Information Prossessing and Retreival
Keyphrase Extraction – Part 2
Group 22
João Coelho – 86448
Pedro Gonçalves – 83651
Marcos Barata - 83511

Exercise 1

For this exercise, we used a PageRank-based method to extract keyphrases. We started of with a document about Google acquiring Kaggle (Ex1/original.txt). We preprocessed it by lowercasing and removing stopwords (Ex1/preprocessed.txt). After spliting the document in sentences, unigrams, bigrams and trigrams of the preprocessed document were considered as keyphrase candidates for the document, and were added as graph nodes (Note that the ngram generator strips punctuation). Then we computed unique pairs of candidates (since the graph will be undirected, the pairs [X, Y] and [Y, X] were considered the same), and if a pair co-occurred in the same sentence, we added it as an edge with no weight – since we want an unweighted graph. For this exercise, we didn't consider self-links if a candidate occurs twice or more in the same sentence, but we will test how that influences the results on Exercise 2. We used the NetworkX package implementation. We wanted a residual probability of 15% (probability of randomly jumping to any other node), so we initialized the damping factor at 85%, which is the default value for the implementation we used. After running 50 iterations, the algorithm returned the following 5 keyphrases: "kaggle", "de", "google", "to" and "data". The first and the third can be considered valid keyphrases. The fifth is also somewhat correct, but the bigram "data science" which includes it would be more accurate. The remaining two are not what we are looking for. Also, note that only unigrams were returned by this naive approach, therefore some tunning must be made in order to get better results.

Exercise 2

This exercise aimed to test the naive approach, and try to improve it by adding edge weights and prior probabilities. We used the same dataset as for the first part of the project (DUC-2001), which consists of 308 files (Ex2/Dataset_as_txt) with a golden set of keyphrases (Ex2/Keyphrases_goldenSet). Some things were reused from the first part of the project, such as the parsing of the golden set json to files. We implemented 3 different ways of computing **edge weights**:

1-Based on word embeddings: The more similar two candidates are, the lower the weight of their connecting edge. This helps the algorithm converge on candidates that have similar meanings. We used a *spacy* model with 1M words. For candidates that are bigrams or trigrams, the *spacy* implementation for computing its vectors takes the average of the single word vectors that make up said candidate.

2-Based on document occurencies: The more times two candidates co-occur in the same document, the lower the weight on their connecting edge. This helps the algorithm converge on candidates that have some sort of importance for the document, as they keep co-occurring. [OC]

3-Based on Jaccard distance: The higher the jaccard distance between two candidates, the higher the weight of their connecting edge. We considered a candidate to be the set of its words to compute the distance between them. This aimed to make the algorithm converge to unigrams, bigrams, and trigrams of the same word. Hopefully, combining it with TFIDF prior probability (see next), would make it converge to ngrams of important words. [JAC]

For **prior probabilities**, we implemented two ways of computing them:

1-Based on TF-IDF: IDF was computed taking into account all the files in the dataset, including the one we were testing, to avoid zero values. The ngram generator we used already returned the TF of each candidate, so we just multiplied the TF by the IDF and multiplied that for the number of words in the candidate, because, as seen in the naive approach, unigrams seemed to have some priority over the others. After having the *weighted TFIDF* value for each candidate, we normalized all the values to sum to 1, and we reached the distribution, which assigned an higher probability for the candidates with higher *weighted TFIDF* value. [TFIDF]

2-Based on candidate lenght and position: For each candidate, we multiplied its lenght by a constant inversely proportional to the index of the first sentence it appears. This gives an higher priority to longer candidates appearing in the first sentences. We normalized all the computed values to sum to 1, and we reached the distribution. The problem with this distribution is that it will have many equal values (i.e., there are many candidates of the same size that make a first appearance on the same sentence). This means that when chosing a random node to jump to, even if some have an higher probability than others, there still won't be a clear difference between the probabilities to make a distinction. [LP]

We also wanted to check the influence of the self-links mentioned on Exercise 1. So first we ran the algorithm for the naive approach with and without self-loops. Then we ran the algorithm for combinations of the aforementioned implementations of weights and priors (always returning the top 5 keyphrases). The weights based on word embeddings were taking a lot of time to compute, so we decided not to include them in the results (the implementation is still commented in the source code). Follows a table with the MAP of each approach:

Naive	Naive w/ loops	P=TDIDF; W=JAC	P=TFIDF; W=OC	P=LP; W=JAC	P=LP; W=OC
2%	1.7%	5.5%	5.7%	2.9%	3.2%

First, we noted that self-links seem to make the algorithm converge on single words that appear very often, have no meaning to the context, but aren't stopwords. That's why the MAP was lower than the simple naive. So, for the others, we used the approach

with no self-links. TFIDF priors with weights based on occurencies were the best performing method. After inspecting the returned keywords (A folder for each one of the approaches is generated, contained the keyphrases for each file: Ex2/Keyphrases_<*approach*>), we saw that they aren't that bad, they are just not what the golden set is expecting. We also noticed that there is somewhat of a *bias* towards a class of candidates: For a document, the methods return mostly one class, either unigrams, bigrams or trigrams. Using the weights based on jaccard attenuated this, and for most of the files, the top-scoring unigram was followed by a bigram containing it. Overall, we managed to improve the naive approach, which was the main objective for this exercise.
Running the code provided in Ex2/exercise_2.py compute the results for all the methods. It'll save the keyphrases in a folder, and will also serialize the page rank scores returned and save them to a file, to be used in the next exercise.

Exercise 3
For this exercise, we tested an unsupervised rank aggregation approach. We considered the following features: Pagerank Scores for the best method in exercise 2(P=TFIDF; W=OC); Weighted TFIDF; Unweighted TFIDF*****; BM25; Position in the document. After computing ranks for all this features, we made combinations of them and used a Reciprocal Rank Fusion approach, as per the statement. We used the Dataset_preprocessed folder we generated in the previous exercise. We used the serialized results of the PageRank-based method computed in exercise 2.
First we computed the results for the features used in supervised approach implemented in the first part of the project, which were weighted TFIDF and the position in the document. On the supervised approach we also considered the length of the candidate (in words) which doesn't make sense to use now, because there will be a lot of candidates with the same length, so the rank will depend on how they are sorted, which won't return reliable values.

Supervised (W_TFIDF + position + length)	**Unsupervised (w_tfidf + posisition)**
3%	6.29%

As stated in the first part of the project, the data weren't handled correctly – the classes were imbalanced. So, it is expected a lower result for the supervised approach. Also, the position feature makes a little more in the unsupervised approach, as it is a document-relative measure, i.e., it doesn't depend on the rest of the dataset.
Results for other combinations we tested:

W_TFIDF + PR_score	**BM_25** + PR_score**	**BM25** + position**	**TFIDF + PR_score**	**PR_score + position**	**Position + PR_score + W_TFIDF**	**Position + PR_score + W_TFIDF + BM25**
7.7%	6.1%	4.6%	6.4%	4.4%	7.2%	7.3%

Comparing the results, we can conclude that giving more relevance to longer candidates is good for this dataset (i.e, using weighted TFIDF while the PR model was already using weighted TFIDF for prior probabilities). This makes us think that maybe we should've "trusted" our priors more, assigning a value to the dumping factor lower than the 85%. Also, not using a graph centrality score hurts the performance. Running Ex3/exercise_3.py will only run the best scoring method (others are commented).
*****We made a distinction between weighted and unweighted TFIDF because the PageRank model scores we were using already considered the weighted TFIDF in the prior calculation.
******An interesting result that we can't explain, was that if we used |D| as the number of letters in a document, instead of the number of words, as the BM25 formula on the 1st part of the project states, the results were slightly better.

Exercise 4
For this exercise we started by coding a simple scraper that will parse Ex4/Tecnology.xml (taken from the link provided in the statement). It will save the title and description of each news article and save it in a file. Also, it can enter the link to the actual news and save the text associated with it. However, since the news texts were very large, we decided to extract keyphrases only from the title and description.
Since our dataset is now smaller, we wanted to try something with word embeddings. So, we used a RRF with features: position in sentence, unweighted tfidf, pagerank with weights based on word embeddings, prior probabilities based on length and position. We used a dumping factor of 50% in the pagerank model, to see what happens when we follow our prior probabilities more. The python code will generate a JSON file containing all the documents and its keyphrases. After running, index.html can be opened. Since we're using word embeddings, it takes a little bit of time, so we provide a JSON for the page to be tested. The provided JSON was generated with this model, but since it depends a bit on RNG, running the python code may change it.
The visualization itself is very simple, and consists of two dropdown menus. In the first one, the user can select a document. Selecting, i'll show the text associated with it, and they keyphrases bellow, ordered by descending score. On the second dropdown, the user can select keyphrases. Selecting one, i'll show all the documents that share the keyphrase, so that a "grouping by theme" can be done for documents. By clicking on a document shown by this second menu, the first menu will change and show the results for that document.
This model didn't do as well as we were expecting. However some good results can be seen, for example, for file1. Also, by choosing "google" as keyphrase, one can see that two documents share it.
(Please read Ex4/readme.txt for instructions to run. A local server is needed. In case of problems, we'll try to have the HTML running here: http://web.ist.utl.pt/~ist186448/pri-p2/)