formula: $2 * \frac{precision * recall}{precision + recall}$. A score of 0 can be interpreted as the model failing at its task, while a score of 1 indicates that the model classifies each instance perfectly. The F_1 Score tends to be unreliable in the face of imbalanced data sets as it does not take into account the false-negative rate.

MCC

Matthew's Correlation Coefficient is mathematically similar to *Cohen's Kappa* and measures the correlation between ground truth and model predictions on a range between -1 and 1. While the measure is robust to imbalanced data sets, absolute values can only be compared when measured on the same data set.

Conclusion

Our best setup turns out to be Support Vector Machines without dimensionality reduction. This may be because PCA does not take labels into account when selecting the best dimensions. Additionally, the quality of classification in this task probably depends on a multitude of minor features.

Running our best classifier on the split-off test data set achieves the following evaluation scores:

Metric	**Values**
Acc	0.7155
Kappa	0.1984
F_1	0.3151
Balanced Acc	0.7064
Informedness	0.4128
MCC	0.2582

This result indicates a good, even if not outstanding, performance of the classifier in generalizing the phenomenon of tweet virality from our training and validation sets to previously unseen data. While the raw accuracy score is lower than on our baseline model (0.7853S), Cohen's Kappa is markedly higher (Baseline: 0.0005). This is likely due to the imbalanced nature of our data set.

Given the features, we extracted from the data and the model we selected, it appears to be possible to classify tweets as viral to some extent, even if we would have hoped for better accuracy.