Capstone Project

Machine Learning Engineer Nanodegree

Luis Bronchal
September 11th, 2017

Sentiment analysis in airline tweets

I. Definition

Project Overview

Millions of people express their opinion, preferences, likes, dislikes and complaints nowadays through social networks. That means that a lot of useful data is available and organisations like companies, institutions and governments can take advantage of this. Companies, for example, can get a powerful feedback, but for this to be a reality, information and knowledge must be extracted from data. This must be done as fast as possible in order organisations be able to react on time: for example, in social networks a complaint can become viral and produce a great damage if it is not kept under control.

To monitor social networks like Twitter can provide a significant advantage for companies helping them to have a better knowledge of their clients and their market. Humans are good understanding raw messages and extracting information and actionable knowledge from them, but the volume of data is so big and is generated so fast that it would be a very slow and inefficient process. In order organisations be able to react on time it is necessary to have an automatized process. Here Artificial Intelligence, and Machine Learning particularly, can be very useful.

Problem Statement

To illustrate how Machine Learning can help to obtain quick knowledge from Social Networks, we are going to apply it to a particular case: we are going to use a set of tweets where U.S. airlines are mentioned, and we are going to build a Machine Learning model able to interpret if a tweet has a negative sentiment associated. This model will be helpful to do sentiment analysis of future tweets in the same domain providing a quantifiable measurement of people consideration about each airline brand and its competitors.

We are going to use the Kaggle dataset "Twitter US Airline Sentiment" which was generated by Crowdflower. This dataset contains tweets with mention to 5 of the major U.S. airlines and was collected in February of 2015. Each tweet has been manually labelled indicating if it expresses a positive, a negative or neutral sentiment.

Metrics

This is a classification problem where we are going to identify two possible classes (negative or non-negative sentiment). The data is a little bit unbalanced with more negative labels than non-negative ones so, accuracy is not going to be a good metric here. We are going to calculate it but also AUC and F1, which are going to be better for the context of the problem.

II. Analysis

Data Exploration

The dataset contains the following fields which are relevant for our project:

- airline_sentiment: sentiment of the tweet [‘negative’, ‘positive’ or ‘neutral’] (String)
- airline_sentiment_confidence: it express how confident we are the labeled sentiment is right. It is a real number between 0 and 1 (Numeric)
- text: text of the tweet (String)

The original dataset contains 14640 rows and it is a bit skewed, because people tend to express more negative opinions (airline_sentiment): about 63% are negative, 21% neutral and 16% positive.

The mean tweet length is 104.2435 characters with a standard deviation of 35.9762 characters.

Removing the stopwords, the mean number of words (tokens) per tweet is 20.2053 with a standard deviation of 7.6421

Here we can see which are the 20 most common words per tweet. We have distinguished the case of tweets with negative sentiment and tweets with non-negative sentiment associated (neutral or positive sentiment):

NON-NEGATIVE TWEETS	
TOKEN	FREQUENCY
@jetblue	1149
@southwestair	1114
@united	1042
flight	890
@americanair	766
thanks	679
@usairways	585
thank	481
get	305
@virginamerica	299
great	243
…	232
help	224
please	213
service	197
flights	193
us	176
time	162
i'm	160
need	155

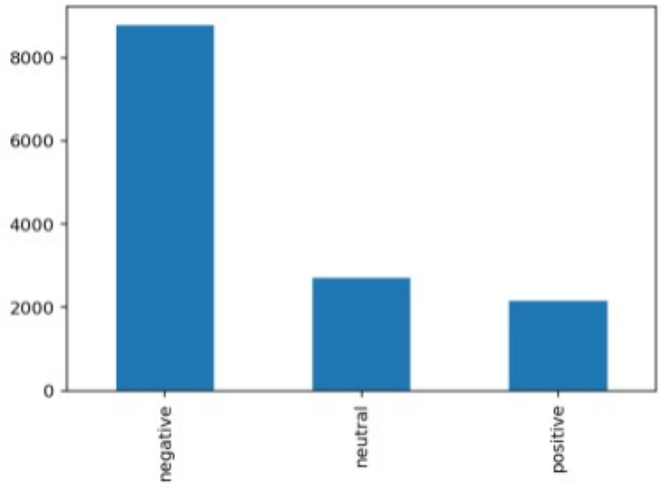
NEGATIVE TWEETS	
TOKEN	FREQUENCY
flight	2799
@united	2580
@usairways	2260
@americanair	2041
@southwestair	1143
get	940
@jetblue	905
cancelled	879
service	731
…	709
hours	641
hold	600
customer	592
time	573
help	568
2	552
plane	509
delayed	489
still	473
i'm	473

In the case of tweets with negative sentiment, some of the most common words make perfect sense: “cancelled”, “delayed”, “hold”, “still”,… In the case of non-negative tweets some top words look like logical: “thanks”, “thank”, “great”,… In both cases, the twitter username of the airlines appears as top words (that’s because they are always mentioned in the tweets)

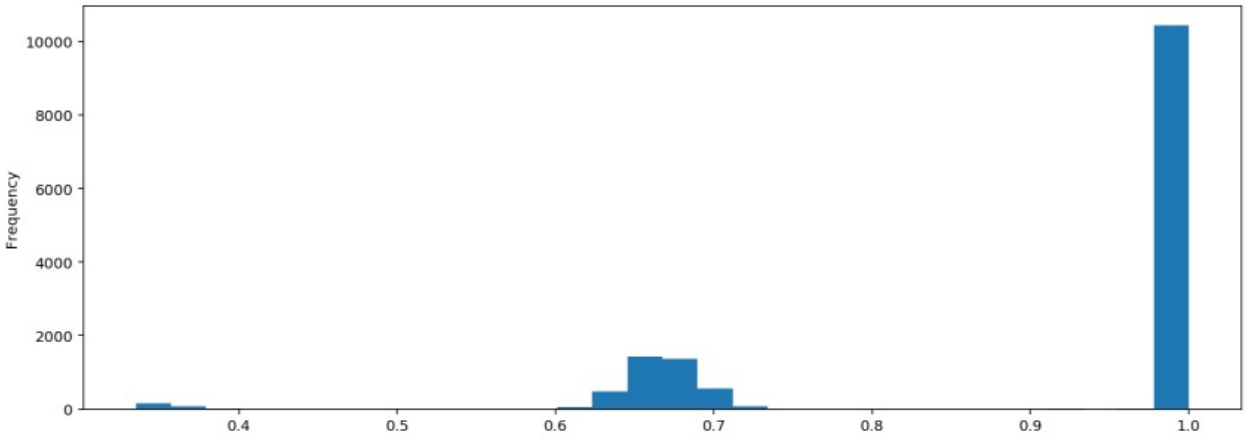
The mean number of mentions (inclusion of twitter usernames) in tweets with negative sentiment is: 1.1108 (+/- 0.3655) which is a little bigger than the same metric for tweets with non-negative sentiment: 1.0933 (+/- 0.3810)

Exploratory Visualization

The following plot shows the proportion of tweets of each class in the original dataset without any preprocessing:

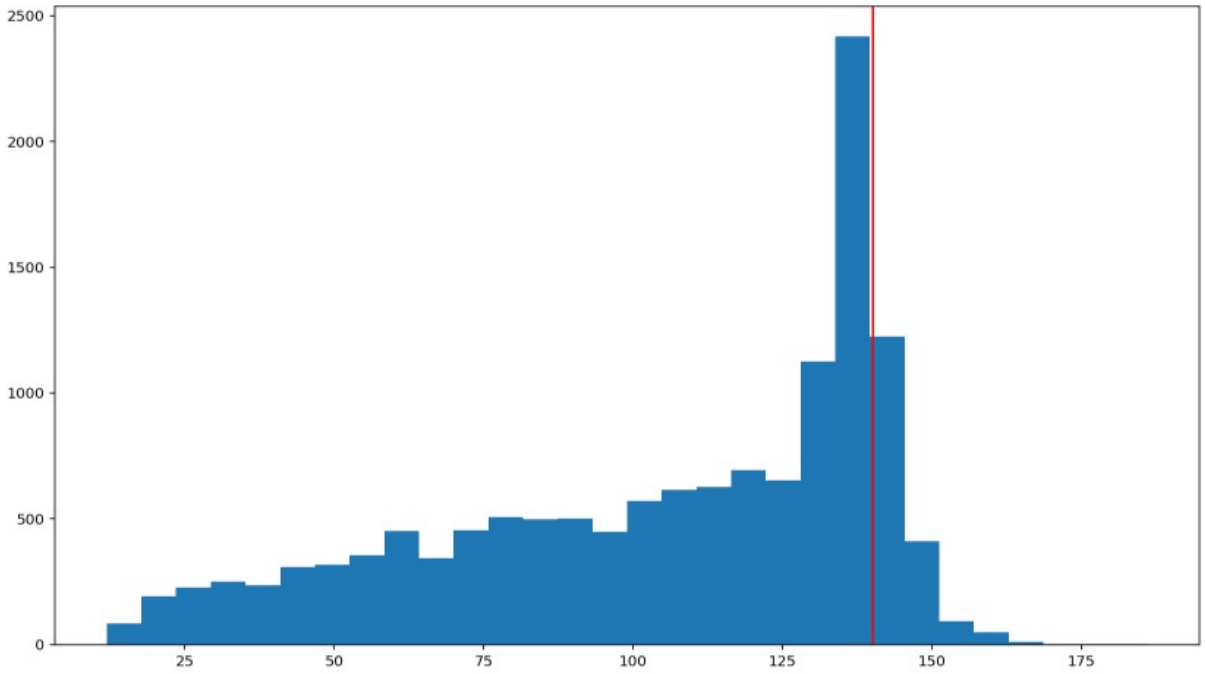


Each tweet has been hand labeled and has associated a confidence level, which indicates how reliable its labeling is. Here we can see the distribution of that variable (airline_sentiment_confidence):



We can see that this distribution has a big left skew (its mean of 0.900169 is lower than its median, which is 1.0). That means that, fortunately, most of the tweets have good labeling. In the preprocessing step (explained later) we will remove the less reliable ones.

The next plot is a histogram with information about tweets length. It shows that unexpectedly there were tweets with more than 140 characters (with in theory is impossible). In the preprocessing step we tackled this problem.



Algorithms and Techniques

We are going to try different supervised models to predict if a tweet has associated a negative sentiment. Particularly we will try some "classic" models like Logistic Regression, SVM, Naive Bayes and more "modern" ones like Random Forest, xgboost and neural networks. We will evaluate the performance of each model (with the metrics defined below) for different data codifications: "bag of words" and embeddings. So, we will be able to compare them to know what is the best solution.

Here is a basic explanation of how these models work:

Logistic Regression:

It tries to separate the data input space into two regions, one for each class, by a linear boundary. During the training step that boundary is calculated minimizing a cost function (the logistic loss). The model predicts the probability of a data point of belonging to a class, which is something related to its position respect the boundary.

Support Vector Machines (SVM):

We use SVM with a linear kernel. With this kernel, the model separates the input space into two regions, one for each class, by a linear boundary. To obtain this boundary, the model tries to find the best one which provides the largest separation, or margin, between the two classes.

Naive Bayes:

It uses the Bayes theorem and the “naive” assumption that all the features are independent. During the training step, it builds frequency and likelihood tables from the training data. During the prediction process, it uses those tables and the Bayes theorem to calculate the probability of a given data point of belonging to a particular class.

Random Forest:

Many decision trees are built (the number is one of the hyperparameters of the model) and each one uses a sample of random variables and is trained with random samples from the training data. The predictions done for all the trees are averaged and returned as output.

Xgboost:

It is very similar to Random Forest but, in this case, there are several levels of trees: the first level of trees is built, following the same principles than Random Forest, and the averaged output is examined. The cases this level misclassify more are given a higher importance (weight) and the next level of trees is built to classify better these cases. The sequential process continues and several levels of trees are generated. At the end, all these levels are combined to produce the output.

Neural Networks:

We have used a feedforward version: their architecture consists of several layers of neurons where the output of each neuron in a layer is connected to the input of each one of the following layer, with an associated weight and activation function. Input data is connected to the first layer, and the last layer, in this case, is an output neuron for the classification result.

During the training step, the backpropagation algorithm is used to find out the weights that fit better for the training data (cost function minimization) and during the prediction process, the feedforward algorithm is applied using the previously obtained weights.

Benchmark

Our benchmark model will be random guessing keeping the same proportion of negative sentiment predictions that the observed in the training dataset. So, if 60% of the tweets express a negative sentiment, our benchmark model will choose randomly 60% tweets from the testing set as negative and the rest as non-negative (neutral or positive sentiment). Tweets text input will be codified as "bag of words" in this benchmark model.

III. Methodology

Data Preprocessing

In the Exploratory Data Analysis we saw that some of the tweets were longer than 140 characters. It looks like that the procedure used to obtain the tweet text was some kind of scrapping and there were some html tags and some special characters codified as html. We applied a filter function based on the BeautifulSoup library to remove the html tags and convert the special chars to utf-8 valid ones.

To obtain a good prediction model it is important to have well labeled data. The dataset was hand-labeled and one of its fields express how confident we can be about the classification done. We removed all rows with a confidence level lower than 0.65 that was about 6.6% of the data. The distribution of the confident level variable is very left skewed so, taking a cutoff bigger than 0.65 would have mean to lose a lot of data, so we decided to keep that value.

We need to convert tweet text into some kind of codification which lets machine learning models work with them. We will use two alternatives: bag of words and embeddings. To train and to use our models we are going to implement pipelines that we will describe in detail in the next section of this document. The first step of these pipelines will be a preprocessing step which will take a tweet text and will perform the following actions:

- Tokenization: we will use the Twitter Tokenizer provided by the NLTK library to obtain all the tokens of a tweet. This tokenizer works pretty well identifying elements as hasthags, twitter user names, emoticons, …
- In the case of the bag of words codification: translate the tokens of each tweet into a bag of words model. We used here the sklearn classes CountVectorizer and TfidfVectorizer (for the tfidf variant).
- In the case of the embeddings codification: we use the Gensim library to build the embeddings and we transform each tweet text into a vector with the average of all its token embeddings.

Implementation

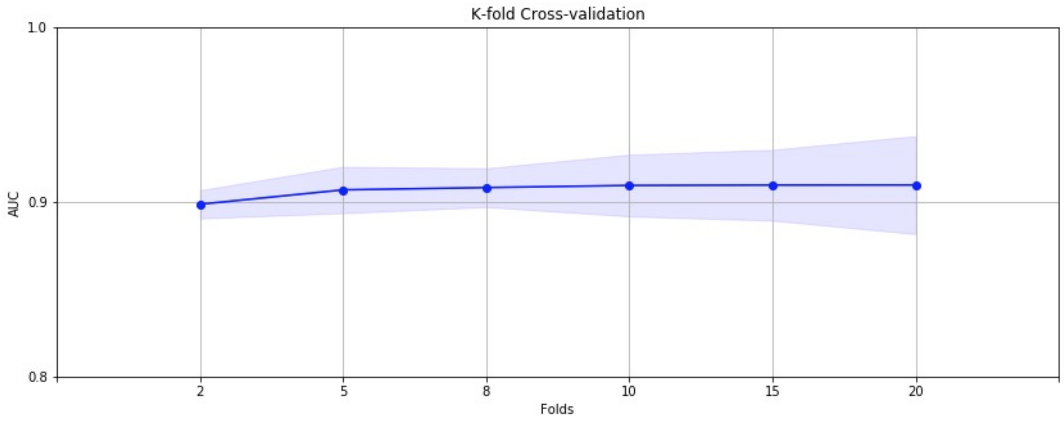
We have split the cleaned dataset into a training and testing set. Over the training data, we have applied cross validation techniques in order to calculate good hyperparameters for our models trying to avoid overfitting. We have used GridSearchCV from sklearn for that, which lets us

explore different combinations of hyperparameters for a model and get their associated cross validation performance score for each one.

When doing cross validation is important to avoid data leakage. This happens when training our model we use some information from the validation or the testing data. If we don't have this precaution we will obtain better but flaw scores and our models won't work so well with new data.

In our particular case, we are going to have different features (vocabulary) depending on the data used to train a model. It is important not to use validation or testing features during the training in order to avoid data leakage. We have built pipelines to preprocess the input data and apply it over the machine learning model to try. We have used this pipeline with GridSearchCV.

To know what could be a good number of folds (k) to use in the cross validation process we have tested an easy model as Logistic Regression with different values of k and we have seen that a typical value of 5 was enough and requires less computational power.



We have evaluated the following machine learning models:

- Logistic Regression
- Support Vector Machine (SVM)
- Naive Bayes
- Random Forest
- Xgboost
- Neural network (feed forward)

For each model we have tried three different input data codification:

- Bag of words
- Bag of words with tf-idf
- Embeddings

So, that produces 18 different initial models (6x3). For each one we have calculated its cross validation auc metric and its auc test validation metric (applying the model over the testing data)

In the case of Logistic Regression, SVM, Naive Bayes and Random Forest we used their sklearn implementation. In the case of Xgboost we used the xgboost package which provides and sklearn interface too (XGBClassifier). To implement the neural network models we used the Keras library.

Some complications we have found during the implementation have been these:

- Unbalanced data: the dataset contains almost twice as many negative tweets as non-negative ones. To get models able to predict with confidence both classes we have used the parameter class_weight="balanced" in the sklearn models, which oversamples the less common classes when training the models. Keras has a similar parameter with the same name that we have used too with the neural network models.

  To build the cross validation folds we have used the sklearn class StratifiedKFold which keeps the same proportion of classes in each fold that is present in the whole data to split.

- Data codification: to process text to be used as input for machine learning models introduces some complexity because there are different ways to do it, like bag of words or embeddings, and each one has its own hyperparameters. The possible combinations to explore can explode if we combine them with the models' hyperparameters.

  We developed custom sklearn transformer classes extending TransformerMixin to convert text input data into averaged embedding vectors.

- Cross validation and bag of words with deep learning models: to be sure about the real performance our models will have with new data we have used cross validation. Each model builds its own features (vocabulary) from their training data and, in the case of our deep learning models, their architecture depend on the number of input features so, we have had to develop a specific function for that.

- Xgboost bug: we found that xgboost had problems to work with the sparse matrix format produced by sklearn vectoricers (CountVectorizer and TfidfVectorizer). It is a reported problem and we found a workaround in the repository project: we had to include a new step in the pipeline to convert the matrix format to one right for xgboost.

- Big number of models: using cross validation and exploring the different combinations of hyperparameters leads to train a lot of models. We used the sklearn and keras option n_jobs=-1 to use all the available cpu capacity of our machine, processing several models in parallel.

Refinement

At the beginning, we tried a simple Logistic Regression over bag of words, with sklearn default hyperparameters. That model produced not bad results:

acc	auc	f1	precision	recall
0.841450	0.919836	0.875966	0.913927	0.841034

Later, for each of the models mentioned in the above section, we explored different hyperparameters combinations to get those with better performance. Some of the models, like

Logistic Regression, Naive Bayes and SVM are fast to train and evaluate and have few hyperparameters, so it was easy to try different values to get good ones. Other models like Random Forest, Xgboost and Neural Networks have far more hyperparameters and are more expensive to train and evaluate. In both cases, we have explored different combinations of hyperparameters to get good ones. In the case of Neural Network we tried different feed forward architectures (different number of layers and neurons per layer)

For the special case of the SVM, we created two different models, where the second one (named “svm_best”) explored a bigger range of hyperparameter combinations, especially those related to the input text codification to bag of words (max number of words, stemming, unigram or bigram, …) It took almost two days to finish but achieved the best results.

To explore and find good hyperparameters we used the sklearn class GridSearchCV which has a parameter named “param_grid” where a dictionary of hyperparameters and its associated lists of values to explore can be passed. GridSearchCV builds a model for each of the possible combinations and evaluate its performance using cross validation. For example, in the case of the “svm_best” model we wanted to explore these hyperparamenters:

```
'countvectorizer__analyzer': ['word', stem],
'countvectorizer__max_df': [0.5, 0.9],
'countvectorizer__min_df': [10, 20, 50],
'countvectorizer__max_features': [1000, 5000, 10000, None],
'countvectorizer__ngram_range': ((1, 1), (2, 2))
'svc__C': [.1, 1, 10, 20]
```

That supposed 384 possible combinations and models (2 x 2 x 3 x 4 x 2 x 4) which were built and evaluated by GridSearchCV. It makes no sense to include a full exhaustive table with the results here, but all the intermediate evaluated models results are reproducible in the notebooks of the project because we fixed a random seed.

In some particular model, the range of hyperparamenter we initially tried and worked right for one of the data representations (bag of words) produced bad results with the other one (embeddings). This happened with the first case of SVM where we explored this hyperparamenter space:

```
'svc__C': [.1, 1, 10]
```

In the case of “bag of words” representation a good solution was achieved for the value of C=1 but, in the case of the “embeddings”, all the three options worked equally bad (worst performance than random guessing). Analyzing the case we found that a so small value of C produced a very big regularization and high bias for a so compact vector representation. We chose bigger numbers to explore with GridSearchCV:

```
'svc__C': [3000, 3500, 4000]
```

and a good solution was found for the point C=4000

We obtained the predictions done by each of these models for the training data (using the best combination of hyperparameters) and calculated their correlation. We saw that for bag of words codification all the models were very correlated with the exception of the “svm_best” (perhaps because it’s the only one which use a very different input data codification). We decided to build a stack combining that model with other, the less correlated one with it, which was the Naive Bayes with tfidf.

To build the stack model we used the StackingClassifier class from the Mlxtend package. We combined the output (probabilities) of the two base models and passed them as input to a new Logistic Regression model. At the end we obtained a stacked model which outperformed all the rest models.

IV. Results

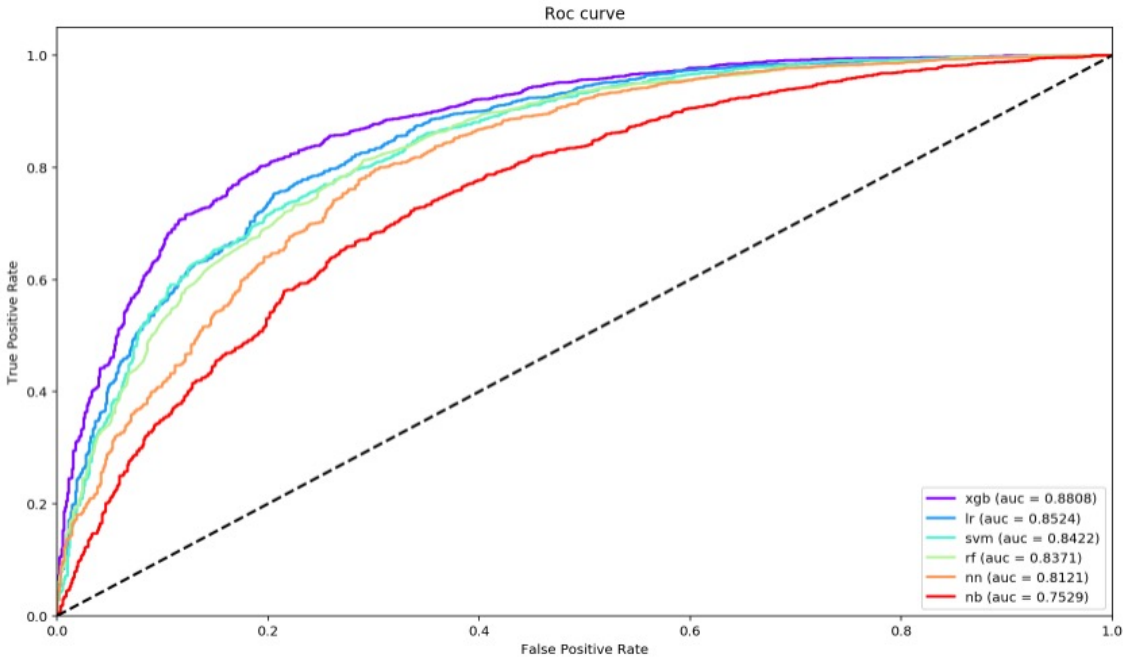
Model Evaluation and Validation

We have seen that models based on data input codified as bag of words produced better performance than those based on embeddings codification. In the first cases, all the evaluated model achieved auc bigger than 0.9 while for embeddings the performance was bellow 0.88

Models and embeddings:

Models show different performance and their predictions are less correlated:

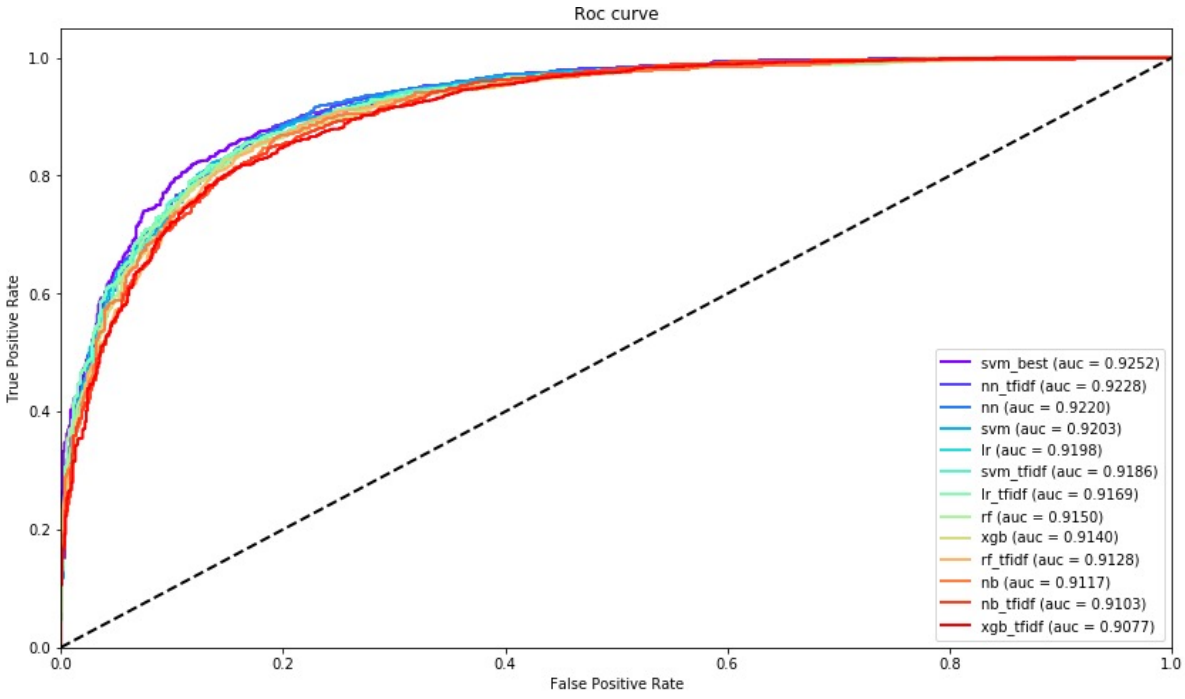
	acc	auc	f1	precision	recall
xgb	0.813255	0.880825	0.866212	0.827984	0.908141
lr	0.779934	0.852449	0.829890	0.854810	0.806381
svm	0.783596	0.842169	0.838480	0.833243	0.843784
rf	0.771512	0.837140	0.847656	0.762072	0.954895
nn	0.778469	0.812081	0.844912	0.791167	0.906491
nb	0.675943	0.752910	0.803639	0.673485	0.996150



Models and bag of words:

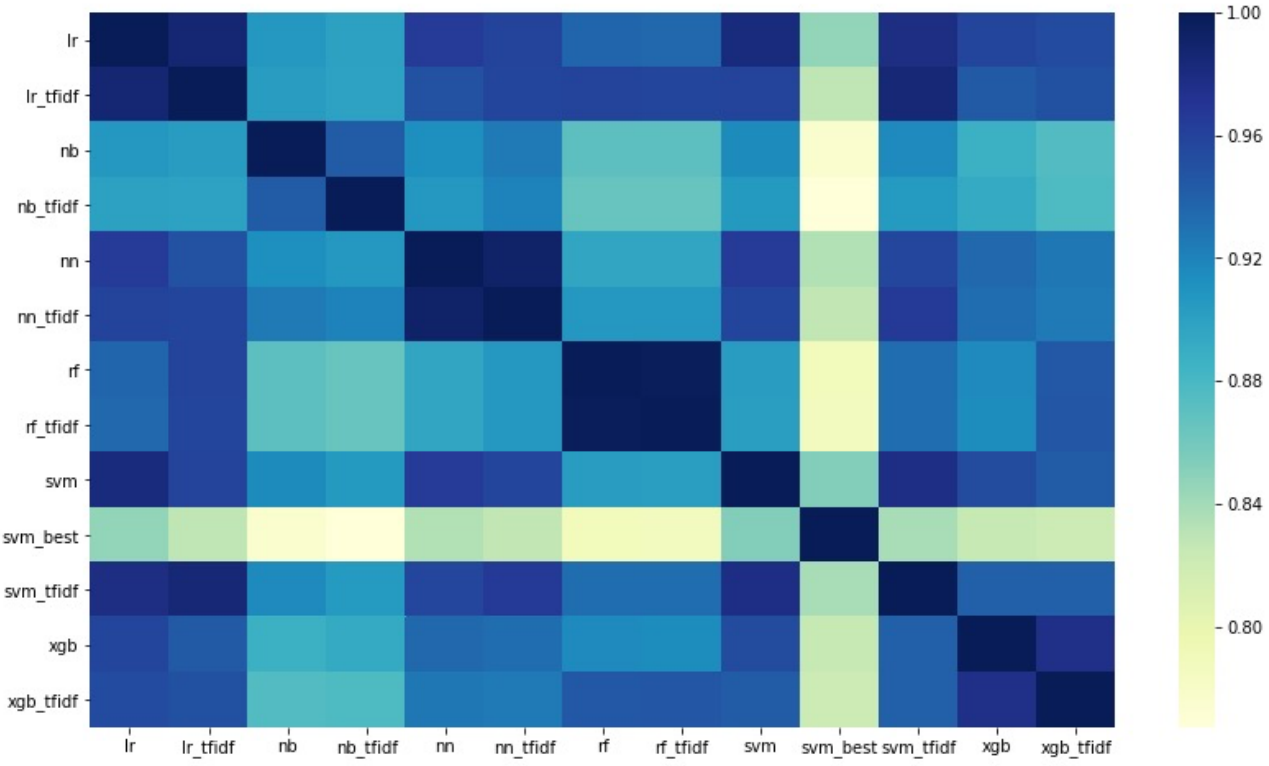
All the models using bag of words produced quite good results and quite similar roc curves:

	acc	auc	f1	precision	recall
svm_best	0.841816	0.925235	0.874710	0.925153	0.829483
nn_tfidf	0.862321	0.922780	0.897715	0.888052	0.907591
nn	0.857927	0.922016	0.892936	0.895903	0.889989
svm	0.836324	0.920276	0.871144	0.915203	0.831133
lr	0.841450	0.919837	0.875967	0.913927	0.841034
svm_tfidf	0.845112	0.918636	0.881213	0.900172	0.863036
lr_tfidf	0.842915	0.916944	0.879596	0.897994	0.861936
rf	0.851337	0.914992	0.891676	0.865803	0.919142
xgb	0.846576	0.913975	0.885675	0.878722	0.892739
rf_tfidf	0.849140	0.912773	0.891807	0.853266	0.933993
nb	0.845112	0.911674	0.890500	0.841076	0.946095
nb_tfidf	0.806298	0.910333	0.871820	0.779125	0.989549
xgb_tfidf	0.842182	0.907657	0.883103	0.871054	0.895490



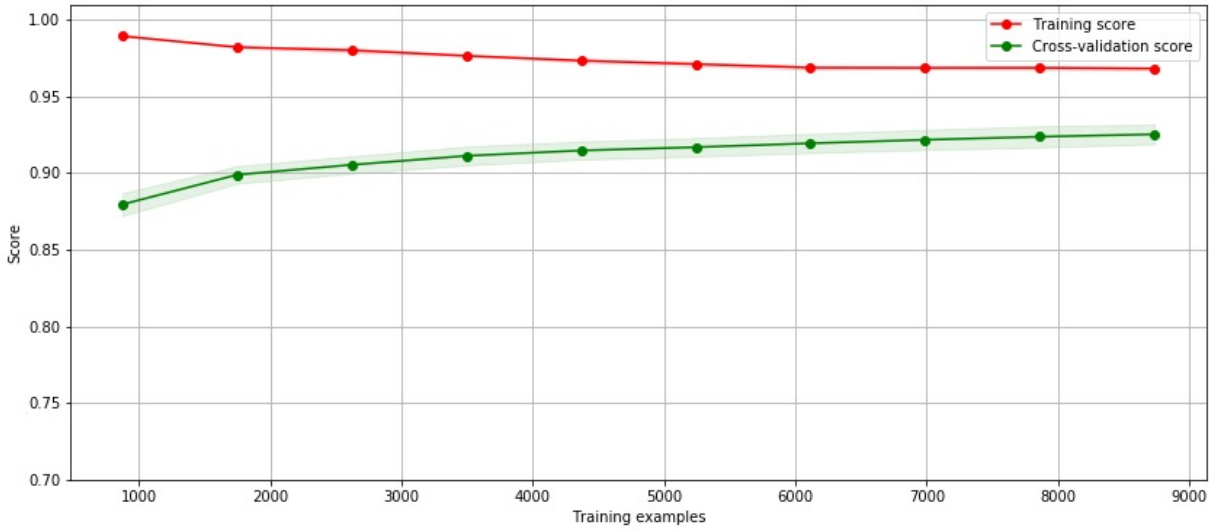
As it is shown in the upper table, the best was a SVM model named “svm_best”. We thought about the possibility of building a stack model to get even better performance and, for that, we looked at the model correlations: a stack can outperform its base model but for that, those must be low correlated. Based on the below graph we chose svm_best and Naive Bayes with tfidf as base models and obtained the best auc, f1 and accuracy performance for the test data:

	acc	auc	f1	precision	recall
	0.872940	0.931535	0.906845	0.885684	0.929043



This stacked model is our final model. It seems to be very stable based on the results we obtained using cross validation: the mean cross validation auc was 0.9249 with a standard error of 0.0049

We obtained its learning curve graph where we can check the model is not overfitting (cross validation score increases as data increases and both lines tend to get closer achieving a good score). It looks like that, with more training data, the model can even get better results.



One of the main downsides of this stacking model is that, given its complexity, it is a bit slow doing predictions. We have calculated it is able to do about 1220 predictions per second while

simpler models like the Logistic Regression one can do about 13503 predictions per second under the same conditions (1107% more)

Justification

Our benchmark model, based on random guessing taking into account the probability of negative tweets seen into the training dataset, reported this performance:

	acc	auc	f1	precision	recall
	0.545222	0.499883	0.651528	0.665609	0.637514

We have seen that our final model, the stacking, outperform the benchmark in all the metrics:

	acc	auc	f1	precision	recall
	0.872940	0.931535	0.906845	0.885684	0.929043

The final model does a pretty good job predicting if a tweet has associated a negative sentiment or not without committing many fails, as we can see because both metrics, auc and f1 are near 1.

In summary, our model looks like appropriated and, in the following section, we will show some application cases where it can be shown in practice.

V. Conclusion

Free-Form Visualization

This is not a visual project, our data here is text, so we are going to show how our final model predicts some interesting new cases:

- "*@united very bad experience!*"

  Ok predicted as a negative sentiment.

- "*@united nice fly. Thanks*"

  Ok predicted as a non-negative sentiment.

- "*@united I hate you. I will never fly again with this company!*"

  Ok predicted as a negative sentiment.

- "*flying with @united is always a great experience*"

  Ok predicted as a non-negative sentiment.

- "*flying with @united is always a great experience. If you don't lose your luggage*"

  This is an interesting case. The first part is the same than the previous tweet which was predicted right as a non-negative sentiment, but it follows another ironic sentence. At the end, the tweet is accurately predicted as a negative sentiment.

- "*@united I prefer to fly with another company*"

  Ok predicted as a negative sentiment.

- "*I always prefer to fly with @united*"

  It is very similar to the previous tweet but it has a non-negative sentiment that our model rightly predicted.

- "I love @united. Sorry, just kidding!"

  Our model failed here and predicted it as a non-negative one but, it is a difficult one.

Reflection

The process used for this project can be summarized using the following steps:

1. Performing an exploratory data analysis and cleaning the data.
2. A benchmark model was created.
3. Implementation of functions and classes to process text to be codified as bag of words, bag of words with tfidf and as embeddings.
4. Building and training the models (Logistic Regression, Support Vector Machine, Naive Bayes, Random Forest, Xgboost and Neural Network) using cross validation and grid search to explore the performance of different combination of hyperparameters. All the models were trained over the three kinds of text codification.
5. Building of of a better Support Vector Machine model ("svm_best") using bag of words codification as input, exploring a bigger range of hyperparameters for the input codification and machine learning model.
6. Evaluation of the created models over test data.
7. Obtaining the correlation between models predictions (over training data)
8. Building of a stack model based on "svm_best" and the less correlated model with it. Training with cross validation technique and evaluation over test data.

One of the most difficult parts of the project was to deal with the conversion of text to a useful format for the machine learning models. We explored a lot of models and data codification techniques and that supposed a lot of work.

The most interesting part of the project has been to find an interesting package to build stacking models and to get confidence in the process of building pipelines for NLP machine learning projects.

Improvement

Here there are some things which could be tried to get an improved solution:

- We have used a naive approach with embedding. There are some modern Deep Learning techniques which can be explored: LSTM and convolutional networks.
- We have built a simple stack model. It could be possible to try different stack architectures combining different base models in different layers.
- We could try bayesian optimization algorithms to explore and get better hyperparameters.
- As we saw in the learning curves, more labeled data would help to get better performance.