Mathematics of Machine Learning & Deep Learning
Homework 2

Marie PHILIPPE & Claire SERRAZ
M2 D3S

Toulouse School of Economics

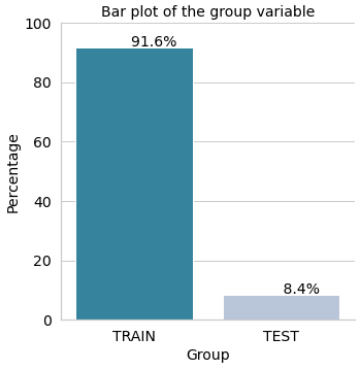
The objective of this project is to analyze the different tweets about Coronavirus that were written in 2020, and to predict whether they are positive or not. To do so, we used two methods: clustering and analysis, but before that we had to do a cleaning work on the dataset, which include four columns: The location, the day and time the tweet has been sent at, the original tweet, and the sentiment. This dataset is directly from twitter, but the observations are anonymous. This project is then split into three parts. The first one will concern the data cleaning, using nltk and the tf-idf key. The second one will be about clustering, using Kmeans, LDA and Non-Negative matrix Factorization, and finally the third one will be about scoring, using several classification algorithms. We decided to go with classification instead of regression as our variable to predict is only composed of three modalities at the end.

Part I: Data understanding

I. Data Understanding

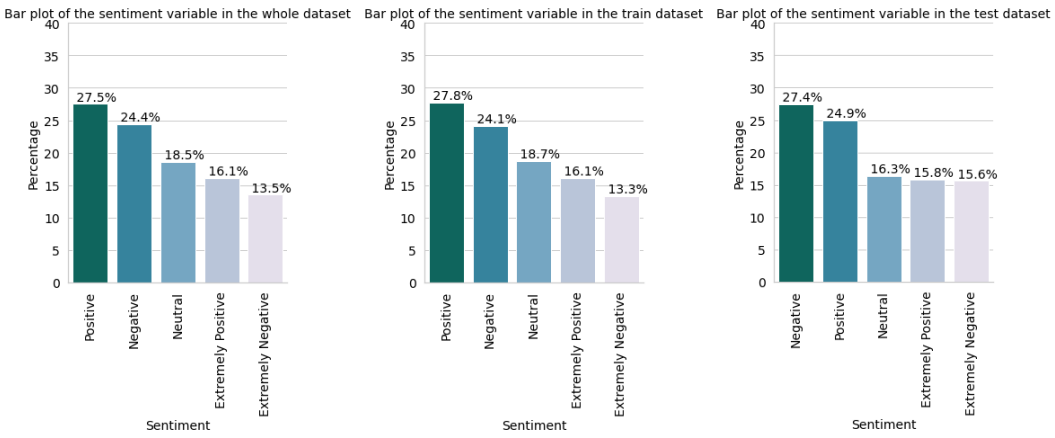
Our data set is divided in two: we have the train dataset that will allow us to fit our models, and the test dataset to make our predictions and validate the models.

Both datasets have six variables (the encoded User Name, the Screen Name and the four variables described above in the introduction). The dataset has 44955 observations, 91.6% of them being in the train dataset and the remaining 8.4% being in the test dataset.



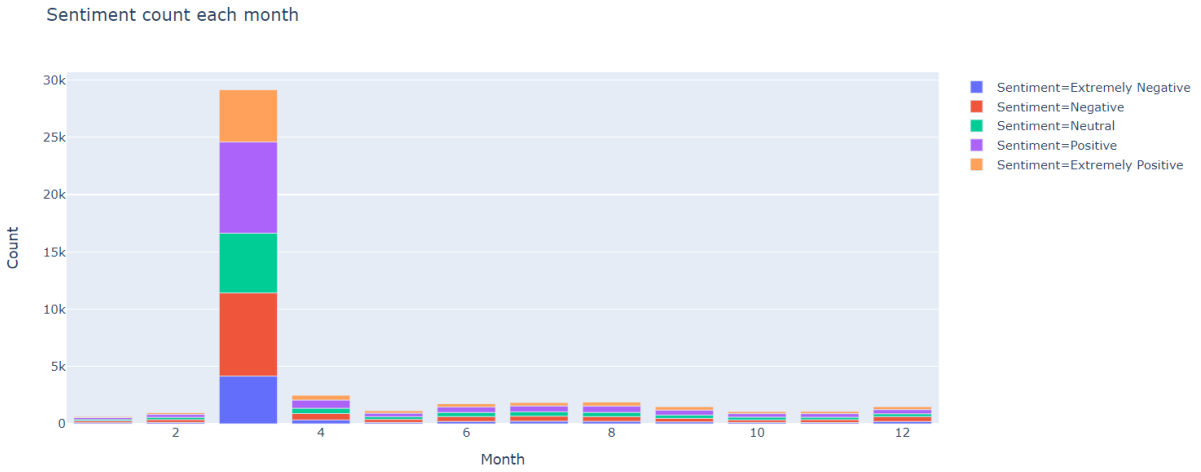
Our dataset only has missing values in the location variable, which is not a problem as we will not analyze this variable in this study as it is not clean. Indeed, this variable has 13137 modalities, but some of them being the same, just written in different ways.

Looking at the distribution of the Sentiment Variable over our dataset, we get:



The distribution is quite the same in both group for the Positive and Negative modalities but slightly differs for the other modalities.

Looking at the evolution with time of the sentiment variable, we have:



The graph is more interactive in the notebook, but globally we can see that most of the tweets were written in March 2020, with the beginning of the quarantines. Sentiments are mostly either positive or negative, but not that much neutral, extremely negative nor extremely positive. In May, October and November, there were not much more tweets about covid than in January or February, i.e. before the pandemic.

II. Data Cleaning

To clean the dataset, we used the libraries *nltk* and *re.* The *nltk* libraries allow to simplify several steps of the data treatment.

We created several functions:

- *Convert_text_to_lowercase*: It allows to put all the charactes in lowercase.
- *Remove punctuation:* It allows to remove some useless punctuation.
- *Tokenize_sentence:* It allows to tokenize the sentence, i.e., to split the word one by one as vectors of letters.
- *Remove_stop_words:* It allows to remove some basic English words and others that we defined
- *Reverse_tokenize_sentence:* It allows to end the tokenization we have dropped some useless words.

We also created a *corpus* function that will more help us to clean by removing urls, mentions, hashtags, digts, html tags, and co.

III. Data visualization

We can now visualize our words and their importance in the dataset with clouds of words. To do so, we used the WordCloud and Image functions from the libraries *wordcloud* and PIL, and a twitter mask.

Figure 1:WordCloud for the full dataset

In the full dataset, the most frequent words are grocery, store, pandemic…

Figure 2: Wordcloud when the sentiment is extremely positive

Figure 3: Wordcloud when the sentiment is positive

When the sentiment Is quite good, the tweet still appears to be about supermarket, store, grocery, and price.

Figure 4: Wordcloud when the sentiment is extremely negative

Figure 5: WordCloud when the sentiment is negative

When the sentiment is less good, we still find key words about supermarket, but more precision: toilet paper, panic, crisis, food, buying…

Figure 6: Wordcloud when the sentiment is neutral

The same words remain when the sentiment is neutral.

Whatever the sentiment is, people seem concern about supermarkets and stores. We cannot them identify which words lead to which sentiment. This is how the clustering part will help us.

IV. Tf-idf vectorization

Later in the analysis, we will need this variable. It is called the *Term Frequency – Inverse Document Frequency*. It is a weighting method that help to estimate the lexical relevance of a word contained in a document, relative to the corpus. The weight proportionally increases according to the number of occurrences of the word in the given document, but also varies according to the frequency of the word in the corpus. Therefore, here is applied a relation between a document and a set of documents that share similarities of lexical fields.

For instance, in the case of a query containing the term X, a document is considered more relevant as a response if it has a certain occurrence of the word X, and if X has a rarity in other documents related to the first.

The function for the TF-IDF key is:

$$W_{X,Y} = tf_{X,Y} * \log\left(\frac{N}{df_x}\right)$$

Where:

- $tf_{X,Y}$ corresponds to the frequency of x in y
- df_x stands for the number of documents containing x
- N is the total number of documents
- X is a word
- Y is a document

As a matter of fact, the TF-IDF key is used by search engines for the referencing of sites during research to propose first the supposedly most relevant site.

Part II: Clustering and topic identification

First, before doing any clustering we changed the number of sentiments to make the prediction easier and code them. Thus, neutral remains neutral and is coded 0. Extremely Negative and Negative both become 1. Finally, Extremely Positive and Positive are put together and are coded 2.

I. Cluster the corpus

To cluster the corpus, we decided to use the k-means algorithm because it is one of the fastest clustering algorithms available. K-means partitioning is a data partitioning method and a combinatorial optimization problem. It is an unsupervised learning algorithm that allows us to identify clusters or similar groups of data points within a data. We randomly initialize the starting centroids K. Each data point is assigned to the nearest centroid. The centroids are recalculated as the average of the data points assigned to the respective cluster. This is done until our stopping criteria are defined. The data points are assigned to the cluster closest to them or, in other words, to the cluster that minimizes the squared distance:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk}(x_n - \mu_k)^2$$

Where J is the sum of the squared distances of each data point from its cluster and r is an indicator function equal to 1 if the data point (x_n) is assigned to group k and 0 in other cases.

In our application of the k-means algorithm, we used 3 parameters:

- n_clusters (= 3): Number of centroids to generate and clusters to create.
- random_state (= 222): Random number generation for centroid initialization to get the same result each time we run the code.
- max_iter (= 10): Maximum number of iterations when the algorithm is run.

Once the k-means algorithm is done, we need to fit the data (here the vector obtained with the tf-idf) with *fit* and finally we get the label of the cluster predicted with *predict*. To know the label of the cluster of each tweet, we create a new column in our dataset that specifies the cluster. We can then split all the words from the tweets of each cluster, and we look at the 10 most common words to define the clusters.

The Silhouette score used after the k-means algorithm enables us to know if a sample has been assigned well to the cluster. It is computed using the mean intra-cluster distance ($\bar{x}_{intra}$) and the mean nearest-cluster distance ($\bar{x}_{nearest}$) for each sample. We then get the score using the following formula:

$$\frac{\bar{x}_{nearest} - \bar{x}_{intra}}{max(\bar{x}_{intra}, \bar{x}_{nearest})}$$

The score goes from -1 (worst value) to 1 (best value). If it is near 0, it means there are overlapping clusters and if it is negative, it indicates that a sample has been assigned to the wrong cluster.

Here we get a score of 0.01 which means we have overlapping clusters. It isn't surprising because when we look at the most common words of each cluster, we can see similarities between them.

II. Profiles of texts

The 10 most common words for the cluster 0 are (in order): Store, grocery, people, workers, retail, go, amp, get, like, going

→ It looks like this cluster refers to stores, especially grocery and retail stores. At the beginning of covid many people wanted to go shopping because of the lockdown announcements, which could also explain why the word "like", "get" and "going" are among the most common words in this cluster.

The 10 most common words for the cluster 1 are (in order): Online, shopping, people, amp, delivery, grocery, home, get, food, time

→ This cluster indicates that people from this cluster tweeted mostly about online shopping. With the lockdowns, people were sometimes afraid to go shopping and seem to have bought or thought about buying online a lot. They probably preferred staying at home and be delivered (especially food) at home directly.

The 10 most common words for the cluster 2 are (in order): Prices, food, supermarket, people, amp, pandemic, panic, stock, need, sanitizer.

→ It is pointed out that the people who wrote the tweets that are in this cluster where panicked. A pandemic means specific sanitary conditions like using hand sanitizer. People usually buy from supermarket and at the time of covid hand sanitizer is bought quite often, like food. However, especially at the beginning of covid there was no more stock of some products needed which could make people worry about the prices for example.

III. Non-negative Matrix Factorization and soft clustering

One of the drawbacks of k-means is that it depends strongly on the initial conditions. Some alternatives are the Non-negative Matrix Factorization (NMF) and soft clustering.

NMF's aim is to extract topics from documents thanks to a dimension reduction. It approximates a matrix $X \in R^{p*n}$ such that $X \approx WH$ where $W \in R^{p*k}$ and $H \in R^{k*n}$ are two nonnegative factors. Each column of X is approximated by a linear combination of the columns of W, the corresponding column H is the coefficient. It enables to extract features of the data in W than can be used for the clustering. We get W and H by minimizing:

$$\frac{1}{2}\left|\left|X - WH\right|\right|_{loss}^{2} \text{ such that } W, H > 0$$

To use the NMF algorithm from the metrics library we used 2 parameters:

- n_components (= 3): Number of components.
- init (= random): Method used to initialize the procedure.

Different methods can be used to initialize the NMF algorithm. Here, we use the non-negative random matrices which means the regularization terms are scaled with scaled with:

$$\frac{\sqrt{X.mean(\quad)}}{n\_components}$$

To learn the NMF model and get the transformed data we use *fit_transform*. We then need to get the indices of the maximum values along the first axis with *argmax* to get the clusters. To get the words the most common in each cluster we use the same procedure as for the k-means.

The 10 most common words for the cluster 0 are (in order): Store, grocery, workers, people, retail, go, amp, get, like, going.

The 10 most common words for the cluster 1 are (in order): Food, supermarket, people, amp, shopping, online, panic, need, stock, sanitizer.

The 10 most common words for the cluster 2 are (in order): Prices, oil, pandemic, amp, price, due, demand, crisis, us, market.

Latent Dirichlet Allocation (LDA) is a soft clustering algorithm that is ideal for text clustering. LDA produces a distribution of groupings whereas usual clustering algorithms produce one grouping, which explains why it is called a soft clustering and not a hard clustering like k-means. This algorithm uses frequency counts of words to find the similarities within topics and find hidden (latent) topics. It is called a mixture model because any topic, here sentiments, are a mixture of words.

The algorithm first goes through each tweet and randomly assigns each word of the tweets to one of the profiles (here we chose 3). For each tweet *t*, the algorithm goes through each word and computes the proportion of words in tweet *t* that are is a profile *k*. If many words from the tweet belong to the profile, then it means that it is very likely that this word belongs to this profile. It also computes the proportion of assignment to the profile k that come from each word, which means how many tweets are in profile k because of a word.

The main differences between LDA and NMF are how they work: LDA with frequency counts whereas NMF works with an approximation by a linear combination, and LDA is a soft clustering algorithm whereas NMF a dimension reduction one. Even though there are both not regular clustering algorithms, they still both enable us to do clustering.

For the LDA algorithm from the decomposition library we used 2 parameters:

- n_components (= 3): Number of components.
- max_iter (= 10): Number of passes over the training data.

To fit the LDA model and get the transformed data we use *fit_transform*. We then need to get the indices of the maximum values along the first axis with *argmax* to get the clusters. To get the words the most common in each cluster we use the same procedure as for the k-means.

The 10 most common words for the cluster 0 are (in order): Store, grocery, supermarket, workers, amp, sanitizer, people, hand, home, food.

The 10 most common words for the cluster 1 are (in order): Supermarket, food, people, store, grocery, panic, shopping, get, amp, need.

The 10 most common words for the cluster 2 are (in order): Prices, food, amp, demand, pandemic, oil, online, due, shopping, crisis.

Cluster 0 Cluster 1 Cluster 2

We can clearly see that many words can be found in several clusters. When looking at each method, we can define three topics by sentiment but when looking at all the methods at the same time we can notice clusters are very similar between the different methods. For example, the cluster 1 obtained with the NMF method is very similar to the cluster 0 obtained

with LDA even though they are supposed to represent two different sentiments. It could be explained by the fact that coronavirus is a pandemic: people get sick, afraid, die… Having a positive thought about covid seems complicated, there aren't many benefits of a pandemic. It could also be due to our text cleaning. Indeed, if we don't take word after word but two words at a time for example, we could get different results. Another reason that could explain these results is that some words can be positive and negative at the same time. For instance, when looking at the word supermarket that can be found in several clusters, we can have tweets of people happy that they stay open despite the pandemic whereas other tweets could be negative because people are worried about shortages.

V. Represent the sentiments according to the clusters

- K-means - cluster in which each sentiment is the most represented with the percentage of tweets of that sentiment in that cluster:

Sentiment	Cluster
Neutral	2 (74%)
Negative or Extremely Negative	2 (81%)
Positive or Extremely Positive	2 (75%)

- NMF - cluster in which each sentiment is the most represented with the percentage of tweets of that sentiment in that cluster:

Sentiment	Cluster
Neutral	2 (55%)
Negative or Extremely Negative	2 (57%)
Positive or Extremely Positive	2 (61%)

- LDA - cluster in which each sentiment is the most represented with the percentage of tweets of that sentiment in that cluster:

Sentiment	Cluster
Neutral	0 (39%)
Negative or Extremely Negative	0 (42%)
Positive or Extremely Positive	0 (40%)

We can notice here that most of the tweets of each sentiment are predicted in the same cluster. It is especially the case with the k-means method since for each sentiment, at least 74% of the tweets are in cluster 2. It is less the case with the NMF and even less with the LDA clustering. It has also been brought to our attention that among all the clusters, the most representative sentiment is the "Positive or Extremely Positive". Since we have 3 clusters and 3 sentiments it could have been nice to be able to attribute a sentiment to each cluster but unfortunately it isn't possible here. This, once again, is probably due to the overlapping of words in the clusters and to the fact that the sentiment maybe doesn't play a role when clustering.

Part III: Sentiment analysis and inference

I. Splitting the data

In this part, we are going to predict as best as we can the sentiment, using several algorithms of classification. As mentioned in the introduction, we choose classification over regression as our variable to predict, the sentiment, is a variable with only three modalities (Neutral 0 – Negative 1 – Positive 2).

To do prediction, we are going to do split our data set in two datasets: train and test. As detailed in the first part, 91.6% of our observations are in the train dataset, when 8.4% are in the test dataset. The train dataset allows to fit our model when the test dataset allows to predict and evaluate the model in terms of accuracy score or error score. The accuracy score corresponds to the percentage of good predictions, and the error score stands for the percentage of bad predictions.

After having split our data set into two datasets, we need to define, for both datasets, our x and y: y being the predicted variable and x being the explicative variable. Here, x is the original tweet, vectorized as a tf-idf score, and y is the sentiment.

As a generalization, we fit our model using the dataset x, predicting y conditionally to x, and then we predict values conditionally to the x from the test dataset, and we compare these predicted values to the y from the test dataset.

For each algorithm we first run it with the default parameters and then optimize it.

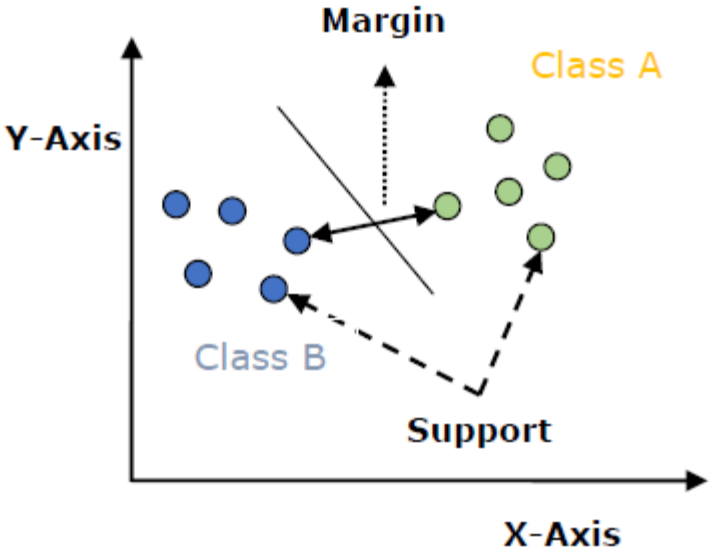
II. Support Vector Machine

Support Vector Machines (SVM) are supervised algorithms that balance power and flexibility. They are mostly used for classification problems. They appeared in the 60s but were well defined in the 90s.

An SVM model is quite easy to implement. It aims to represents different classes in a hyperplane in multidimensional space. This hyperplane is generated in an iteration that aims to minimize the errors. The model tends to divide the datasets into classes to find the maximum marginal hyperplane. To do so, the model will in a first step generate hyperplanes with an iteration to segregate the classes the best as it can. Then, it will decide which hyperplane is the one that split the classes the best.

An SVM model has three important notions:

- **Hyperplane:** As described above, it is a decision space/plane divided between a set of points having different classes.
- **Margin:** Good margins are large margins when bad margins are represented by small margins. A margin corresponds to the gap between two lines on the closest data points of different classes.
- **Support Vector**: It is the space of datapoints that are the closest to the previously defined hyperplane.



To implement a SVM model, we use the package *sklearn* and the function *svm.SVC* (SVC specifies that we use a classification).

We can play on several parameters:

- **C:** It is the regularization parameter, strictly positive and by default set on 1. Regularization avoid overfitting by penalizing high-valued coefficients. We always use l2 penalty here.
- **Kernel:** It specifies the kernel type to be used by the model. It can either be linear, poly, rbf (by default), sigmoid, precomputed...
- **Degree:** It is the degree of the polynomial kernel function if we previously choose poly.
- **Gamma:** It is the kernel coefficient if kernel is set to rbf, poly or sigmoid. By default, it uses 1/(n_features * X.var()). Otherwise, it is named auto and it uses 1/n_features as coefficients.
- **Coef0:** If kernel is set to poly or sigmoid, this is an independent term.
- **Shrinking:** It allows to use or not the shrinking heuristic.
- **Probability:** If set to True, it enables probability estimated.
- **Tol:** It represents the tolerance for stopping criterion (set to 1e-3)
- **Cache_size:** For the kernel, it specifies the size of the cache. By default, it will be 200.
- **Class_weight:** It allows to add a weight to the previously defined parameter C.
- **Verbose:** It enables verbose output if set to True.
- **Max_iter:** It limits the number of iterations.
- **Decision_function_shape:** It helps us to choose between a one vs rest (default value) and a one vs one decision function. The one vs one is always used as multi-class strategy.
- **Break_ties:** By default, it is set to False but can be turned in True, it allows to break ties.
- **Random_state:** If probability is set to true, this allows to control the pseudo random number generation for shuffling the data for probability estimates.

Most of the time, we play on the kernel, the degree, the gamma, the coef0, the tol, the nu, and the max_iter.

When using the default parameters, we get the following results.

	Train	Test
Accuracy Score	59,7%	50,4%

In the detail of each sentiment, we have in the test data set:

Sentiment	Precision
Neutral	53%
Negative	46%
Positive	54%

The confusion matrix is as follows:

		Predictions		
		Neutral	Negative	Positive
Real Sentiments	Neutral	326	154	139
	Negative	522	758	353
	Positive	393	320	833

Here we tried to optimize svm by tuning the kernel between the modalities linear, poly, rbf, sigmoid.

The chosen kernel is sigmoid.

The results are as follows:

	Train	Test
Accuracy Score	45,5%	44,9%

In the detail of each sentiment, we have in the test data set:

Sentiment	Precision
Neutral	40%
Negative	53%
Positive	39%

The confusion matrix is as follows:

		Predictions		
		Neutral	Negative	Positive
Real Sentiments	Neutral	249	196	174
	Negative	484	858	291
	Positive	495	450	601

We get an accuracy score that is less good than without the optimization. It can be due to other default parameters that, set up with the sigmoid kernel, are not optimal anymore. Here, the predictions are quite bad with and without optimization. We could help to improve it by increasing the maximal number of iterations (here set up at 300) but the optimized algorithm would take too long to be run and we would risk overfitting anyway. The accuracy score measures how well the predictions have been on average. For instance, here, the SVM basic algorithm predicted well 45% of the test observations.

III. Linear Support Vector Classification

Following the one below, we now get an interest into the Linear Support Vector Machine for Classification (LSVC). It is quite similar to the one previously used, but with kernel set to linear. Then, it allows to have more flexibility to choose the penalties and loss functions. It is supposed to be better scaled when using many samples. Comparing to before, we here only use the one vs the rest decision function.

We can play on different parameters, common or not with the previous model:

- **Common parameter with SVM:** tol, C, class_weight, verbose, random_state, max_iter
- **penalty:** It gives more flexibility about the penalties. We can either take l1, which leads to sparse vectors, or l2 (default value), which is the one used in SVM, the standard one. More in details, the L1 penalty is the absolute value of the magnitudes of coefficients. It limits the size of the coefficients. The L2 penalty is the square of the magnitude of the coefficients. It means that all coefficients are shrunk by the same factor, and none are eliminated.
- **loss:** It allows to use between the standard SVM loss (called hinge), and the square of the hinge loss (called squared hinge). The hinge loss is defined as $l(y) = \max(0{,}1 - t * y)$.
- **dual:** Set to default by true, it allows to solve the dual optimization problem. If it is turned to False, it solves the primal optimization problem.
- **multi_class:** If y contains more than two classes, we can also use crammer singer that optimizes a joint objective over all class. But this is rarely used in practice (expensive to compute).
- **fit_intercept:** Set by default to True, it permits to compute the intercept of the model.
- **intercept_scaling:** If the previous parameter is set to True, this allows add a weight to the intercept. By default, it will be 1, and then there isn't any weight.

Most of the time, we play on the max_iter, the penalty, the loss.

When using the default parameters, we get the following results.

	Train	Test
Accuracy Score	76,2%	72,9%

In the detail of each sentiment, we have in the test data set:

Sentiment	Precision
Neutral	62%
Negative	72%
Positive	78%

The confusion matrix is as follows:

		Predictions		
		Neutral	Negative	Positive
Real Sentiments	Neutral	383	147	89
	Negative	184	1176	273
	Positive	109	226	1211

Here we tried to optimize lsvm by tuning these parameters:

- **penalty:** between l1 and l2
- **loss:** between hinge and squared hinge

The chosen penalty is l2 and the chosen loss is hinge.

The results are as follows:

	Train	Test
Accuracy Score	76,4%	73,5%

In the detail of each sentiment, we have in the test data set:

Sentiment	Precision
Neutral	71%
Negative	71%
Positive	77%

The confusion matrix is as follows:

		Predictions		
		Neutral	Negative	Positive
Real Sentiments	Neutral	437	108	74
	Negative	207	1159	267
	Positive	121	228	1197

Here the results are better when optimizing the algorithm than without optimization. We predicted well 73,5% of the test observations The precision is the best when predicting the positive sentiment as 77% of the positive sentiment in the test dataset have been well predicted. If we look at the two confusion matrices, we can see that when optimizing, the sentiments neutral and negative are more well predicted than when not optimized, but it is not the case for the positive sentiment. For instance, 437 neutral tweets were well predicted when optimizing, against only 383 neutral tweets well predicted when not optimizing.

IV. Linear Classifiers with SGD training

This estimator uses regularized linear model with stochastic gradient descent learning. The mechanism is as follow: the gradient of the loss is estimating each sample at a time and the model is updated along the processes, with a decreasing learning rate. SGD permits mini back learning. This method can only work if data are represented as dense or sparse arrays of floating-point values for the features. The loss parameter allows to control the fitting of the model, and by default will fit a linear SVM as defined below. The regularizer is here a penalty added to the loss function, allowing to shrink model parameters towards 0, using either the l1 or l2 penalty or combination of both.

We can play on different parameters, common or not with the previous model:

- **Common parameter with Linear SVM:** Loss, Penalty (the elasticnet modality is a combination of both), fit_intercept, max_iter, tol, verbose, random_state, class_weight).
- **alpha:** It is constant that acts as a weight for the regularization term. The higher the alpha, the stronger the regularization.
- **L1_ratio:** It is the l1_ratio in the elastic net penalty.
- **shuffle:** By default, it is set to true, which allow the training data to be shuffled after each epoch.
- **epsilon:** It is the epsilon-insensitive loss functions. For huber loss function, it is the threshold at which it becomes less important to get the prediction exactly right.
- **n_jobs:** It is the number of CPUs to use to do the one versus all.
- **learning_rate:** It can either be constant (eta=eta0), optimal (eta=1/(alpha*(t+t0)), invscalling (eta=eta0/pow(t,power_t)), or adaptive (eta=eta0). It allows to compute eta conditionally to eta0.
- **eta0:** It is the initial learning rate.
- **power_t:** It is the exponent for inverse scaling learning rate.
- **early_stopping:** When set on True, it allows to use early stopping to terminate training when validation score is not improving.
- **n_iter_no_change:** It is the number of iteractions with no improvement to wait before stopping fitting.
- **warm_start:** When turned to True, it recycles the solution of the previous call to fit as initialization.
- **average:** When set to True, it allows to compute the averaged SGD weights across all updates and save the result. It set to an integer greater than 1, averaging will begin after the total number of samples seen reaches the average.

Most of the time, we play on the max_iter, the penalty, the loss, and the learning_rate.

When using the default parameters, we get the following results.

	Train	Test
Accuracy Score	75,4%	72%

In the detail of each sentiment, we have in the test data set:

Sentiment	Precision
Neutral	51%
Negative	73%
Positive	80%

The confusion matrix is as follows:

		Predictions		
		Neutral	Negative	Positive
Real Sentiments	Neutral	318	181	120
	Negative	127	1200	306
	Positive	72	241	1233

Here we tried to optimize Linear SVM by tuning these parameters:

- **The penalty:** between l1 and l2
- **The learning rate:** between optimal, and constant

The chosen penalty is l1, le learning rate is optimal.

The results are as follows:

	Train	Test
Accuracy Score	75,8%	73%

In the detail of each sentiment, we have in the test data set:

Sentiment	Precision
Neutral	74%
Negative	69%
Positive	77%

The confusion matrix is as follows:

		Predictions		
		Neutral	Negative	Positive
Real Sentiments	Neutral	461	95	63
	Negative	233	1122	278
	Positive	146	210	1190

If the accuracy score is better when optimizing the algorithm (we predict well 73% of the test observations), we also can see with the precisions and the confusion matrices that actually, the optimized algorithm predicts more accurately the neutral sentiment, compared to the non-optimized one, but not the negative and positive sentiments. For instance, 1190 positive tweets were well predicted in the optimized algorithm, against 1233 positive tweets well predicted in the non-optimized algorithm.

V. Gaussian Naïve Bayes

This is a different model form the ones above. This is a variant of Naïve Bayes (a group of supervised machine learning classification algorithms based on the Bayes theorem) that follows Gaussian normal distribution and supports continuous data. Naïve Bayes executes quite simple classification, but that gives high predictions. As a reminder, the Bayes Theorem states:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A) * P(B|A)}{P(B)}$$

With P(i) being the probability of i (A or B) occurring, P(i|j) being the probability of i given j (i and j being A or B), P(A∩B) being the probability of both A and B occurring.

An important assumption is the strong independence assumptions between the features of a classification. Thus, the Naïve Bayes Classifiers will be trained efficiently. Gaussian Naïve Bayes Classifiers assume that the continuous values associated with each class are distributed according to a Gaussian distribution. As a reminder, here is the probability:

$$P(x_i|y) = \frac{1}{\sqrt{1\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

A Gaussian Naive Bayes Classifier has only two parameters:

- **priors:** It is the prior probabilities of the classes. If specified, the priors are not adjusted according to the data.
- **var_smoothing:** It is the portion of the largest variance of all features that is added to variances for calculation stability.

When using the default parameters, we get the following results.

	Train	Test
Accuracy Score	63,7%	60%

In the detail of each sentiment, we have in the test data set:

Sentiment	Precision
Neutral	78%
Negative	55%
Positive	59%

The confusion matrix is as follows:

		Predictions		
		Neutral	Negative	Positive
Real Sentiments	Neutral	485	78	56
	Negative	457	893	283
	Positive	343	293	910

Here we tried to optimize Gaussian Naïve Bayes by tuning the parameter var_smoothing making it evoluing between 1e-9 and 0.1.

The chosen var_smoothing is the one by defaut: 1e-9. We get the exact same results as before since the tuned var_smoothing is equal to the default value.

We predicted well 60% of the observations. It is not as good as the previous algorithm we saw. However, the precision for the neutral tweets is quite high: 78% of prediction. The confusion matrix confirms this high precision, as 485 neutral tweets were well predicted, and only 78 neutral tweets were predicted to be negative and only 56 neutral tweets were predicted to be positive.

VI. Extreme Gradient Boost

Extreme Gradient Boost (XGBoost) is built on the principles of stochastic gradient boosting and enables classification. It is an ensemble machine learning algorithm: with a single model we get aggregated output from several models. From decision tree models we can construct the ensembles. In boosting, trees are built sequentially and added to the ensemble. The models are fitted each time to reduce the prediction error of the previous tree. The loss function and gradient descent optimization algorithms are used for the fitting. We use the function *XGBClassifier* from the package *xgboost*.

XGBoost has three types of parameters.

- General parameters: Specially to set if the algorithm will work with tree or linear model
- Booster parameters: Parameters of the booster at each step
- Learning task parameters: Parameters for the optimization

Among the general parameters we can find:

- **booster:** It is by default equal to gbtree to use a tree model.
- **verbosity:** To choose if messages are printed or not. By default, warnings are printed.
- **nthread:** Number of parallel threads.

We then have several booster parameters:

- **learning_rate:** Step size, the default value is 0.3.
- **min_split_loss:** Minimum loss reduction required to make a split, i.e a new partition on a leaf node. The default value is 0.
- **max_depth:** Maximum depth of a tree, by default it is 6. If the value is high, it can lead to overfitting.
- **min_child_weight:** Minimum sum of weights of the observations needed in a child, the default value is 1. It is used to control overfitting. If a leaf node has a sum of weights smaller than the value of min_child_weight then the next partitioning isn't done.
- **max_delta_step:** Maximum delta step allowed at each tree weight estimation (0 as default value).

- **subsample:** Ratio of the training subsample for each tree, the default value is 0. Small values are better to avoid overfitting but if the value is chosen too small then it can underfit.
- **colsample_bytree:** Ratio of the columns subsample when constructing each tree.
- **reg_lambda:** L2 regularization term on weights, it can reduce overfitting (1 by default).
- **reg_alpha:** L1 regularization term on weights, 0 by default. It is essentially used in high dimensionality because it can enable the algorithm to run faster.

Finally, in the learning task parameters there is:

- **objective:** Enables to choose the loss function (linear (default), logistic regression for binary classification, multiclass classification or multiclass classification returning predicted probability…).
- **eval_metric:** Evaluation metrics for validation data (mean squared error, logloss, mean absolute error, auc…), it depends on the objective.
- **seed:** To get reproductible results, the default value is 0.

Parameters we can play with in XGBoost are: learning_rate, reg_lambda, reg_alpha, max_depth and min_child_weight.

When using the default parameters, we get the following results.

	Train	Test
Accuracy Score	78,3%	72,1%

In the detail of each sentiment, we have in the test data set:

Sentiment	Precision
Neutral	70%
Negative	69%
Positive	76%

The confusion matrix is as follows:

		Predictions		
		Neutral	Negative	Positive
Real Sentiments	Neutral	484	114	71
	Negative	214	1126	293
	Positive	133	235	1178

We didn't manage to optimize the XGBoost algorithm because it was taking too long, even when tuning only one parameter at a time. The method would have been the same as for the other algorithms. But still, this is the most satisfying algorithm in terms of accuracy score on the training score as 78,3% of the observations were well observed. It is not our best testing accuracy score (72,1%) but it is still quite good, it means that 72.1% of our test tweets were predicted in the correct sentiment. Here, the positive tweets are very well predicted. Indeed, the precision is of 76%. 1178 positive tweets were correctly predicted, when only 368 positive tweets were not correctly predicted.

VII. Non-Negative Matrix Factorization

From what we did in the clustering part, here are the results we got it:

	Test
Accuracy Score	34%

In the detail of each sentiment, we have in the test data set:

Sentiment	Precision
Neutral	20%
Negative	27%
Positive	39%

The confusion matrix is as follows:

		Predictions		
		Neutral	Negative	Positive
Real Sentiments	Neutral	186	20	413
	Negative	361	49	1223
	Positive	374	114	1058

This algorithm doesn't give satisfying result and is yet to be improved by more parametrization. The accuracy score is very low: only 34% of our observations were well predicted. The precision for the neutral sentiment is the worst of all our algorithms: 20%. The confusion matrix shows us that 1223 negative tweets were predicted to be positive, which represents here a big error as there are only 1633 negative tweets.

VIII. Other algorithms tested

We also tested other basic algorithms. We tested Decision Tree and Random Forest as classifiers, but our first intuition was that they're not well suited for our analysis, and the results confirmed our intuition. Indeed, the training accuracy score are very good, around 99% for both, but not that good the testing accuracy score, around 60%. We are in a situation of underfitting.

However, using the logistic regression as classifier was quite surprisingly good. Here as the results:

When using the default parameters, we get the following results.

	Train	Test
Accuracy Score	76%	73.1%

In the detail of each sentiment, we have in the test data set:

Sentiment	Precision
Neutral	58%
Negative	73%
Positive	79%

The confusion matrix is as follows:

		Predictions		
		Neutral	Negative	Positive
Real Sentiments	Neutral	361	164	94
	Negative	161	1194	278
	Positive	95	228	1223

The results are quite satisfying result. Indeed, we get an accuracy score of 73,1% with the test observations and 76% with the train ones. The precision is a little less good for the neutral sentiment than for the other: 58% against more than 73%. Only around half of the neutral tweets were predicted to be neutral. It is much better when we look at the positive tweets because 1223 were predicted to be positive, knowing there are 1546 positive tweets.

Part IV: Generate text sequence (Optional)

We didn't manage to do this part, but this is how we would have done it. First, we wonder here if the tweets are enough specific to the sentiment. The biggest issue is whether the text will make sense and will grammatically be correct. We can use two models for that. The first one is a Long Short-Term Memory. The second is a GPT-2 Transformer. According to the literature, the GPT-2 Transformer gives sentences that are quite grammatically comprehensible, but the deep meaning of the sentence is not that clear.

Conclusion

To conclude, the different methods (k-means, NMF, LDA) of clustering of the tweets gave us very similar results. Furthermore, several words could be found in different clusters of a same method. As explained, it is probably due to overlapping or we could have cleaned more the tweets. This made difficult the representation of the sentiments according to the clusters. We couldn't define well if a cluster is related to a sentiment in particular. After the clustering, we used several algorithms to do classification. For the algorithms not running too long, we tuned some parameters to try to get better results. Some of the best results in terms of accuracy were gotten with the optimized linear support vector classification, the linear classifiers with SGD training, and XGboost even though we didn't manage to optimize it because it was running hours.

Bibliography

"Scikit-Learn: Machine Learning in Python — Scikit-Learn 1.0.1 Documentation." *Scikit-Learn: Machine Learning in Python — Scikit-Learn 0.16.1 Documentation*, https://scikit-learn.org/stable/. Accessed 1 Dec. 2021.

"Support Vector Machine (SVM)." *Biggest Online Tutorials Library*, https://www.tutorialspoint.com/machine_learning_with_python/classification_algorithms_support_vector_machine.htm. Accessed 1 Dec. 2021.

"XGBoost Parameters | XGBoost Parameter Tuning." *Analytics Vidhya*, https://www.facebook.com/AnalyticsVidhya/, 1 Mar. 2016, https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/.

"XGBoost Parameters — Xgboost 1.5.1 Documentation." *XGBoost Documentation — Xgboost 1.5.1 Documentation*, https://xgboost.readthedocs.io/en/stable/parameter.html. Accessed 1 Dec. 2021.