Data Analytics - Τεχνικές Ανάλυσης Δεδομένων Υψηλής Κλίμακας

Project 2021-2022

ΒΑΣΙΛΙΚΗ ΜΑΥΡΙΚΟΥ
Α.Μ.: 2020514

Contents

1) Requirement 1

The requirement is related to text classification of news articles. There are 4 categories of articles: ‘Entertainment’, ‘Technology’, ‘Business’ and ‘Health’.
The dataset we are given consists of 2 files:

1. train_set.csv (111795 items) which contains labeled article samples and is going to be used as our training set.
2. test_set.csv (47912 items) which contains unlabeled article samples and is going to be used as our test set.

1.1) Question 1 - Wordclouds

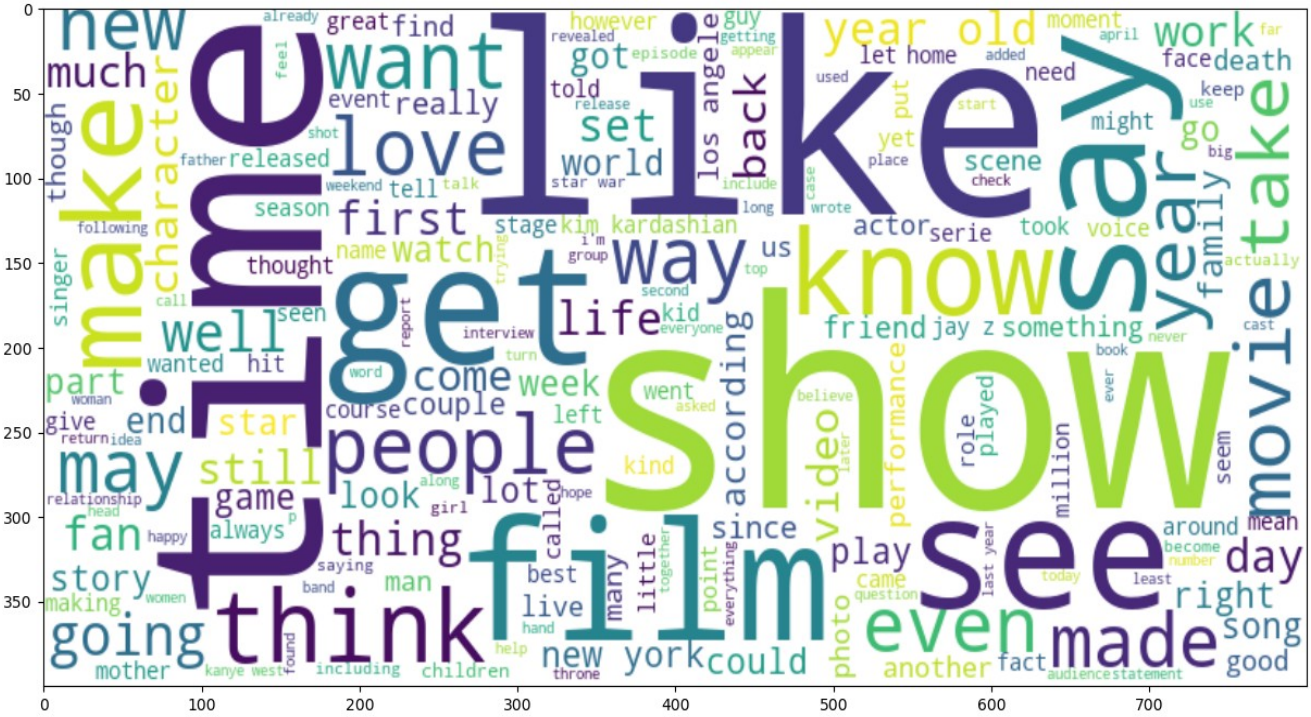
The first requirement is to create a wordcloud for each article category. We do that using our training set, which consists of labaled samples.
In order to process our data, we create a dataframe out of the content of file train_set.csv. In this dataframe, we also add an extra column ‘Text’, which contains both the title and the content of each article-sample, so that we can easily make use of both of them.
Then, we split it in four new dataframes, according to the target value of each sample, each one of them containing all samples of each category.

We create a wordcloud out of each one of those 4 dataframes, using the ‘WordCloud’ library. Before we do that, we follow some data-preprocessing steps:

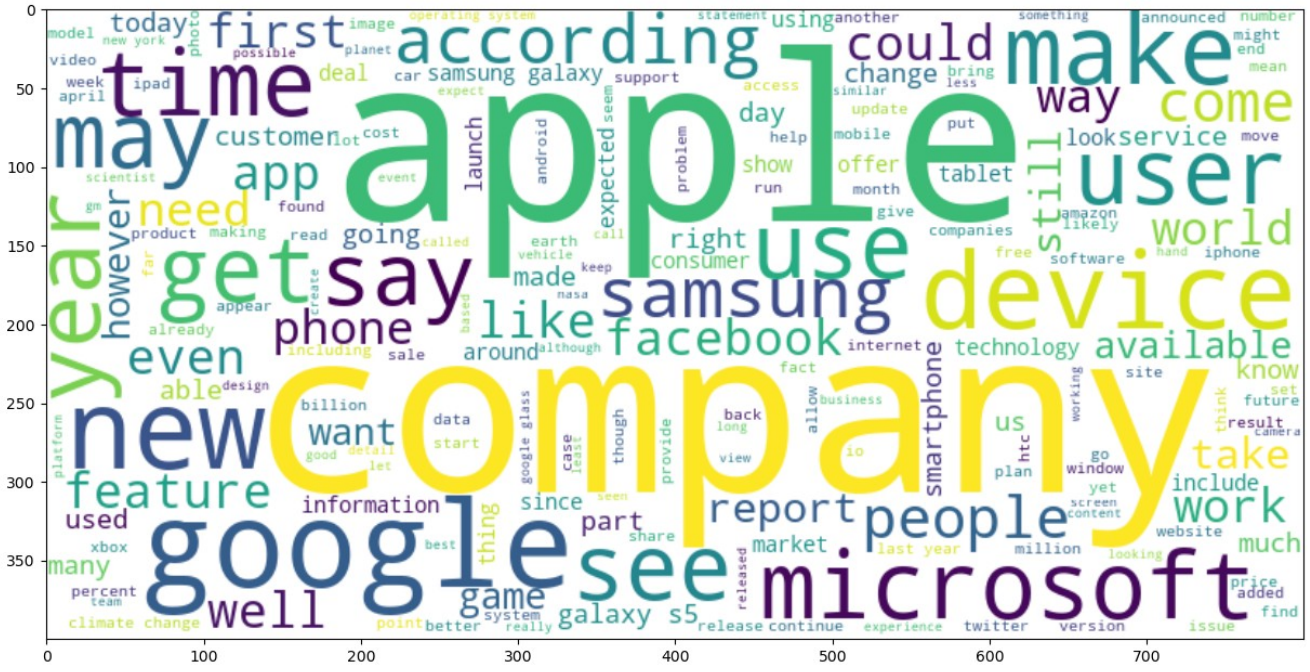
→ For each dataframe, we merge the content of column 'Text' of all of its samples, into a big string, which we then split into seperate words.
→ We define a set of stopwords (which we obtain from the ‘nltk’ library). Stopwords are words that may appear frequently in text, but dont indicate anything useful about its content, and so we remove them.
Such words in our case might be: “my”, “you”, “in”, “a”, e.t.c.
→ Finally, for each dataframe, we feed the words of column ‘Text’, minus the stopwords to WordCloud().generate() in order to get the wordclouds.

As we can observe from the pictures that follow, the workclouds we created indeed manage to provide a good general description of each category, without needing to significantly intervene to the default set of stop_words we used.

i) class ‘Entertainment’:



ii) class ‘Technology’:



iii) class ‘Business’:

iv) class ‘Health’:

1.2) Question 2 - kNN

In this question, we are asked to implement the k Nearest Neighbor algorithm, in order to classify the unlabeled samples of our test set.
We do that, using our training set, and the Jaccard similarity coefficient as our distance metric.

1.2.a) Training Model:

According to the kNN algorithm, in order to classify a new unlabeled sample:
→ We calculate its distance from each one of the samples in our training set.
→ Among all those distances, we choose the kth smallest, which correspond to the kth closest neighbors to our test sample.
→ Finally we find the class that is most common among those k neighbors, and we assign it to our test set-sample.

We are asked to use the Jaccard similarity coefficient as a distance metric.
For 2 text samples, we define Jaccard similarity as:

Jaccard Similarity = (intersection size) / (union size) = $|A \cap B| / |A \cup B|$,

where A and B are the sets of the words that appear in the texts of the 2 samples we want to compare.

In our case that distance corresponds to similarity, the closest neighbors to our test sample are going to be the train samples with which it scores the highest similarity.

1.2.b) Feature vectors:

So, in order to implement the classification, we need to be able to compare text-samples with each other. We can achieve that by finding a way to represent those text-samples as points in some space. In this manner it will make more sense to calculate “distances” between pairs of them, and define their nearest neighbors.

What we are going to do is convert the text into numerical representation, so that we will get numerical feature vectors out of each one of the text-samples in our dataset.

To do so, we are going to implement the bag of words model.
According to this, we scan through every text-sample of our training set (and only our training set), and keep each unique word found in it.

The numerical feature vectors corresponding to each one of the text-samples of our dataset are going to be like dictionaries, each one having all of these words as keys. For each sample, we assign value 1 to every key that appears at least once in the samples text, and value 0 to the keys that don't appear at all in its text.

This means that all of our samples (both those of the training, and those of the test set) will be mapped into a feature space, as arrays of '1s' and '0s' called feature vectors. The size of those feature vectors will be equal to the number of all different words that appear in all of the texts of the samples of our training set.

The good thing is that it is now possible to find the k nearest neighbors of a test-sample in a very straightforward way, and thus implement the classification.

However, the process we descrived above, usually produces quite large feature vectors, which make their storing and processing quite tricky.
In fact, when we tried to convert to numpy arrays the feature vectors we generated from our bag of words model, it immediately crashed our 16 gb RAM. Even if we managed to overcome this problem (by choosing the type of our stored values to be of type 'bool' or 'int8'), the prediction task would take days to finish.

Furthermore, we probably don't even need all of these words as features, as many of them don't indicate anything about the text's content, that would prove useful fot the classification task.

So we need to perform further preprocessing on our dataset, in order to reduce the size of our feature vectors as much as possible, and at the same time to make sure that they are comprised of features that are actually useful to us.

1.2.c) Data Preprocessing:

Text Preprocessing

First of all, for each one of our samples (of both the training and the test sets), we merge the contents of columns 'Title' and 'Content' into a new column called 'Text', so that we can easily make use of both of them.

Then for the content of column 'Text', for each one of those samples we:

→ Remove the numbers.
  Maybe the numbers shouldn't be the first thing to remove, as they might provide some useful information, but we do it anyways as it helps reduce the size of our feature vectors.

→ Remove whitespaces.

→ Convert all characters in lowercase.
→ Remove all words of less than 3 letters, as well as those included in the list of stopwords we have defined.
→ After that, we can remove all punctuation, as it won't be of much use. We perform this step only after we've removed the stopwords from our texts, because some of those words may contain punctuation, and without it, our program won't be able to track and remove them.
→ Then we use a stemmer, that transforms every word into a basic form (for example words like "runs" and "running" become "run"). In that way we avoid having multiple features that actually correspond to the same word.

So we observed that the preprocessing task for the whole training and test set was quite time-consuming, so we parallelized it in order to help reduce the total running time of our code.

CountVectorizer

In order to generate feature vectors out of the preprocessed texts, we then use CountVectorizer method of sklearn library. CountVectorizer produces a feature vector for every sample in our dataset with just a few lines of code. And so we get our training and test sets, in the form of numpy arrays of "1's" and "0's" that are ready to be used for the implementation of the kNN algorithm.

Some of the text preprocessing steps we described above, and implemented before we feed our data to CountVectorizer, the method itself would perform internally while creating the feature vectors, and it would do so in an optimized way.
However, we opted to implement them outside CountVectorizer, despite the time cost, as in this way we ensured that all of the desired preprocessing steps were followed (for example there wasn't a way to implement stemming inside CountVectorizer). Moreover, in this way we also get to define the order in which those steps are executed.

In general, CountVectorizer offers a variety of parameters, that have to do both with the text preprocessing and the form of the generated output. Depending on the problem we want to solve, some of them prove to be very helpful, so it is definitely worth exploring them.

As far as text preprocessing is concerned, we made use of CountVectorizer's 'max_df' and 'min_df' parameters. Those 2 parameters are used in order to filter the words that are included as features in our feature vectors. Particularly min_df sets a minimum number (or percentage if it is float and <1) of train set samples in which a word must appear in order to be included as a feature. In the same way, max_df sets a

maximum number (or percentage) of samples in order to make sure that words that appear too often in our train set are not included as features.

What we observed was, that setting min_df to a number bigger or equal to 2 alone, had a very big impact on the reduction of the size of the produced feature vectors. In fact, all of the text preprocessing steps we followed before CountVectorizer combined, didn't manage to achieve such results.
We ended up setting min_df to 50 (and max_df=0.8, but the results weren't as impressive), which means that a word will be considered as a feature only if it appears in more than 50 different train set samples. 50 is a small number compared to the 111795 which is the total number of samples in our training set, but it manages to reduce the size of the produced feature vectors from 254639 to 16725, without reducing significantly the performance of our classifier.

The whole data preprocessing task ended up taking about 13 min:

```
Preprossing started:
2022-02-18 22:45:42.024165
shape of X_train:
(111795, 11314)
shape of X_test:
(47912, 11314)
end of data preprocessing:
2022-02-18 22:56:46.034342
```

So now that we've generated the feature vectors out of all samples in our dataset, it has become very easy to compare them using the jaccard coefficient, and we are ready to implement kNN.

1.2.d) kNN

This is going to be the most time-consuming part of the code, as we have to calculate the distances between each one of our 47912 test samples and each one of the 111795 samples of our training set. Our samples are feature vectors of size 11314.

We used the KneighborsClassifier method of scikit learn library, that also supports parallelized implementation, which we used in order to speed up the calculations.
We chose the number of nearest neighbors k to be equal to 15.

```
Data fitting started:
2022-02-18 22:56:46.034379
end of fit of training set:
2022-02-18 22:56:46.044322
0   2022-02-18 22:56:46.044666
100   2022-02-18 22:57:41.698374
```

We first feed our training set and its labels to the classifier for the fitting, and then we also feed it with the test set, in order for it to make the predictions for its labels. As we can observe, it takes approx. 1 min to make predictions for 100 test samples.

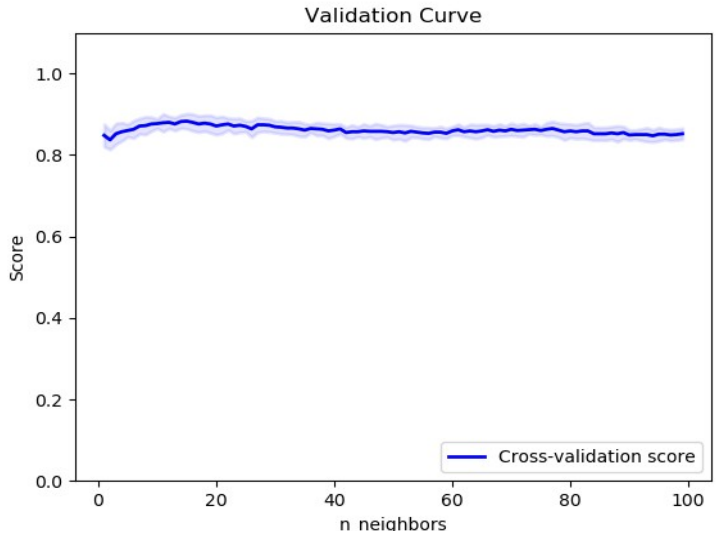
Overally, the classification task took about 8 hours. We then uploaded our predictions on the test set on kaggle, to find out that our model scored an accuracy of 95.63%.

```
47600   2022-02-19 06:30:01.634833
47700   2022-02-19 06:30:58.098263
47800   2022-02-19 06:31:54.529239
47900   2022-02-19 06:32:50.986632
```

1.2.e) Model parameters – how to improve the acuracy of our model

In order to improve the accuracy of our model, there isn't a standard procedure we follow that ensures that in the end we will definately get better resuts. We just try out different ideas, change the values of our model's parameters, and see what makes it perform better.
In our case that we have to deal with such a big dataset that it takes so long to perform calculations on, this is not always easy.
Of course we can always test the performance of our model using only a small part of our dataset, but what we find in this case, doesn't necessarily apply when we use the whole dataset as well.

For example, our model it self has only one parameter, that being the number of nearest neighbors k. We can perform cross validation in order to find the value of k (among a range of different values) that makes our model perform better, for a specific number of training samples.
As an example, we implemented cross validation in order to monitor the performance of our model for k in the range of (1,100), using a subset of 1000 samples (CV uses 90% of those for training and 10% for testing), and we present the results we got:



What we observed was:

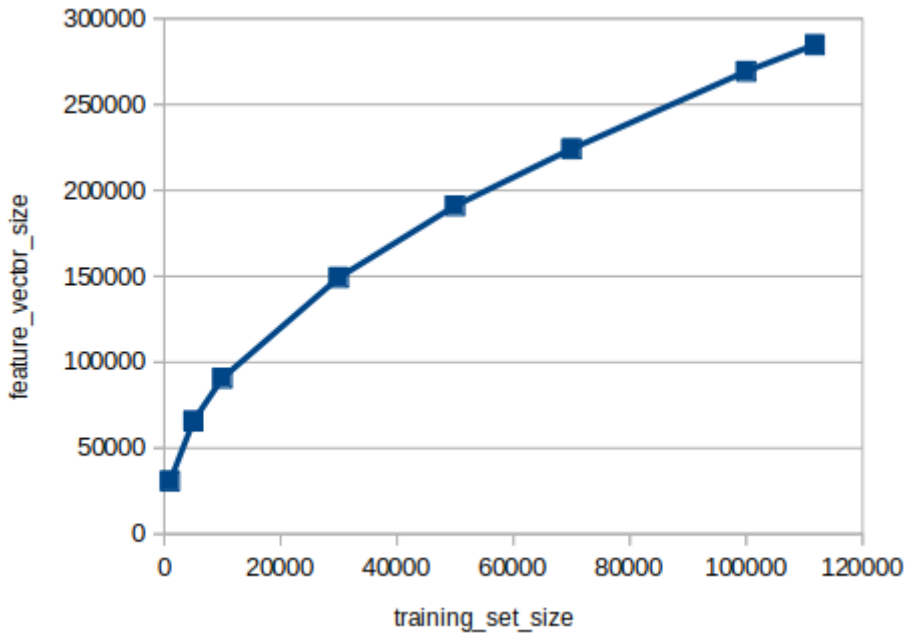
→ Using a trainset of 900 samples, the max accuracy of our model is 88.3% for k=15.

→ The performance of our model does not change more than 8% for any value of k in the range of (1,100).

However, this is not the case for other sizes of training sets. And in order to improve the performance of our model from 88.3%, we definitely need larger training sets, but implementing cross validation on them would take too long to finish.
So we just keep k=15, knowing that it probably isn't the optimal value for this parameter.

Another thing we should consider is the size of our training set. Our training set is the source of our features, and so its size defines both the size of our feature vectors and the generalization ability of our model (more training samples give us a better model).

We have a very big training set, so if we had the computational power to perform the kNN algorithm on its feature vectors without needing to apply any text preprocessing techiques on them, then we would probably get nearly optimal accuracy.
We have already seen that we dont even have to use our whole training set in order to achieve a descent performance, as with 900 training and 100 test samples and a good parameter tuning, we managed to score an accuracy of approx. 88%.
So using only a subset of our labeled samples as our training set would definitely be sufficient to score a good accuracy. The problem is that even for much smaller parts of our initial training set, the generated feature vectors are still very large to process.

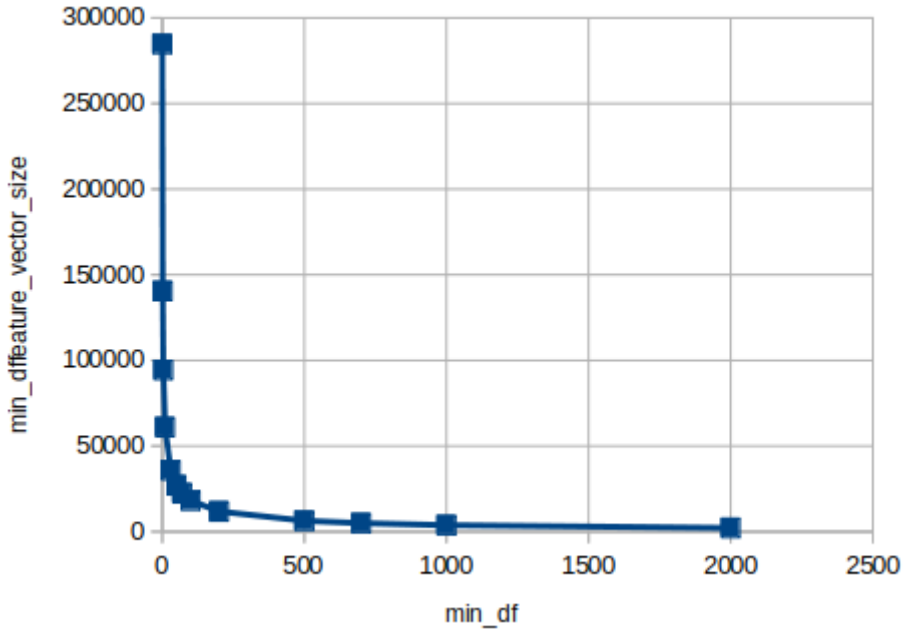


From the graph above, we can observe that even if we used about 10% of our initial training set, the sizes of the generated feature vectors would still be very large to process.

However, we still have the text preprocessing to explore. We have seen that by implementing some text preprocessing, we can reduce the size of our feature vectors and make them completely processable, even when we use our whole training set.

One problem that occurred, was that the text preprocessing phase itself took about 40 min, when we were using our whole training set. This is quite some time considering that we'va managed to reduce the prediction phase time to 8 hours. So this posed the question whether text preprocessing was indeed necessary. In order to answer to it, we also ran the kNN algorithm to make predictions for our test set, using the whole training set, but this time without the preprocessing (we only kept the min_df parameter, which, again we set to 100, in order for our feature vectors to be of processable sizes). Now the generated feature vectors were of size 16992, and so the prediction time for 100 test samples took about 1 min and 20 sec, only for our model to score an accuracy of 94.9%, which is lower campared to the one we got when followed those text preprocessing steps.
So we concluded that this text preprocessing indeed slightly improved the accuracy of our model. Also, by parallelizing it we 've managed to significantly speed up the whole procedure, so in end we decide to keep it.

As we have already seen, amongst all text preprocessing steps we 've followed, tuning CountVectorizer's 'min_df' parameter was the one to cause such radical reduction in the size of our feature vectors.



Of course removing features from our training set comes with a cost, and this is the loss of generalization ability of our model.
On the other hand adding more features, significantly increases the amount of time it takes for kNN to make the predictions on our test set.

So we need to find a balance between the size of our feature vectors and the performance of our model.
This decision may depend on various factors such as our expectations for the accuracy of our classifier, the computational resources we have in our disposal or the time we can afford to spend on running the kNN algorithm.

And that is the reason we chose min_df=100, as it ensures that we got an acceptable accuracy, with only 8 hours of running the kNN algorithm.
However, we probably could do even better for smaller values of parameter min_df (of course with longer running times of kNN), or if we conducted further inspection on combinations of other text preprocessing methods, or combinations of different training set sizes and min_df values.

2) Requirement 2

The requirement is related to sentiment analysis of IMDB movie reviews. They can be either ‘positive’ or ‘negative’.
The dataset we are given consists of 2 files:

1. imdb_train.csv (40000 items) which contains labeled reviews and is going to be used as our training set.
2. imdb_test.csv (10000 items) which contains unlabeled reviews and is going to be used as our test set.

2.1) Question 1 - LSH

In this question, we are asked to speed up the kNN classification method (for k=15), using the LSH technique, and then compare our results with those we get using the kNN classifier alone. Again we are going to use the Jaccard similarity coefficient as our distance metric.
For the LSH implementation, we use the Min-Hash LSH family, setting the number of permutations to {16,32,64} and the threshold value t to 0.8.

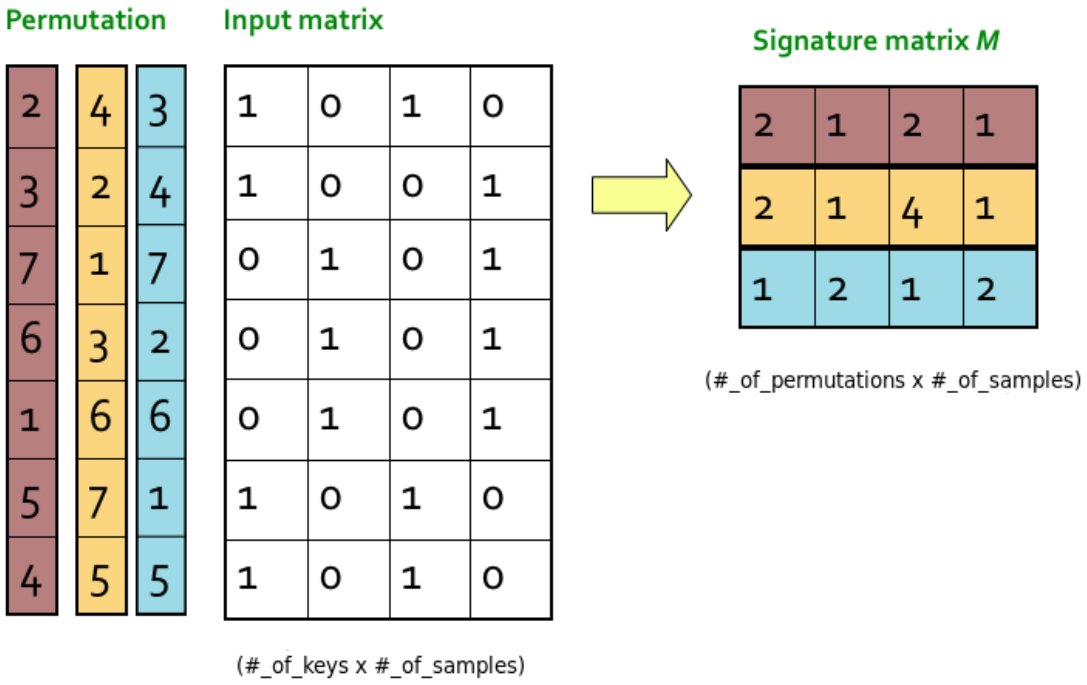
2.1.a) Training model

MinHash

Suppose we create a dictionary (a set) of all the words that appear in all of the samples of our training set, and use it in order to generate feature vectors for our samples, exactly like we did in the previous question. Again, for each one of our review-samples (both of training and of test sets), we assign value ‘1’ to every key that appears at least once in the samples text, and value ‘0’ to those keys that don’t appear at all in its text.

Then, we create a number p of random permutations of the objects of our initial set.

For each one of our feature vectors/samples, and for each permutation of the keys of our dictionary, we keep the smaller index that corresponds to a key that actually appears in the text of the sample- that is the first key of the permutation, that has value ‘1’.



So for each sample (both of training and of test sets), we get a signature H(s), of size equal to the number of permutations p we implemented on the objects of our initial set.

Now the thing is that the probability that 2 such signatures are equal one with other, is equal with the Jaccard similarity of the corresponding samples:

→ If similarity(s1,s2) is high, then probability(H(s1)=H(s2)) is high.
→ If similarity(s1,s2) is low, then probability(H(s1)=H(s2)) is low.

Moreover, the longer the signatures, the lower the error. So by using this permutations-method, we ve managed to aleviate the problem of *space complexity* by eliminating the sparseness, while *preserving the similarity*.

However, the more documents in our training set, the bigger the dictionary, and thus the higher the cost of creating permutations, both in time and in hardware.

So, instead of creating p permutations, we can just use p hash functions H on every key in our dictionary, and find a MinHash value for each function. Those values will be the elements of the signatures that correspond to each one of our feature vectors/samples.

This hash function trick is equivalent to what we ve described above. What both a permutation and a hash function do, is mapping each word in our dictionary to a different number. The difference is that in the case of the permutations, in order to generate the feature vector of a test sample, we need to scan our whole dictionary, while in the case of the hash functions, we can easily calculate its MinHash.

The only problem is that once we've generated signatures for all the samples in our dataset, we still have to compare each test-sample with every train-sample in our training set.

Locality Sensitive Hashing (LSH)

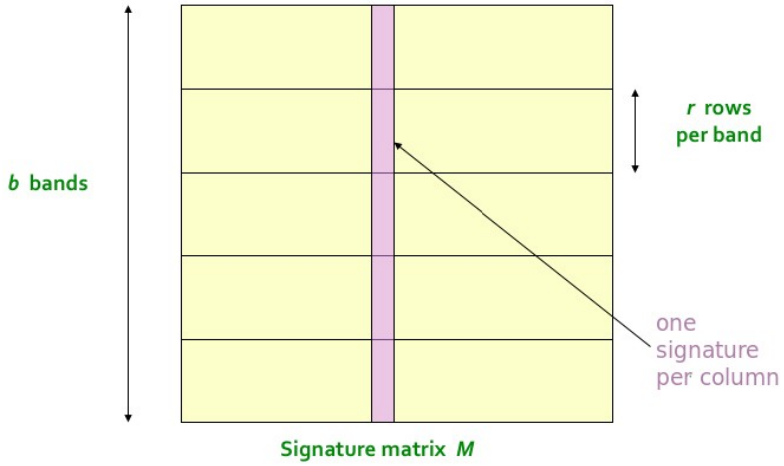
The general idea of LSH is to find an algorithm that takes as input the signatures of 2 documents, and it tells us wether the similarity of those 2 documents is equal or greater than a threshold t or not. If it is, we consider that this pair of documents is a candidate pair. Again, we do this considering that the similarity of signatures is a proxy for Jaccard similarity between their original corresponding feature vectors.

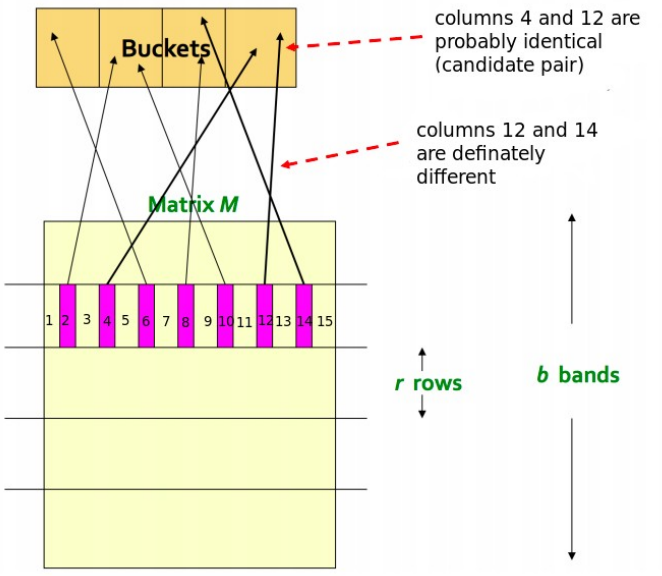
In this way, we avoid the brute force implementation that the kNN algorithm imposed, as each document in the test-set had to be compared to each document in the train-set. Instead, we identify candidate pairs of one train-set document and one test-set document where the similarity is expected to be more than a threshold t, and we only compute the actual similarity between them if the expected similarity is above that threshold. In this case, our algorithm has potential to be of O(n) complex, which is way better than $O(n^2)$ we had with kNN.

More specifically, LSH refers to a family of functions (LSH families), we use to hash our sets and create the siganture matrix we' ve mentioned above. The choice of the

hash function is tightly linked to the similarity metric we are using, and in our case that we use the Jaccard coefficient, the appropriate hashing function is MinHash.

Suppose we divide our signature matrix into b bands, so that each band will have r rows. Then, for each band we hash each column to a hash table with k buckets. Different signatures/sets correspond to different columns of the signature matrix. We can use the same hash function for all the bands, but we use a separate bucket array for each band, so columns with the same vector in different bands will not hash to the same bucket.
Sets that are similar with each other will be located in the same buckets with high probability, while sets that aren't, will be more likely to fell into different buckets, so we tune b and r accordingly. Candidate pairs of sets, are those that hash into the same bucket for at least 1 band- that is at least 1 of the hash functions we use.

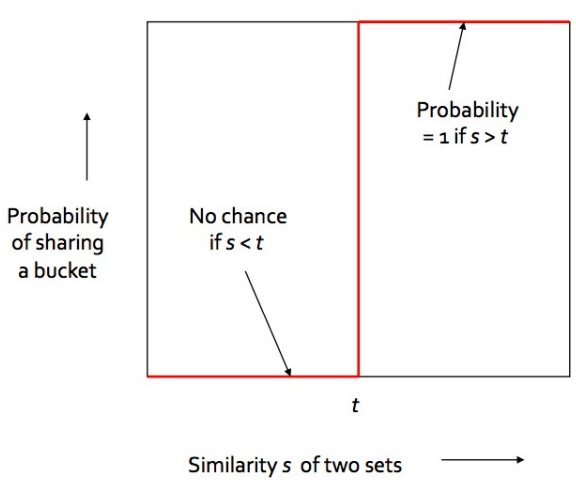


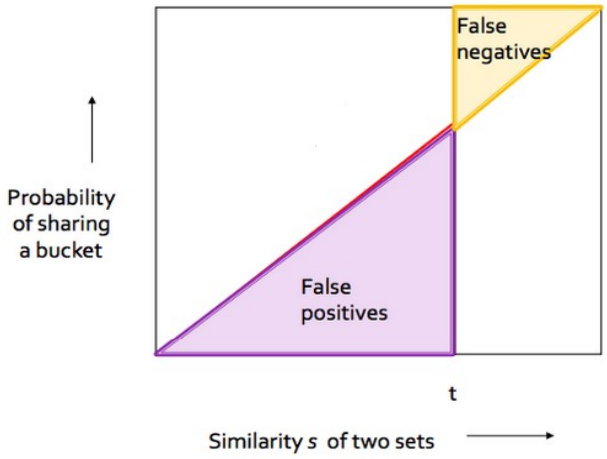


→ As we can observe r*b = p = const.

However, we cant avoid having some dissimilar pairs that do hash to the same bucket under at least one of the hash functions (false positives), as well as some similar pairs that never hash to the same bucket (false negatives).
Depending on our choices for the values of b and r, we will be having some proportion of those false positives and false negatives. Generally, higher b implies lower similarity threshold (higher false positives), while lower b implies higher similarity threshold (higher false negatives).

What we ideally want is the probability of 2 documents sharing the same bucket in at least one of the bands to be '1' if they have a similarity greater than the threshold, and '0' if they have a similarity lower than the threshold.





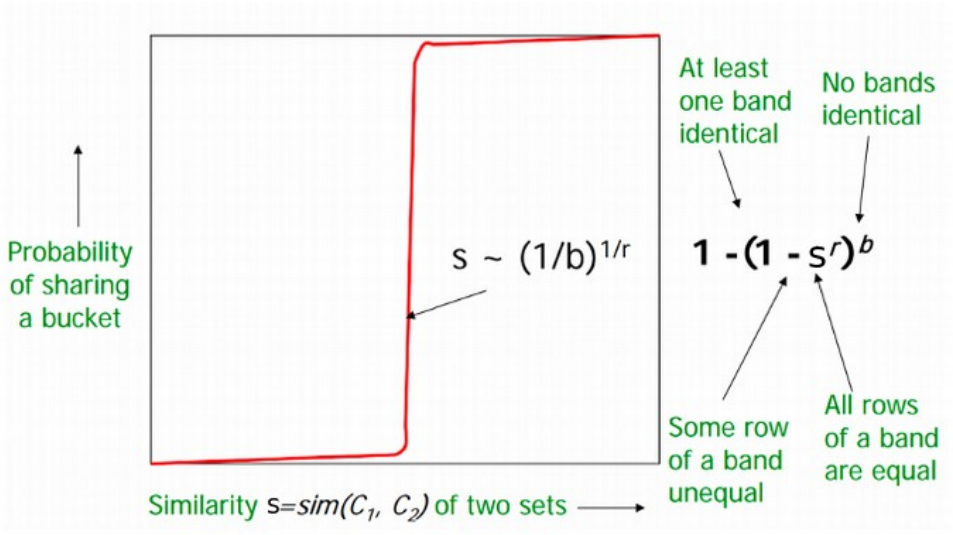
However, achieving step function is impossible in our case, so by manipulating our 3 parameters b, r and p, we try to approximate it as close as possible, so that we manage to have the minimum false negative and false positive rate possible.

Suppose that a particular pair of documents has Jaccard similarity s. Then, we know that the probability the minhash signatures for these documents agree in any one particular row of the signature matrix is also s. Also:

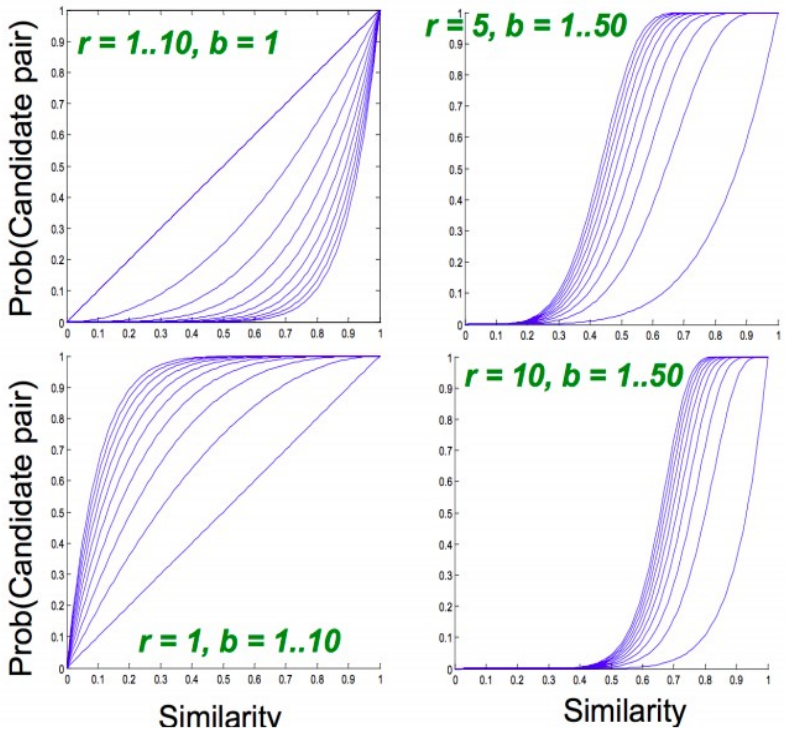
→ The probability that the signatures agree in all rows of one particular band is s^r. Therefore $(1 - s^r)$ is the probability that the signatures disagree in at least one row of a particular band.

→ The probability that the signatures disagree in at least one row of each of the bands is $(1 - s^r)^b$. Therefore, the probability that the signatures agree in all the rows of at least one band, and therefore become a candidate pair, is $1 - (1 - s^r)^b$.

→ An approximation to the threshold is $(1/b)^{1/r}$.



→ Regardless of the options of b and r, this function has the form of an S-curve.
→ The threshold is roughly where the rise is the steepest.
→ For large b and r, we find that pairs with similarity above the threshold are very likely to become candidates, while those below the threshold are unlikely to become candidates, which is exactly the situation we want.
→ Worst case will be for $b = p$.



→ If avoidance of false negatives is important, we should select b and r so that they produce a threshold lower than t. On the other hand, if speed and avoidance of false positives is more important, then we select b and r to produce a higher threshold.

→ We should also keep in mind that the bigger the number of permutations p, the more the computing power we need, as we have more hashes to calculate.

2.1.b) Feature vectors:

Once again, we want to implement the classification on a dataset comprised of text-samples. So we are going to use tha bag of words model, exactly like we did in the previous question:
We scan through every text-sample of our training set (and only our training set), and keep each unique word found in it. The numerical feature vectors corresponding to each one of the text-samples in our dataset are going to be like dictionaries, each one having all of these words as keys. For each sample, we assign value 1 to every key that appears at least once in the samples text, and value 0 to the keys that don't appear at all in its text.

However, again, the generated feature vectors are quite large to process (vectors of size 92949), so we proceed with further preprocessing on our dataset, in order to reduce the size of our feature vectors as much as possible, and at the same time to ensure that they are comprised of features that are actually useful for the task of sentiment analysis.

2.1.c) Data Preprocessing:

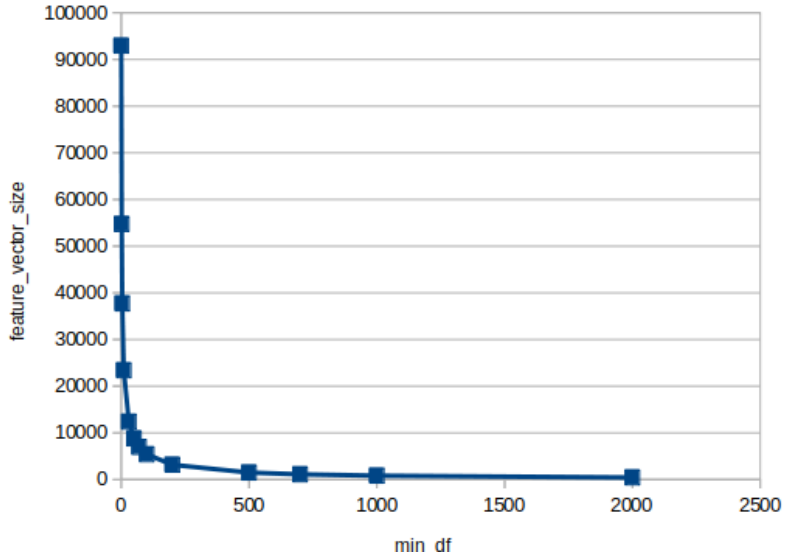
Text Preprocessing

In this question we are asked to implement sentiment analysis. We need to be careful, as some of the text preprocessing steps we followed in the previous question, where we had to classify texts based on their content, might not prove very useful here.
So, for the content of column 'review', for each one of the samples of our training and test sets, we:
→ Convert all characters in lowercase.
→ Remove all punctuation.
→ Use a stemmer. We did that, only after we ensured that this step doesn't strip our dataset of potentially useful information.

All of these steps help reduce a bit the size of our feature vectors, and increase the accuracy of our classifier, but we still need to resort to CountVectorizers min_df parameter, in order to generate feature vectors of processsable sizes, without unecessary features.

CountVectorizer

We have in mind that we need to implement the algorithm many times on our test set. So we set min_df equal to 200, because for this parameters value, we get both processable feature vector sizes and an acceptable accuracy for our model.



Appart from this, there's also one other thing we can do to improve our model:
For each feature/word that appears in a sample's text, instead of a '1', we keep the number of times it appeared in this text.
Then for each feature we calculate the average of this number, among all samples in our training set, and we keep in a matrix av[i].
We know we need feature vectors of '1s' and '0s', and so we assign '1' everywhere that a feature's appearance is equal, or more frequent than the feature's average, and '0' everywhere else.
In this way we generate feature vectors of '1s' and '0s', that also 'contain' info about the number of appearances of each word in the sample's text.
Of course in order to achieve this, we have to compare each one of our feature vectors (of both training and test sets) with matrix av[i]. But we end up keeping this step because with paralellization of the whole preprocessing task, we manage to make it taking only 3 min.

```
Preprocessing started:
2022-02-23 05:39:33.643351
shape of Xtrain:
(40000, 2742)
shape of Xtest:
(10000, 2742)
end of data preprocessing:
2022-02-23 05:43:08.088952
```

So now that we've generated our feature vectors, we can proceed to MinHashing.

2.1.d) LSH – model parameters

We are ready to hash our feature vectors using MinHah, and create the siganture matrix. We will then use LSH to identify candidate pairs of one train-set document and one test-set document where their similarity is expected to be of more than a threshold t. In this way we only compute the actual similarity between 2 documents if the expected similarity is above this threshold.
For the implementation of LSH, we keep the number of permutations equal to this we used for MinHash.
We do that for p = {16,32,64}, and we also try 3 different values of the threshold t: {0.4,0.6,0.8}, instead of just t=0.8. We do this because we noticed that for t=0.8, LSH, can't find k=15 candidate pairs for none of our test samples, and so we had to resort to finding the kth closest neighbors of each one them using the bruteforce implementation.
However, as we set t to smaller values, we notice that LSH indeed manages to find k candidate pairs for more and more of our test samples.

We also implement pure kNN, in order to compare our results with the ones we got from the LSH method. Of course in order to properly compare the running times of our algortihms, we make sure that in both implementations we follow exactly the same text preprocessing steps, and also that we don't use parallelization while implementing kNN here. We present our observations below:

	# of perm	min_df	Threshold t	Build time	Query time	Total time	LHS impl (out of 10000)	Accuracy (%)
LHS-Jaccard	16	200	0.4	3min 34 sec	3min 22 sec	6min 56sec	9922/10000	50.3
LHS-Jaccard	16	200	0.6	3min 34sec	24min 31sec	28min 5sec	2387/10000	72.1
LHS-Jaccard	16	200	0.8	3min 34sec	34min 48sec	38min 22sec	0/10000	78.9
LHS-Jaccard	32	200	0.4	4min 14sec	4min 32sec	8min 46sec	8879/10000	53.1
LHS-Jaccard	32	200	0.6	4min 14sec	34min 55sec	39min 9sec	314/10000	78.2
LHS-Jaccard	32	200	0.8	4min 14sec	35min 31sec	39min 45sec	0/10000	78.9
LHS-Jaccard	64	200	0.4	4min 30sec	17min 1sec	21min 31sec	5259/10000	63.8
LHS-Jaccard	64	200	0.6	4min 30 sec	35min 10sec	39min 40sec	0/10000	78.9
LHS-Jaccard	64	200	0.8	4min 30sec	35min 27sec	39min 57sec	0/10000	78.9
kNN	0	200	0	0	34min 44sec	34min 44sec	0	78.9

*LHS impl refers to the number of test samples (out of the 10000 in total), for which LSH indeed finds candidate pairs.

→ As the number of permutations p increases, so does the accuracy of our model, and this holds for all values of our threshold t. However, it comes with a price, as both our Build and Query times become larger as well.

→ As the value of our threshold t increases, LSH finds it harder to find k candidate pairs for each sample in our test set. For t=0.8, for all values of p, it doesn't manage to find k candidate pairs for any test sample. This means that the Query time and the accuracy we get, in all of these cases, are of the bruteforce implementation.
As the number of p, goes up, LSH manages to find less and less candidate pairs. Maybe if we had chosen a smaller number of nearest neghbors, the LSH algorithm would manage to find candidate pairs for more of our test samples.

→ As we have mentioned above, we know that $(1/b)^{1/r}$ is an approximation to the threshold. However in the documentation of the MinHash method of DataSketch library it is mentioned that there is a parameter optimization step for the values of b and r, that has to do with the chosen value for the threshold, so we decided not to mess with that.

→ The only implementations for which we indeed managed to get some results from LSH, are those where both the threshold t and the number of permutations p, are low, and these are the cases where we knew we shouldn't expect LSH to perform well (indeed in all those cases we get accuracies of approx. 50%).
In fact, during the whole process where were trying to make LSH run and produce some results, no matter what we've tried (different text preprocessing techniques, different values of min_df, different values of the threshold, combinations of all of the above, etc), we never managed to boost the accuracy to more than approx 50%.
The fact that for thresholds over 0.5, LSH does not manage to find k candidate pairs for many of our test samples definately contributes to this situation.

3) Requirement 3

In this part we are asked to compute the similarities between time series of different time resolutions, using the Dynamic Time Wrapping (DTW) algorithm.
The dataset we are given consists of 1 file:

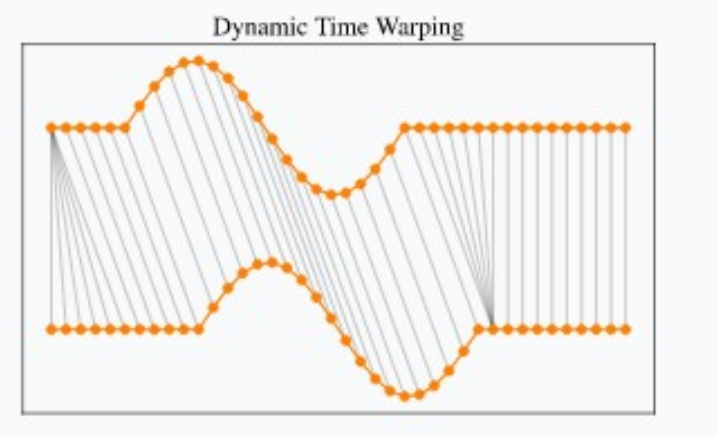
1. A test dataset dtw_test.csv (1000 items) where each row contains two time series: (a) seq_a and (b) seq_b.

The goal is to calculate the distance between the time series using DTW with euclidean distance.

3.1.a) Dynamic Time Wrapping (DTW)

A time series is a sequence of features: x = (x1, x2,…., xn), and we want to compute the similarity between two such sequences.

DTW is an instance of alignment-based metric (although, mathematically speaking, it is not a valid metric, as it is invariant to time shifts):



This means that the returned similarity is the sum of distances between matched features (matches are represented by the gray lines).
The distance associated to a match between i^{th} feature in time series x, and j^{th} feature in time series x', is $d(x_i,x_j')$.
So DTW matches distinctive patterns of the time series, according to the features values.

Let us consider 2 time series x and x' of respective lengths n and m. All elements x_i and x_j' are assumed to lie in the same p-dimensional space. The exact timestamps at which observations occur are disregarded, and we only care about their ordering. DTW seeks for the temporal alignment, that being a matching between time indexes of the two time series, that minimizes the Euclidean distance between aligned series, under all admissible temporal alignments.

We can write the above as:

$$DTW_q(x, x') = \min_{\pi \in \mathcal{A}(x,x')} \left(\sum_{(i,j)\in\pi} d(x_i, x'_j)^q \right)^{\frac{1}{q}}$$, where:

→ π: aligment path of length K. That is a sequence of K indexes $((x_0,j_0),\ldots,(x_K,j_K))$.

→ A(x,x') is the set of all admissible paths. In order to be considered admissible, a path should satisfy the following conditions:

> Beginning and end of time series are matched together:

- $\pi_0 = (0,0)$
- $\pi_{K-1} = (n-1, m-1)$

> The sequence is monotonically increasing in both i and j, and all time series indexes should appear at least once:

- $i_{k-1} \leq i_k \leq i_{k-1} + 1$
- $j_{k-1} \leq j_k \leq j_{k-1} + 1$

Despite the fact that this formulation describes a minimization problem over a finite set, the number of admissible paths becomes very large even for moderate time series lengths. Particularly, assuming that m and n are the same order, there exist $O(\frac{(3+2\sqrt{2})^n}{\sqrt{n}})$ different paths in A(x,x'), making it intractable to sequentially list them all, in order to find the minimum.
However using dynamic programming, we can find a solution to this optimization problem.

In order to do that, we use the following:

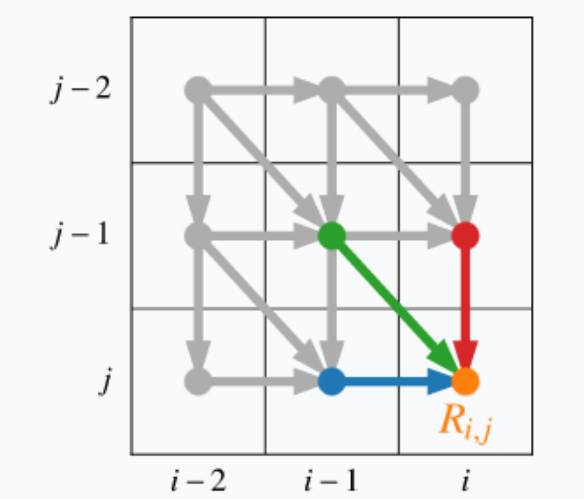
$R_{i,j} = DTW_q(x_{\rightarrow i}, x'_{\rightarrow j})^q$, where $x_{\rightarrow i}$ is time series x observed up to timestamp i.

Moreover:

$$\begin{aligned} R_{i,j} &= \min_{\pi \in \mathcal{A}(x_{\rightarrow i}, x'_{\rightarrow j})} \sum_{(k,l)\in\pi} d(x_k, x'_l)^q \\ &\overset{*}{=} d(x_i, x'_j)^q + \min_{\pi \in \mathcal{A}(x_{\rightarrow i}, x'_{\rightarrow j})} \sum_{(k,l)\in\pi[:-1]} d(x_k, x'_l)^q \\ &\overset{**}{=} d(x_i, x'_j)^q + \min(R_{i-1,j}, R_{i,j-1}, R_{i-1,j-1}) \end{aligned}$$

(*) : comes from the constraints on admissible paths π: the last element on an admissible path needs to match the last elements of the series.
(**): results from the contiguity conditions on the admissible paths. Indeed, a path that would align time series $x_{\rightarrow i}$ with $x'_{\rightarrow j,}$ necessarily encapsulates either:

- → a path that would align time series $x_{\rightarrow i}$ and $x'_{\rightarrow j}$, or
- → a path that would align time series $x_{\rightarrow i}$ and $x'_{\rightarrow j-1}$, or
- → a path that would align time series $x_{\rightarrow i-1}$ and $x'_{\rightarrow j-1}$



So, filling a matrix that would store $R_{i,j}$ is sufficient to retrieve:
$DTW_q(x,x') = R^{1/q}_{n-1,m-1}$.

3.1) DTW - Implementation

We implement the above in python code:

```
def DTW_Distance(a, b):

  DTW = np.ones((len(a)+1,len(b)+1), dtype='float') * math.inf

  DTW[0][0] = 0

  for i in range(len(a)):
    for j in range(len(b)):
      cost = calculate_euclidean(a[i],b[j])
      minimum = min(DTW[i][j+1], DTW[i+1][j], DTW[i][j])
      DTW[i+1][j+1] = cost + minimum

  return DTW[len(a)][len(b)]
```

In our implementation:

→ DTW[i+1][j+1] corresponds to $R_{i,j}$, and so DTW[i][j+1], DTW[i+1][j], DTW[i][j] correspond to $R_{i-1,j}$, $R_{i,j-1}$, $R_{i-1,j-1}$ respectively.

→ We use the euclidean distance, so q = 2.

→ The resulting algorithm is of O(mn) complexity.

It takes approx. 8 sec for our algorithm to calculate the distance for 1 tse sample. Overally, it took about 2 hours and 6 min to produce the results for the whole dataset.

```
999   2022-02-23 21:54:12.474287
1000   2022-02-23 21:54:20.325231
1001   2022-02-23 21:54:28.008706
end of calculation of distancies:
```

We uploaded them on kaggle and got a score of 3.23490.

4) References

- http://infolab.stanford.edu/~ullman/mmds/ch3.pdf
- https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134
- https://medium.com/@hubert_46043/locality-sensitive-hashing-explained-304eb39291e4
- https://en.wikipedia.org/wiki/Dynamic_time_warping
- https://rtavenar.github.io/blog/dtw.html