Manuel Ortiz Cachá (k20114995)

7CCSMDM1 Data Mining

Coursework 2

Due: Thursday 26 March 2021 (23:59 UK time)

PART 1: TEXT MINING

We will start by downloading the respective dataset: *Coronavirus Tweets NLP*, from Kaggle. The aim of this part of the Coursework is to predict the sentiment of Tweets relevant to Covid-19.

From the *.info()* method we can infer that this dataframe contains 41,157 entries (rows) and 6 columns. These columns are explained in the Coursework description, we will benefit from the table that they use there to explain these attributes:

Attribute	**Description**
UserName	Anonymized attribute.
ScreenName	Anonymized attribute.
Location	Location of the person having made the tweet.
TweetAt	Date.
OriginalTweet	Textual content of the tweet.
Sentiment	Emotion of the tweet.

The two first columns (*UserName* and *ScreenName*) are integers that do not give us much information. They are anonymized, I suppose that for privacy reasons. The following 4 columns (*Location*, *TweetAt*, *OriginalTweet* and *Sentiment*) are of object type, meaning that they are strings.

From all the columns, only one of them contains missing values, this is the *Location* attribute. It has 32,567 values, so it contains 8,590 null values.

1. ***[20 points] Compute the possible sentiments that a tweet may have, the second most popular sentiment in the tweets, and the date with the greatest number of extremely positive tweets. Next, convert the messages to lower case, replace non-alphabetical characters with whitespaces and ensure that the words of a message are separated by a single whitespace.***

1.1: Compute the possible sentiments a tweet may have.

This question is asking us to find out how many possible classes our dataset has. In this exercise we will define the classes as possible sentiments a tweet may express. We can find out easily with the built-in method *.unique()*. We observe that there are 5 possible sentiments characterising a tweet. They go from Extremely Negative to Extremely Positive, passing by Negative, Neutral and Positive.

In the following table I will leave the number of times each sentiment is found in this dataset. The values are obtained by passing the built-in function *.value_counts()* to the *Sentiment* column:

Sentiment	Number of occurrences
Extremely Negative	5,481
Negative	9,917
Neutral	7,713
Positive	11,422
Extremely Positive	6,624

From the table above we can infer that this dataset is slightly more optimistic. There are more Positive than Negative tweets, which also occurs for the extremes: there are more Extremely Positive than Extremely Negative tweets.

1.2: Compute the second most popular sentiment in the tweets.

As we have seen in the last question, the second most popular sentiment in our dataset is the **Negative** sentiment, containing 9,917 negative tweets. As an optimistic fact, we will also say that the sentiment most represented in this dataset is Positive.

1.3: Compute the date with the greatest number of extremely positive tweets.

This question is asking us to find out which day there were the most extremely positive tweets. In order to compute this, we should first convert the dates from the *TweetAt* column into a timestamp format. Currently, they are strings, so we are going to convert them by using the built-in function *.to_datetime()* offered by pandas. Now, our dates are of type *pandas._libs.tslibs.timestamps.Timestamp*.

Once we have our dataframe correctly defined, we have to filter it first by the sentiment we are looking for (Extremely Positive). After that, we can use the pandas built-in method *.groupby('TweetAt')* to group our dataset by the date. And once we have our *groupby* object, we can apply the *.count()* method to return the number of Extremely Positive tweets made each day.

After sorting the resulting pandas Series, we find out that the most Extremely Positive day during the quarantine was the **25th March 2020**. This specific day a total of 545 Extremely Positive tweets were posted by the users the dataframe collects.

Further analysing this issue, it seems like it was during the initial days of the lockdown where the people was the most positive or optimistic. From the 17th to the 26th March were the days with the most Extremely Positive tweets. After that there is a heavy fall of this kind of tweets. However, at the last days our dataframe collects, there is another slight rise of optimistic tweets.

Something similar happened to me, at first, I tried to look at the bright side of being locked at home with my family: I could pass time with them, I could catch up on my behind schedule tasks, etc. But as the time passed by, I struggled to find the optimism until the last days of quarantine where I could see the light at the end of the tunnel.

1.4: Convert the messages to lower case, replace non-alphabetical characters with whitespaces and ensure that the words of a message are separated by a single whitespace.

This last job of the first question is to ease the analysis we will perform later. As we are using a pandas DataFrame, we have the possibility and advantage of realising vectorised built-in functions to effectively perform this kind of tasks. We will act in a similar manner for these subtasks: we will first create a function and we will later pass that function to the *.apply()* function. This *.apply()* function will be applied to the *OriginalTweet* column so our tweets can be transformed.

To convert the messages to lower case we only have to pass *str.lower* to the *.apply()* method we are applying to the *OriginalTweet* column. We will store it in a new column called *ModifiedTweet*.

For the second subtask (replace the non-alphabetical characters with a whitespace) we have defined a function called *alpha*. This function accepts a message as an input and returns that same message with the non-alphabetical characters replaced by a whitespace. This is done easily with *.replace()*. Later, we pass this *alpha* function through the *.apply()* method to all the tweets in our dataframe, the *ModifiedTweet* column.

Lastly, we must ensure that the words of a tweet are separated by a single whitespace. So, we have to eliminate the repeated whitespaces that are one after the other. We will do so by first splitting the words in each tweet (*.split()*), so we can later *" ".join()* them with a space in between. We can *.apply()* this directly to the *ModifiedTweet* column in a single clause, helping us by the lambda function, so we do not need to create a function like in the previous subtask.

2. [20 points] Tokenize the tweets (i.e. convert each into a list of words), count the total number of all words (including repetitions), the number of all distinct words and the 10 most frequent words in the corpus. Remove stop words, words with ≤2 characters and recalculate the number of all words (including repetitions) and the 10 most frequent words in the modified corpus. What do you observe?

2.1: Tokenize the tweets (i.e. convert each into a list of words).

There are several packages that can help us to do this. One of them is *NLTK*, which has a built-in function called *.word_tokenize()* that specifically does this. As the Coursework description asks us explicitly not to use it ("*you are expected to use raw Python string functions for text processing operations*"), we will use Python string functions to do so.

The good thing is that the method we used in the above 1.4 section, *.split()*, does the same work. And since we pre-processed the tweets, it will work perfectly on our dataset. So, we create a new column which we will call *TokenizedTweets*, in which we will store the tokenized tweets.

2.2: Count the total number of all words (including repetitions).

We could do so by applying the *.len()* method to the new column we just created, *TokenizedTweets*, and then sum all the lengths. Instead, we will create a dictionary where we will store all the words that appear in our tweets. For this we first have to create a corpus.

A corpus is "a collection or body of knowledge or evidence" [1]. In our specific case, we will create a corpus that will contain all the tweets of our dataset. We will join all of them and store them in a list called *corpus*. We will do so by combining the values in the *TokenizedTweets* column, so it is faster.

Once created the corpus, we could already return the number of words (repetitions included) in our dataframe (length of the *corpus* list). However, we will go a step further and create a dictionary that will contain the unique words of our *corpus* as the keys and the number of occurrences of each word as the values. We will do so by looping through *co rpus* and creating a new key-value entry in case a word does not exist yet in our dictionary; and adding 1 to the value of that word if it already exists. We will name this dictionary *words*.

Finally, we could count the number of repeated words in our dataset by summing the values of our dictionary (*words*), or by returning the length of the list *corpus*. For efficiency methods we will apply the second method. At the end, we obtain that there is a total of **1,354,474 words (repetitions included)** in all the tweets we have.

2.3: Count the number of all distinct words.

This step is really easy having in mind that we already have created a dictionary, *words*, with the unique words as keys. We only have to count the number of keys in *words*. In total, this dataframe contains **81,517 unique words**.

2.4: Count the 10 most frequent words in the corpus.

As before, with the dictionary counter already created, this step is quite straightforward. We have to sort the values in *words* in descending order and print the 10 most frequent words, the top 10 values. The result is the following:

RANKING	WORD	FREQUENCY (number of occurrences)
1	the	44,894
2	to	38,495
3	t	29,887
4	co	24,150
5	and	24,097
6	https	24,007
7	covid	23,229
8	of	21,565
9	a	19,947
10	in	19,348

As we can see, the great majority of the words in the top 10 are stopwords or very short words that do not contribute much to our analysis. The only words that could make more sense to analyse further are 'covid' and 'https', which may indicate why are all these tweets grouped

[1] https://www.merriam-webster.com/dictionary/corpus

together (*Coronavirus Tweets NLP*) and that there are a lot of links or photos in our database. To make a good analysis we should remove this kind of words (stopwords and very short words) and make the same exercise.

2.5: Remove stopwords and words with 2 characters or less. Recalculate the number of all words (including repetitions) and the 10 most frequent words in the modified corpus. What do you observe?

We will proceed in a similar manner as we did in 2.2, we are going to create another dictionary called *words2*. This dictionary will add key-values from the *corpus* list. The key difference this time is that we will filter the words by excluding words that belong to a list of stopwords or if they have 2 characters or less. I have obtained the stopwords from the *NLTK* package *nltk.stopwords.words.('english')*. In particular, this is a list of English words that do not add much meaning to a sentence.

Once created *words2*, we can extract some additional information. Without stopwords, **our modified corpus has a total of 782,506 words, including repetitions** (57.77% of the words from the previous corpus). **This modified corpus has 80,654 unique words** (98.94% of the previous corpus). This is very explanatory, removing the stopwords we have reduced the vocabulary of our corpus very little, removing just over the 1% from the last vocabulary. However, we have removed more than the 40% of the total words from the previous corpus. Which means that we were analysing a corpus with almost the half of its words being uninformative.

Finally, the top 10 words of the modified corpus regarding its frequency are the following:

RANKING	WORD	FREQUENCY (number of occurrences)
1	https	24,007
2	covid	23,229
3	coronavirus	18,167
4	prices	7,944
5	food	7,172
6	supermarket	7,082
7	store	6,919
8	grocery	6,284
9	people	5,574
10	amp	5,198

As we can see, this frequency table has nothing to do with the one we saw in the section 2.4. For a start, the most frequent word, 'https', has almost the half of the occurrences compared to 'the', the most frequent word in the previous corpus. We can also observe that from the initial frequency table, only two words have held out. The analyse could be improved by removing some words of three characters, like 'amp', which does not mean nothing related to Covid; or by removing words like 'https' which does not add value to the analysis neither.

This table frequency is much more explanatory. If we would like to compare tweets and compute a cosine similarity on them, the second dictionary (*words2*) will help us much more than the first one. Having 4 similar words may be a good indicator, but if they are 4 stopwords or 4 very short words, it contributes little to the analysis. On the other hand, if 3 of those 4 words are from the *words2* dictionary, it may indicate that these tweets are talking about a similar subject.

3. ***[10 points] Plot a histogram with word frequencies, where the horizontal axis corresponds to words, while the vertical axis indicates the fraction of documents in which a word appears. The words should be sorted in increasing order of their frequencies. Because the size of the data set is quite big, use a line chart for this, instead of a histogram. In what way this plot can be useful for deciding the size of the term document matrix? How many terms would you add in a term-document matrix for this data set?***

This exercise asks us to show a line plot of the word frequencies. These frequencies are the fraction of the number of times each word appears in a document. So, if a word has a frequency of 0.5, it means that it appears in half of the documents (tweets).

To do this we apply a set of steps, which I have explained in more detail in the python script. Mainly, what we are doing is creating a pandas multi-index Series that contains all the unique words in each tweet (one unique word of a tweet per row). The outer index contains an entry per tweet, so it has the same length as the initial dataframe. The inner index contains an entry for each unique word in each tweet.

We have managed this by creating a set (without duplicates) of the *TokenizedTweet* column, and then stacking the unique values in each tweet. This way we have ended up with a Series that contains as values the unique words in each tweet. This allows us to apply *.value_counts()* so we can have the count of the documents each word appears in.

This Series has all the words in our corpus, meaning it also contains the stopwords and the very short words. As we are not interested in the frequencies of these words because they are not very informative, we will *.drop()* these values from our Series. Lastly, by dividing all the values in this Series by the total number of tweets (documents) in our corpus, we finally obtain the frequencies of each word.

We show the image in the next page so it can be analysed better. We have rendered two different plots. The one above shows the frequencies of all the words in our vocabulary, we have showed some words (1 of each 2500). The one below shows the frequency of the 20 most frequent words, it is easily observable how the y-axis grows much faster at the end of the line plot. The plot below could be thought as a zoom of the last section of the first plot.

As we can see, there is a huge vertical asymptote when a particular word is reached. This asymptote is at the very final of our line plot, so there is a small number of words that appear in more than the 10% of documents/tweets, meaning that the frequency is above 0.1.

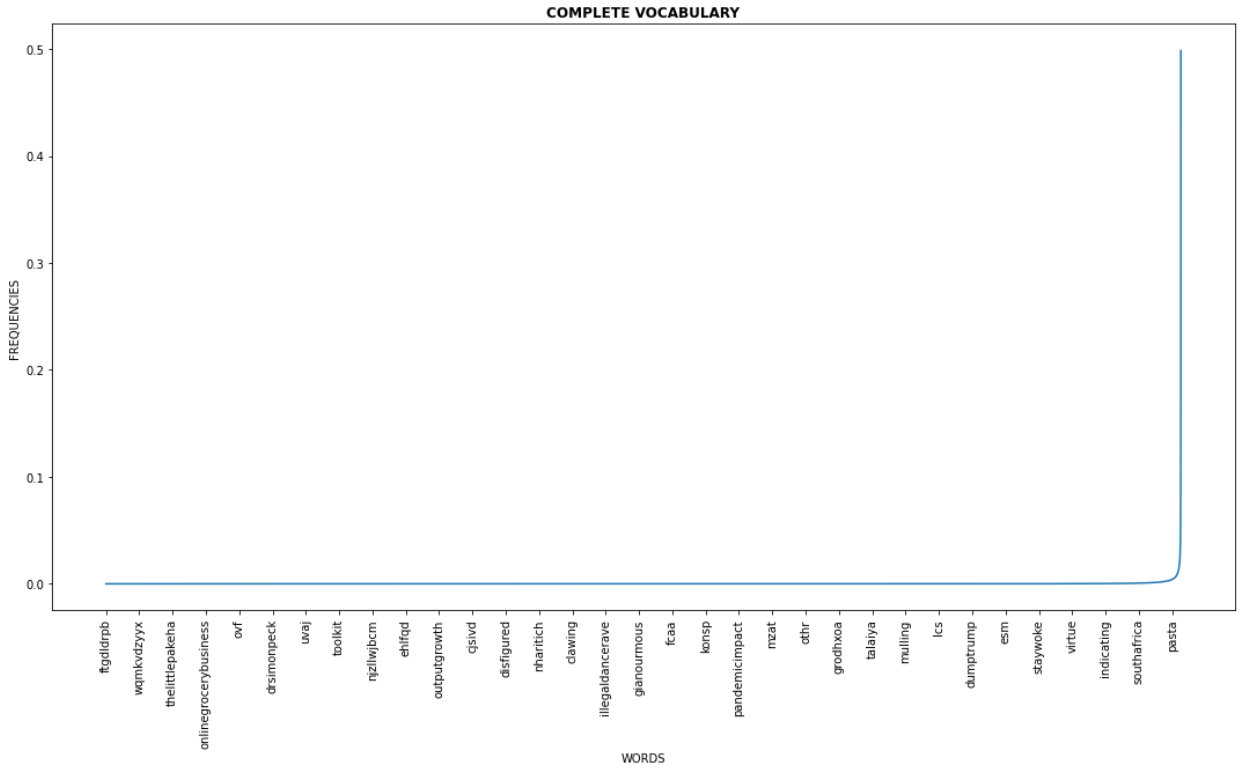
Regarding the number of words I would choose for my term-document matrix, it would be a difficult choice. There is a large number of tweets in this document, and there are not very many words that appear in a lot of them. The term that appears in the most number of documents has a frequency below 0.5, so even the most important word does not appear in the half of the documents. For sure I would choose a small number of terms (between 10 and 20) to form this term-document matrix.

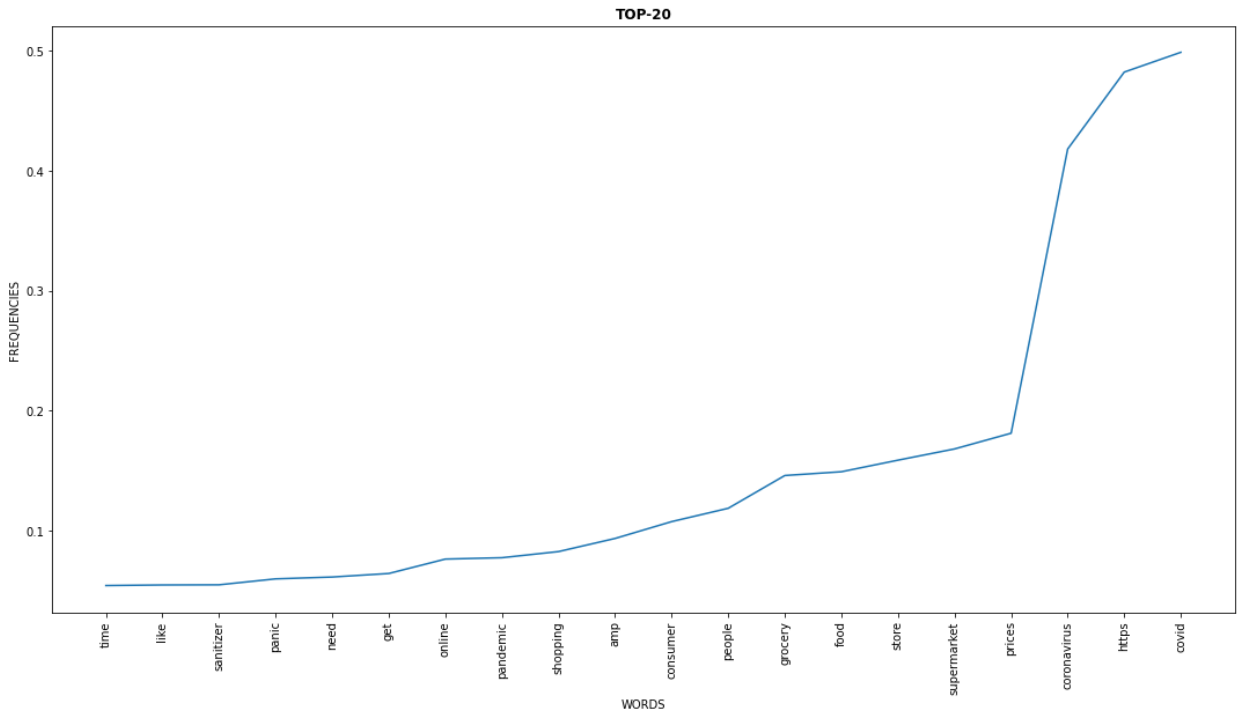
If we analyse a little further the frequencies, there are only 10 words that have a frequency above 0.1, so these worlds would be included for sure in our matrix. The number of words with a frequency above 0.05 are 23, so maybe I would choose the 20 more explanatory words from the top-23 words to form my document-term matrix. For example, the word ‘get’ is one of the 23 words that I would not choose. Or, for example, the word ‘amp’, which is there because it is

the encoding for &, should also be removed as it is technically a stopword. So, the words used for our model should be chosen after some inspection.

Yet it is true that the final decision for the number of terms in a term-document matrix will also depend on the processing power of your laptop or computer. The largest the term-document matrix, the better would be the analysis. Therefore, if a large matrix is feasible, it will surely be better.

We plot the final result with matplotlib, resulting in the following line plots:





4. ***[10 points] This task can be done individually from the previous three. Produce a Multinomial Naive Bayes classifier for the Coronavirus Tweets NLP data set using scikit-learn. For this, store the corpus in a numpy array, produce a sparse representation of the term-document matrix with a CountVectorizer and build the model using this term-document matrix. What is the error rate of the classifier? You may want to check the scikit-learn documentation for performing this task.***

As this question can be done independently from the previous three, I will start from scratch. The aim is to construct a Multinomial Naïve Bayes model to predict the sentiment of a tweet, in other words, a multiclass classifier as we have 5 possible sentiments. This will be achieved by training our model with the tweet data we have on our dataset. So, technically, we are only interested in the columns *Sentiment* and *OriginalTweet*.

The first thing I have done in the python script has been to prepare the target. First, I have stored the *Sentiment* column in a new array, *y*. Then, I have converted this categorical array in a numerical one. I have helped myself with the *sicikit-learn* package *preprocessing*, that includes a function named *OrdinalEncoder*, which encodes an array in a given order. I have set 'Extremely Negative' as 0, and 'Extremely Positive' as 4.

After preparing the target, I have to clean and prepare the data we will feed to our model. First, I created a new dataframe called *x*, from the *OriginalTweet* column. Then, I created a new column called *ModTweet*. This column has the modified tweets.

To have a good performance with our model, we have to modify in some manner our original tweets, so they provide more information than misinformation. For example, we have got rid of the stopwords, the URL's and links, and the Twitter usernames. The whole process followed for the data cleaning has been the following:

I. Convert the tweets to lowercase, like in the first part of this coursework.
II. Remove links and URLs. This is done with the help of the *re* library, which manages really well Regular Expressions, RegEx. We have removed any word that starts with 'http'.
III. Remove Twitter usernames, which do not add much information to the model. This is done with the *re* library too, by removing any word that starts with '@'.
IV. We have also removed the non-alphabetical characters with the *alpha* function we created in the question 1.4.
V. Lastly, we have ensured that there is only one whitespace between each pair of words.

It would sound strange not to remove the stopwords, but as *CountVectorizer* does this for us, we will avoid this step.

Now, we have to create a sparse representation of the term-document matrix for the words in our corpus. A term-document matrix is a 2D matrix with the frequencies of words in a collection of documents, rows correspond to documents and columns represent terms. In our case, there will be a row for each tweet, making a total of 41,157 rows. And the number of columns will depend on how we create this matrix.

We have created the term-document matrix with a built-in function of *scikit-learn* called *CountVectorizer*. This function accepts different parameters, but I have only used two of them: *stop_words='english'* to remove the stopwords from our columns, and *ngram_range=(1,2)* to set the lower and upper boundary of the range of n-grams.

A n-gram is a contiguous sequence of n items from a given sample of text or speech. As the *CountVectorizer* function uses by default the words as the analyser, we are talking about n-grams of words. So, our model will have unigrams and bigrams. This will increase its complexity, alongside with the possibility of overfitting, but it will also have higher classification power.

We could use a range going from 1 to 5, which will give us a high accuracy. Nonetheless, it will also overfit the model with high probability. As we are not going to evaluate it against a test set we cannot know this for sure, but it makes sense. Increasing the upper boundary of the n-grams will fit the model strictly to the training data we have. Using common sense, bigrams can be very descriptive, and trigrams are harder to be repeated from one tweet to another, but 5-grams are almost impossible. That is the main reason of why we are going to use unigrams and bigrams and avoid trigrams, to avoid overfitting.

Once we have created our sparse term-document matrix, we are in position of training our model. We have created the *CountVectorizer* object by passing the *ModTweet* column to our vectorizer, storing it in an array called *X*. As we said before, this array will have 41,157 rows (one for each document/tweet). Regarding the number of columns, it will have one per each unigram or bigram present in our corpus. In our case it will contain 946,195 columns. We are talking about a 2D matrix with $41157 * 946195 \approx 39$ thousand million values, yet it is true that most of them are 0.

As a side note, I will say that this is the main reason of why NLP is so damaging for the environment. They need to use immense quantities of power because of their complexity. Normally, NLP models are done with Neural Networks, models that contain several (of the order of millions) neurons/parameters. Depending on where the processing power of these deep learning algorithms need is consumed, it can be detrimental for our environment, speeding the climate change [2].

When we create the model, which we have named *nb* in representation of Naïve Bayes, we need to pass a value for the *alpha* parameter. This parameter represents the additive (Laplace) smoothing parameter, 0 is for no smoothing, and 1 is the maximum. Smoothing is the process of assigning a non-zero probability to sequences that do not appear in the training corpus. As we have all the tweets in our training corpus, we do not need to set *alpha* to a high value. We will set this parameter to *1e-6* so it does not give us a Warning.

The steps left are straightforward, me have trained our model *nb*, with the term-document matrix *X* and with the targets of our tweets, the sentiment (*y*). Each row of the term-document matrix corresponds to the terms represented in each tweet, and each value of the target *y* is the sentiment behind each tweet.

Once the model is trained, we have predicted the sentiments of each tweet/document in our dataset applying the Multinomial Naïve Bayes model. When we compare it with the actual targets we can observe that our model performed really good, obtaining an accuracy of 99.37%.

Obviously, this score is not very representative, as the model learned from those same instances. We could obtain an even higher accuracy if we had used *ngram_range=(1,5)* for example, but we have set the model to unigrams and bigrams to mimic how we would have faced up a real-world problem.

[2] https://medium.com/@julian.harris/deep-learning-natural-language-processing-and-climate-change-a62cfebcc26f

BRIEF COMMENT ON THE OBTAINED RESULTS:

We have faced a sentiment analysis problem by using NLP. This type of data is quite tricky as it is very hard to process in acceptable times. We have benefitted ourselves with the vectorized properties of pandas and NumPy, which has eased the analysis greatly.

The execution times for each section of the first part of this coursework are the following:

SECTION OF PART 1	**TIME OF EXECUTION** (seconds)
QUESTION 1 (preliminary analysis)	0.823
QUESTION 2 (data cleaning)	4.519
QUESTION 3 (frequency histogram)	2.675
QUESTION 4 (Multinomial Naïve Bayes model)	8.064
TOTAL EXECUTION TIME	16.201

These execution times are the ones obtained when executing the whole script while writing the report. They have to be analysed carefully as I have obtained several different execution times, these ones have been some of the best I have obtained, if not the best. I have got times of execution that go from 15 seconds to 30 seconds or a little bit more, it depends on a lot of external factors. Like executing the program in Spyder, an environment for scientific programming, or directly in the terminal; or executing the program while other tasks are being performed. Anyway, I believe this is an acceptable execution time, and I am quite happy with it.

Overall, I have learned a lot of new things regarding NLP, which hopefully I will be able to give use while carrying out my final master's thesis (*The impact of gender bias on models learned from labelled electronic health record text*).

On the following bibliography I have listed some articles that have been of help during this part of the coursework. I am leaving them here for referencing purposes and to have them localised somehow for the future. The articles are the following:

- https://towardsdatascience.com/sentiment-analysis-of-tweets-using-multinomial-naive-bayes-1009ed24276b
- https://towardsdatascience.com/sentiment-analysis-introduction-to-naive-bayes-algorithm-96831d77ac91
- https://towardsdatascience.com/text-analysis-basics-in-python-443282942ec5
- https://medium.com/@julian.harris/deep-learning-natural-language-processing-and-climate-change-a62cfebcc26f
- https://towardsdatascience.com/very-simple-python-script-for-extracting-most-common-words-from-a-story-1e3570d0b9d0
- https://towardsdatascience.com/introduction-to-nlp-part-1-preprocessing-text-in-python-8f007d44ca96

PART 2: IMAGE PROCESSING

This second part of the coursework involves images and image processing operations. A good definition for image processing is given in the lecture slides: methodology with efficient algorithms and systematic approaches for enhancing and extracting information from images.

As each section of this part of the coursework uses a different image, we will introduce each image in each respective question.

1. ***[8 points] Determine the size of the avengers imdb.jpg image. Produce a grayscale and a black- and-white representation of it.***

The image we are going to use is at the left side of this text. It is the actual movie cover of Avengers¸ the blockbuster produced in 2012 by Marvel and directed by Joss Whedon [3]. As we can see, it has some great actors on the front, like Robert Downey Junior, Scarlett Johansson or Samuel L. Jackson.

Images in *scikit-image* (open-source image processing library for the Python language) are stored as *scipy.ndimage* objects. This is very similar to *NumPy* arrays. So, when the question asks us to determine the size of the Avengers movie cover, it is asking us for the shape of the array.

After reading the image via *imageio* (Python library that provides an easy interface to read and write a wide range of image data, including images, animated images, videos, etc.), we are in position of obtaining this information. The shape (size) of this movie cover is **(1200, 630, 3)**.

This means it is a digital image with RGB format. This format has each pixel value distributed equally between 3 different colours: 8 bits for red (R), 8 bits for green (G) and 8 bits for blue (B). That explains why this image is composed of three 2D $matrices$, each one of this matrices encodes the intensity of each colour (I_R, I_G and I_B).

The next task of this question is to produce a grayscale and a black-and-white (binary) representation of this image. A grayscale representation has each pixel value between 0 (black) and 255 (white). It can be obtained easily applying the following formula:

$$I_{gray}(x,y) = \frac{w_R \cdot I_R(x,y) + w_G \cdot I_G(x,y) + w_B \cdot I_B(x,y)}{w_R + w_G + w_B}$$

Where w_i is the weight associated to each 2D matrix. These w_i can be distributed equally to each matrix (0.33, 0.33, 0.33), or they can be set to (0.3, 0.59, 0.11), which are computed based

[3] https://www.imdb.com/title/tt0848228/

on their respective wavelengths. All the same, we will use the *skimage.color.rgb2gray* function, which handles this for us.

The last step of this question asks us to represent the movie cover into a black-and-white representation. In this type of representation, also known as binary, each pixel value either is 0 (black) or 1 (white).

So, to convert a greyscale image into a black-and-white one we only have to apply a threshold to the 2D matrix of the greyscale. Therefore, the equation we will apply to the pixels in the grayscale format is the following:

$$I\_binary\ (x,y) = \begin{cases} 1\ if\ I_{gray}(x,y) \geq T, \\ 0\ otherwise \end{cases}$$

To estimate the threshold we will use the built-in function *skimage.filter.threshold_otsu*, which returns the threshold of a given image by applying Otsu's method [4]. This method returns a threshold value that separate pixels in two classes, foreground (white) and background (black), it is selected by maximising the inter-class variance.

The two images above correspond to the grayscale (left) and the black-and-white (right) representations of the movie cover of Avengers.

2. ***[12 points] Add Gaussian random noise in bush_house_wikipedia.jpg (with variance 0.1) and filter the perturbed image with a Gaussian mask (sigma equal to 1) and a uniform smoothing mask (the latter of size 9x9).***

This question asks us to add Gaussian noise to an image to later smoothen it. This is a form of low-level image processing, applied at the pixel level. Gaussian and uniform smoothing masks

[4] https://en.wikipedia.org/wiki/Otsu%27s_method

are used for reducing noise in the image. They produce a smoothed (blurred) image, and the larger the size of the mask, the more intensive the smoothing.

The image we are going to use in the exercise is from the Wikipedia. It corresponds to a building in London, the Bush House [5], which is part of the Strand Campus of King's College London. Campus I hope to meet some day when this darn virus disappears. The image we are talking about is on the left side of the images above this text.

The first thing we have to do is add Gaussian random noise to the image with variance 0.1. This can be done easily with a built-in function from the *scikit-image* library called *random_noise. W*e only have to pass to the parameter *var* the variance we want for our image. The perturbed image is the one at the right of the original Bush House image.

Now that we have the perturbed image, we are in position of applying a mask to smoothen the image. As we said before, this is done to reduce noise from an image. We are going to compare the two different masks: Gaussian Mask and Uniform Mask, also known as Average Smoothing Mask or Box Mask.

We will start by implementing the Gaussian Mask. This is achieved by applying the Gaussian function to each *(i, j)* element of the mask, which will be then applied to the image. So, the elements of the Gaussian Mask would be the following:

$$M(i,j) = \frac{1}{2\pi\sigma} \cdot exp\left(\frac{-(i^2 + j^2)}{2\sigma^2}\right)$$

Where the standard deviation (σ) is a parameter to be chosen. It can be set either by experimentation or by applying the rule of the thumb $\sigma \approx l/3$, where l is the number of pixels in each dimension. In our case, σ is set to 1 as it is indicated in the question description. The smoothed image can be obtained easily by applying the built-in *scikit-image* function *filters.gaussian*. As we can see, the latter equation gives more weight to the pixel where the

[5] https://en.wikipedia.org/wiki/Bush_House

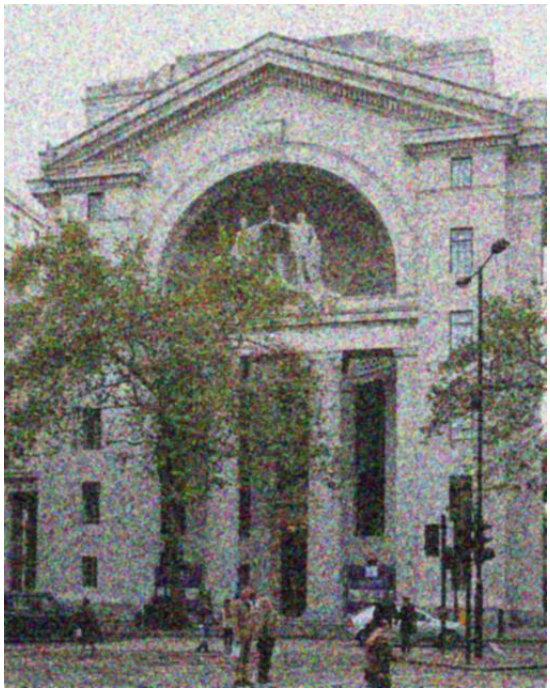
mask is being applied $(i = 0, j = 0)$, and its value is reduced as the pixel is far from the corresponding centred one.

The other smoothing mask we have to apply is know as Uniform Smoothing Mask, Average Smoothing Mask or Box Mask. This is a smoothing mask too, meaning that it is composed with positive weights that sum up to 1, similar to the Gaussian Mask.

However, this type replaces each pixel by the average of itself and its neighbours. So, having in mind we have to apply a filter of size 9×9, the mask applied to the image would have 81 values, all set to $1/81$. As it averages the corresponding pixel and its 80 neighbouring pixels, it blurs the image equally and reduces noise.

We can apply this kind of filters with a library called *SciPy* (open-source Python library used for scientific computing and technical computing). This library has a package called *ndimage* that counts with a built-in function named *uniform_filter* that carries out this type of mask. We only have to specify the size of the mask, and as it is a RGB digital image, it has three 2D matrices. Hence, we have to apply a 2D filter of size (9, 9, 1) to avoid a greyscale representation, because if not, the function will average all three colour channels, returning a greyscale image.

After applying these two types of smoothing, we can compare their results. First, we will show the images to have an idea. The image to the left is the perturbed image after applying the Gaussian Mask, and the image to the right is how it looks after applying the Uniform/Box/Average Smoothing Mask:



The main difference between one mask and the other is that the Gaussian Mask gives more weight to the nearby pixels of the pixel the mask is being applied to, while the uniform mask simply performs a uniform average to the pixel and its neighbours.

Consequently, as we can observe in the two images above, the Gaussian Mask has performed worse in eliminating the Gaussian noise created in the previous step, yet it has a better resolution.

On the other hand, as we applied a 9×9 uniform filter, the image looks much more blurry, but it has eliminated the Gaussian noise to a larger extent. If we had applied a filter of larger size, the image would look even more blurry.

3. *[8 points] Divide_forestry_commission_gov_uk.jpg into 5 segments using k-means segmentation.*

From the lecture slides, segmentation is the process of partitioning an image into multiple objects (set of pixels). It aims to detect local sharp changes in intensity. This process is useful because it simplifies and/or modifies an image into a more meaningful and easier representation to analyse.

There are multiple ways of performing segmentation on an image: Thresholding, Region-Based, Clustering-Based, Canny Edge Detection, HoughTransform, etc. For this exercise we will use the Clustering-Based approach, which works similarly to the k-means clustering algorithm we have already seen in a previous lecture. It groups image pixels based on similarity using clustering algorithms, such as k-means. This means that we also need a distance metric, which in our case will be the well-known Euclidean distance, which will be based on intensity and position.

The image this question is based on is a landscape found on the Forestry Commission organisation page in the https://www.gov.uk webpage. The image at issue is the following:

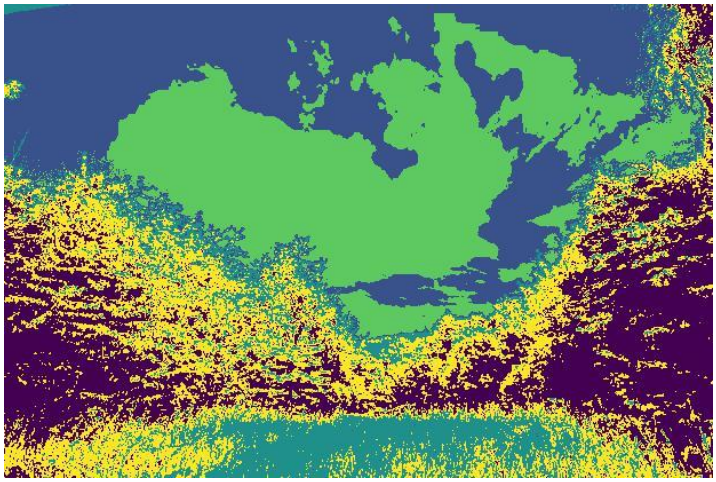
The question asks us to perform a *k-means* clustering. The output of this model will depend on some factors. The first and most important one is the number of clusters. However, this election does not depend on us as the question description requires us to set the number of segments to 5. The second factor will be the variables considered, this technique can be performed using the colour or using the colour alongside with the position. Both procedures will give different results. Therefore, we will perform both to see their differences.

The first clustering we will do will be using colour alone. For this, we have to create a dataset with *n_pixels* rows and three columns, one per each intensity of colour (red, green and blue). The *KMeans* algorithm of *scikit-learn* will directly create the clusters depending on the similarity of the colours.

To create the array we have used *np.reshape()*, which directly creates a 2D matrix of shape (n_pixels, n_colours). Now that we have the data prepared, we can train our model with these colour pixels. Therefore, the *scikit-kearn KMeans* algorithm will make a clustering and assign a class label from 0 to 4 depending on the similarity of the colours.

Once we trained our model with the pixels and the labels, we can assign each label to a pixel. Hence, we can reshape the array of the labels to obtain an image-shaped array like the one we had at the beginning. The difference in shape from the original image is that this image will not have 3 2D matrices (one per colour), it will only have one 2D matrix for all labels.

After reshaping our labels array, we can plot it. The k-means segmentation with 5 labels using only the colour looks like the following image:



For the colour and position segmentation we will make use of a package from *scikit-image*. This package is called *segmentation* and it has a function called *slic*. This function directly performs a k-means clustering based on the position and the colour. The description in the *skimage* documentation says this about the function: “segments image using k-means clustering in Colour-(x,y,z) space”.

There is a number of parameters to be specified so the segmentation is correctly done. These parameters are the following:

- *image*: input image, in our case the landscape.
- *n_segments*: (approximate) number of labels in the segmented output image. This number will be set to 5, but the algorithm might return a different number of segments depending on other parameters, like *compactness.*
- *compactness*: balances colour proximity and space proximity. Higher values give more weight to space proximity. The default value is 10, and after testing with different values, we have set this parameter to 20. This correctly differentiates the floor from the other parts of the picture.
- *sigma*: width of Gaussian smoothing kernel for pre-processing for each dimension of the image. The same *sigma* is applied to each dimension if a scalar value is passed and zero means no smoothing. After testing with different values we have set this parameter to 0.5, which correctly differentiates the sky from the trees.
- *multichannel*: Boolean parameter that indicates whether the last axis of the image is to be interpreted as multiple channels or another spatial dimension. As we have an RGB digital image, we set this parameter to True.

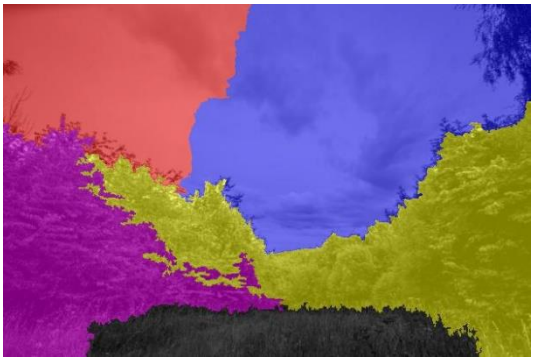
This algorithm directly returns an image-shaped array with the pixel labels, so no processing is needed neither before, nor after applying the function.

Finally, the k-means segmentation with 5 labels using only the colour looks like the following image:

As we can see in the two segmentations, the first one only distinguishes the colours in the original image, while the second one, the one above, also includes the position of the pixels in the segmentation analysis.

As these images are hard to analyse by their own, we will also show both segmentations with the original image overlapping them, so it is easier to interpret. For this part of the work we will use a package from *skimage* called *color*, which we already used for the first task of this second part. The function we will use is called *color.label2rgb*, which returns an RGB image where colour-coded labels are painted over the image.

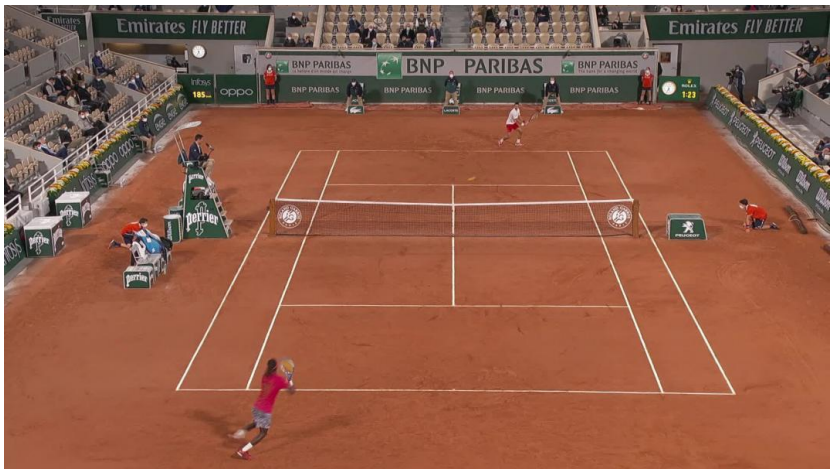
This is the final result:



4. ***[12 points] Perform Canny edge detection and apply Hough transform on rolland_garros_tv5monde.jpg.***

These two algorithms are special types of segmentation algorithms. They aim in identifying the boundaries of the objects, they are segmentation approaches based on edge detection (consecutive pixels where the intensity changes sharply).

The first thing we will do will be to show the original image we will work on. This image was taken from TV5Monde, a French TV channel that broadcasts Rolland Garros, the French Open. This is one of the four major tennis tournaments, and it is held in Paris. The image we are talking about is the following:



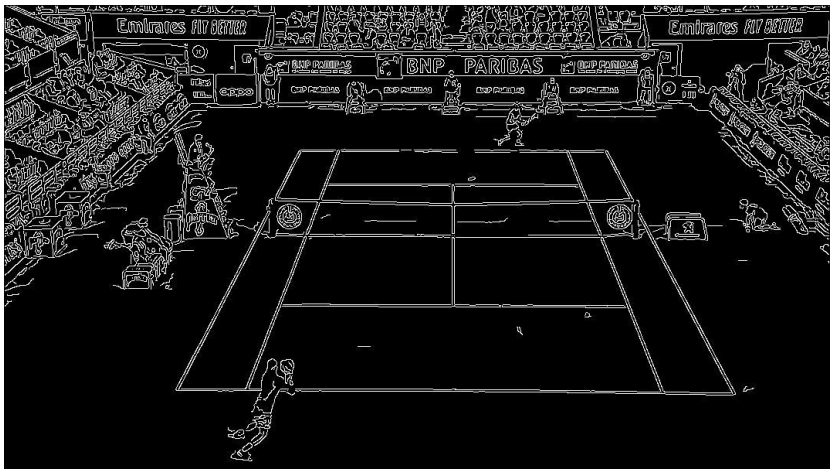
The first edge detection algorithm we will apply will be the Canny Edge Detection one. This operator uses a multi-stage algorithm to detect a wide range of edges in images. It is called like this because of John F. Canny, the developer behind this algorithm.

As we said, this algorithm follows four steps, which are:

1. Convolve with a Gaussian mask to reduce noise.
2. Compute the gradient magnitude and the gradient direction (angle)
3. Non-maximum suppression: computes a new image depending the gradient magnitude and direction calculated in the previous step.
4. Double thresholding: improves the previous image by removing detected boundaries that may not represent true edges. It reduces the number of false edge points.

There is an algorithm in the *scikit-image* library from the package *feature* that is called *canny* that does all this work for us. We will leave the *sigma* parameter as it is, 1.0, because it gives good results. However, this algorithm will convolve the image with a Gaussian mask to eliminate the noise. This is good practice and if it is not done by the algorithm, it should be done manually.

The image obtained after the Canny Edge Detection algorithm looks like this:



Now we will proceed by performing the Hough Transform algorithm on this image. This is a feature extraction technique used widely in image analysis, the purpose of the technique is to find imperfect objects within a certain class of shapes by a voting procedure.

This algorithm is applied after Canny Edge Detection because in practice, the pixels seldom characterise edges because of noise, non-uniform illumination and other effects. This is why edge detection (Canny) is often followed by linking algorithms (Hough Transform) which assemble pixels into meaningful edges and region boundaries. Therefore, Hough Transform may detect multiple occurrences of a shape in an image even if there are gaps between the edges. As a negative aspect, it can be computationally expensive, and it requires a careful selection of the size of the accumulator array.

The accumulator array is a 2D matrix that stores the values of the parameters ρ and θ, as they have bounded ranges. These parameters express the lines in an image, being ρ the distance, and θ the angle of any line.

We will apply a built-in function for this algorithm too. It belongs to the package *transform* and it is called *probabilistic_hough_line*. This algorithm has a number of parameters you can tune:

- *image*: input image.
- *threshold*: the threshold, the default value is 10, which we will leave like this.
- *line_length*: Minimum accepted length of detected lines. Increase the parameter to extract longer lines. Default value is 50, we have set this parameter to 45 (by trial and error) to distinguish correctly all the lines of the court.
- *line_gap*: Maximum gap between pixels to still form a line. Increase the parameter to merge broken lines more aggressively. Default value is 10, we have set it to 5 (by trial and error) to avoid multiple lines in the output (from the stands or from the players).

After specifying these values we end up with the following image: