Course Project: Part 2

Group 9: Joel Almeida (81609), Matilde Gonalves (82091), Rita Ramos (86274)

I. Exercise 1

In the first exercise, we started by iterating over all the sentences and representing them as vectors with TF-IDF weights. Having the sentence as vectors, we were able to compute which sentences were linked and the respective graph, by considering the pairs of sentences with a similarity above a certain threshold (0.2, as default). The implementation details involved iterating over each sentence (node), computing its cosine similarity with the others, and considering as edges only those sentences that were most similar to the sentence (above the threshold). To save the node and respective edges, we used a dictionary as data structure, in which each key is the node (sentence), and the values are the edges (the sentences linked to that node). It should be noted that, for each sentence, we computed the similarity only with the sentences that follow, in order to avoid computing the cosine similarity twice . Indeed, when we find for a node x that a sentence y is similar to it, we put in the dictionary that the node x has the edge y, and, at the same time, that the node y has the edge x.

After having the undirected graph, we easily computed the page rank of each sentence, by using the equation expressed in the assignment, and, then returned the top five, in the same way as we did in the first project.

II. Exercise 2

In this second exercise, the graph is less sparse, since it considers all connections. Therefore, we decided to represent the graph as a matrix, instead of a dictionary, in order to compute page rank more efficiently. The general idea was having a matrix of probabilities, the stochastic adjacency matrix M and the probability distribution r, so that page rank can be efficiently computed by simply doing r=Mr; or, in our case considering jumping to some random page

$$r = ((d)[\frac{1}{N}]_{\text{NxN}} + (1-d)M)r \tag{1}$$

In terms of implementation details, we started by iterating over the files, in which we represented the sentences as vectors, removing all stop words. We use these sentences to compute the dictionary graph and page rank of the exercise 1, as well as, to compute the matrix graph and the improved page rank of exercise 2. For creating the matrix graph, we put, for each node (row), the cosine similarity of that sentence with the others. We performed this operation using an auxiliary triangular matrix, since the graph is unidirectional and, consequently, the matrix is symmetric. After having the matrix with the respective weights, we checked if there was a sentence that had zero cosine similarity with all the other sentences. In fact, if there is a row/column with only zeros, page rank will not be a probability distribution anymore (it will not sum up to 1); the Markov process will not converge, since the graph is not fully connected. Therefore, we remove those nodes that are not linked to any other.

After having the graph, we computed the respective stochastic adjacency matrix M (dividing all columns by the sum of its values), and we also computed the matrix of priors. Having these two matrices, we calculated the page rank with the equation mentioned earlier (1).

We have considered different priors and edges weights in our experiments. We implemented 4 different variants for the edges weights. The first variant was using the cosine similarity of sentences represented as TF-IDF; the second one used BM25 instead; and, for the third and fourth variants, the vector space representations of the sentences considered not just individual words, but also bi-grams, in which we leveraging again with TF-IDF and BM25, respectively. We have also thought about having weights based on the entities shared between the sentences (ex:names of persons, organizations,etc). However, when implementing this, we saw that most sentences do not have entities, so we discarded this idea.

Noticing that the first approach (uni-grams TF-IDF weights) achieved the best result, which was already better than the first project, we then focused in finding interesting and creative variants for the priors, implementing 8 different possibilities. We first considered all the priors suggested in the assignment (position, cosine similarity and Naive Bayes). For the priors based on the position of the sentences in the document, we gave to first sentence the highest value, which corresponds to the number of sentences N, and then give to the second sentence N-1, to the third N-2, and so on. As to the priors weights based on the cosine similarity towards the document, we decided to leverage with TF-IDF, since we had seen that TF-IDF was having slightly better results than BM25, when we did edges weights. Finally, for the priors based on the Naive Bayes, we calculate P(D/S)=P(D)*P(S/D)/P(S), simplified as P(D/S)=P(S/D) (by assuming independence of terms and by consider the same prior).

After implementing the priors suggested in the assignment, we invented 5 variants that we considered relevant. The first prior that we thought was the length of the sentence, which lead to a better MAP than the ones suggested in the assignment (see Table I). After we thought it would be interesting to have a prior that combine two variants, and, thus, we combine the length of the sentences with its position, prior(s)=len(s)*position(s), which lead to even better results. Following the same thought, we consider combining three variants, adding tf-idf, prior(s)=len(s)*position(s)*cos(s,d). This solution had also a great performance, being just slightly lower than the previous one. The fourth variant that we implemented was based on the cosine similarity between the sentence with the most relevant sentence, instead of the document. The sentence that we selected as most relevant was the one most similar with the document. This solution was not as good as the others that we invented, but it was better than one of the assignment's suggestions (Naive Bayes). Finally, we implement prior weights based on the position of terms in the doc. The idea behind is that terms that appear in the beginning might be more important, so we give more relevance to the sentences that have more terms that appear in the beginning. In terms of implementation details,

Table I: Results with edged based on cosine similarity with TF-IDF and different priors.

	Prior	Ex2
1	Position	Map: 0.1956
2	TD-IDF	Map: 0.2030
3	Naive Bayes	Map: 0.1301
4	Length of the sentences	Map: 0.2058
5	Length of the sentences with position	Map: 0.2149
6	Lenght of the sentences with position and TF-IDF	Map: 0.2123
7	Similarity with most relevant sentence	Map: 0.1807
8	Terms position	Map: 0.2119

we iterate over all terms in the document and save the position of their first appearance. Then we give a score to each term relative to its position, specifically we give for the first term the value of the number of terms, N, and for the second term N-1, for the third N-2, and so on (same approach as we did for the prior position of sentences). Then, for each sentence, we just compute the product of its terms' scores (positions). This solution was the third best one, being just slightly worse than the first and second approach invented.

In [Table I], we can see the results of exer.2, and, for exer.1 the MAP was 0.08114, for a threshold of 0.2, increasing its results with lowers thresholds (ex:MAP of 0.1588 with threshold of 0.1).

III. Exercise 3

In the supervised learning approach, we started by iterating over all documents of the dataset TeMario 2006, to build our training dataset. We represented those documents sentences as features, using 7 out of the 8 priors that we have implemented before, excluding Naive Bayes, since we had low MAP with it. We also included a graph centrality feature (see Table I), using the improved pang rank of exercise 2. After having the sentences represented as features, in a matrix form, we then add an extra column that corresponds to the respective classes (1=in summary, 0=not in the summary).

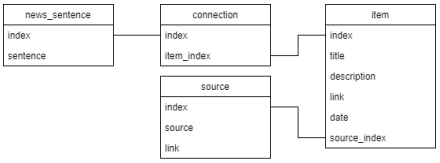
Having our training dataset, we trained it by implementing PRank algorithm, an adaption of the perceptron suited to rank. We then iterate over the docs of TeMario(2017), and represent each sentence with the same features used for the training dataset. In this way, we could rank our dataset applying the trained algorithm. Usually, the summary is built with the 5 highest score sentences, but in case of PRank, the values predicted need to be scored in descending order (lowest value w*x).

Besides the PRank algorithm, we tried several classifiers available in sklearn, such as SGD with Logistic Regression, Multilayer Perceptron, AdaBoost. For preprocessing we used Principal Component Analysis, increasing our MAP in each classifier. Then, since we have an unbalanced dataset (less positive classes), we tried oversampling the positive values, but did not get much greater results. After, we used feature selection, we found that PRank scored the highest with MAP=0.217, using only feature 7 (see Table I). The second best algorithm was SGD with Logistic Regression with MAP=0.214 and features 1, 2, 6 and 7. AdaBoost was able to get a MAP=0.212 using only the feature 6.

IV. Exercício 4

We collect the respective news using Python's urllib to get the RSS' XML and ElementTree to parse the XML items. When parsing the XML items, we used four main variables to store our items in a way that makes it straightforward to then access a complete news article from a sentence summary: sources, news_sentences and items. The algorithm then: iterates through each news source, storing it in the sources variable; iterates through each item in the source; separates the title and the description in sentences and adds them to the news_sentences variable. Both are stripped from their HTML tags using BeautifulSoup4 so we can get better results when applying the scoring algorithms; creates a new instance in the items variable, with the title, description, link, date and the index to the current source; creates multiple connection instances for each news_sentences that were previously added, containing the index for the item we created. The relations between the four variables can be easily understood in Figure 1.

Figure 1: Relationships between the variables in the algorithm



Then we applied the extractive summarization method of Exercise 2. Having the results, we applied the sorting algorithm using the news_sentences variable, obtaining the 5 top scored news. We could then easily obtain the original news item by using the index of the sentences returned and the connections variable. To generate the HTML website, we opted by using a template file with a custom tag inside (%CONTENT%). We generate HTML code for a table that contains the top scored news, load the template file and replace the tag with the generated code, saving the new file with another name. This mechanism makes it easy to change HTML elements (like style) without having to write it in Python, while also making it more readable. In the page we decided to show the news sorted by relevance, showing the sentence chosen, the source (with a link to the respective RSS XML), and the original Title, Date and Description. The titles can be clicked to access the original news article. Finally, we added an explanation for the user to know why those specific sentences were chosen, and a custom CSS stylesheet to make the website presentation more pleasant (see Figure 2).

Figure 2: Screenshot of the HTML summary page