Data Mining - Assignment 1

Group 18
Gustavo Teodoro D. Beck - Fernando García Sanz

November 9, 2020

1 Data

The data employed have been extracted from *El País*, a Spanish newspaper (figure 1). The eleventh most relevant articles of the day were gathered. The topics mainly covered the coronavirus crisis, economy, and US elections.

In order to extract these articles, the python library `BeatifulSoup` was used. By means of the class `NewsScraper.py`, data was collected, useless components of each site (e.g. advertisements) were filtered, and then, each document was saved into a different `.json` file.

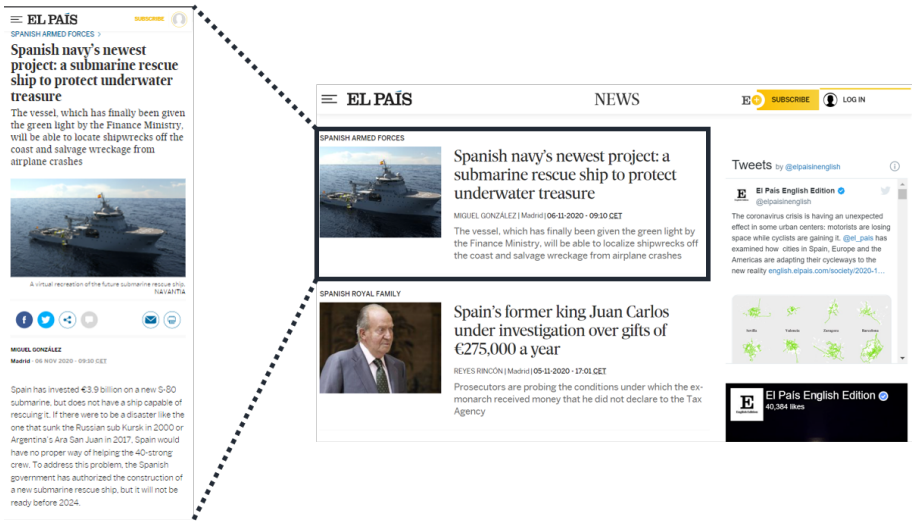

Figure 1: Document scraped from El País.

2 Methods

In order to compute the document similarity two classes have been coded: `FindSimilarItems.py`, in charge of loading the documents and executing the necessary algorithms, and `CompareSets.py`, where a `CompareSets` class is defined, implementing the necessary algorithms for document similarity computation, i.e. Shingling, MinHashing, and Locality-Sensitive Hashing.

First, in order to load the documents in proper order, a natural sorting function has been implemented, which makes use of regular expressions for this purpose.

Once all documents are loaded in the correct order, all the shingles of size $k = 5$ are extracted. The number 5 was used since this is the average length of English words [1]. At the same time, a vocabulary was created, composed of all the unique shingles of all documents together.

```python
def main():
    # ----- DEFINE VARIABLES ----- #
    json_dict = {}
    k = 5  # Shingle size
    files = sort_human(glob.glob('data/' + '*.json'))
    vocabulary = set()
    permutations = 100
    np.random.seed(42)

    # ----- IMPORT FILES ----- #
    for idx, file in enumerate(files):
        with open(file, 'r', encoding='utf8') as f:
            # Read text
            json_file = json.load(f)
            json_text = json_file['text']
            # Create and hash shingles (compression)
            shingles = [hashing(json_text[i:i + k]) for i in range(len(json_text) - k + 1)]
            json_dict[idx] = sorted(shingles)
            # Store set of hashed shingles
            vocabulary.update(shingles)

    # Sort vocabulary
    vocabulary = sorted(vocabulary)
```

Figure 2: Defining hashed shingles vocabulary.

After all, shingles are obtained, they are 32-bit hashed, reducing their representation size and ensuring uniqueness. In order to always reproduce the same hash value for debugging purposes, the python environment variable `PYTHONHASHSEED=42` is modified adding a specific seed, avoiding pure randomness.

The next step is to define a Boolean matrix composed by one-hot encoding each document based on the vocabulary and then, instantiate an object of the class `CompareSets`. This class contains three algorithms. Firstly, an implementation of the Jaccard Similarity that analyzes the similarity between the documents' shingles (figure 3) and returns an overall similarity between the documents. The developed function allows either to compute the similarity between two documents or a combination of the similarity of the whole corpus, representing it as a heat-map (figure 4).

```python
# ----- CREATE BOOLEAN MATRIX ----- #
boolean_matrix = pd.DataFrame(0, columns=files, index=vocabulary)

# One-hot encode matrix based on each document
for idx, shingles in json_dict.items():
    boolean_matrix.loc[shingles, files[idx]] = 1
boolean_matrix = boolean_matrix.reset_index(drop=True)  # Resetting hashings to ease operations

# ----- CLASS CompareSets that computes the Jaccard similarity ----- #
set_comparator = CompareSets()
set_comparator.set_boolean_matrix(boolean_matrix)
jaccard_similarity = set_comparator.jaccardSimilarity(files=files, heatmap=True)
```

```python
def jaccardSimilarity(self, doc1=None, doc2=None, files=None, heatmap=False):
    if heatmap:
        heatmap_matrix = np.ones((self.boolean_matrix.shape[1], self.boolean_matrix.shape[1]))
        for i in range(heatmap_matrix.shape[0]):
            for j in range(i + 1, heatmap_matrix.shape[1]):
                val = compute_jaccard_similarity(self.boolean_matrix, files[i], files[j])
                heatmap_matrix[i, j] = val
                heatmap_matrix[j, i] = val

        sns.heatmap(heatmap_matrix, annot=True, cmap="YlGnBu")
        plt.show()

        return heatmap_matrix

    else:
        return compute_jaccard_similarity(self.boolean_matrix, doc1, doc2)
```

Figure 3: Jaccard Similarity implementation.

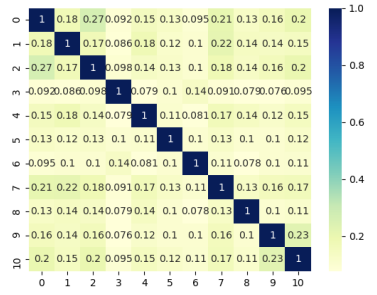

Figure 4: Jaccard Similarity corpus heat-map.

Secondly, we computed a Signature matrix based on 100 permutations of the Boolean Matrix. In each permutation, we computed a minihash function (row) that shuffled the rows of the Boolean Matrix and then computed the signature row of this shuffled Boolean matrix. This process generates a Signature matrix of size "*# permutations* x *# of documents*". Once the Signature matrix is computed we performed a signature similarity analysis that returns how similar documents are based on their signatures (figure 5). Again, our method can perform the analysis by a pair of documents or return a heat-map of all the documents' signatures similarity analysis (figure 6).

```python
# ----- MIN-HASHING ----- #
# Perform Min-Hashing and define the Signature Matrix to reduce the size of the matrix
signature_matrix = pd.DataFrame(0, columns=files, index=np.arange(permutations))  # Number of permutations x
# Number of documents
files_array = np.array(files)

for permutation in range(permutations):
    # Copying original boolean and shuffling it
    aux_matrix = boolean_matrix.copy(deep=True)
    hashed_indices = minhashfunction(aux_matrix.index, prime_module)
    aux_matrix = aux_matrix.reindex(hashed_indices).reset_index(drop=True)
    # Iterate through the rows of the now shuffled boolean matrix
    for index, row in aux_matrix.iterrows():
        # Check if all of the columns in the Signature matrix have their shingle hash,
        # otherwise check next row of the shuffled boolean matrix
        if np.count_nonzero(signature_matrix.loc[permutation].values == 0) == 0:
            break
        else:
            # Check the documents that have the current shingle
            match = np.where((row == 1) & (signature_matrix.loc[permutation].values == 0))
            if match[0] is not None:
                # Assign the shingle to the positions that haven't been assigned to any shingle before
                signature_matrix.loc[permutation, files_array[match[0]]] = index

signature_matrix = signature_matrix.iloc[::-1].reset_index(drop=True)  # Flip matrix in order
# to match the theory requirements
set_comparator.set_signature_matrix(signature_matrix)
_ = set_comparator.signatureSimilarity(permutations, files=files, heatmap=True)
```

```python
def signatureSimilarity(self, permutations, doc1=None, doc2=None, files=None, heatmap=False):
    if heatmap:
        heatmap_matrix = np.ones((self.signature_matrix.shape[1], self.signature_matrix.shape[1]))
        for i in range(heatmap_matrix.shape[0]):
            for j in range(i + 1, heatmap_matrix.shape[1]):
                val = compute_signature_similarity(self.signature_matrix, files[i], files[j], permutations)
                heatmap_matrix[i, j] = val
                heatmap_matrix[j, i] = val

        sns.heatmap(heatmap_matrix, annot=True, cmap="YlGnBu")
        plt.show()

        return heatmap_matrix

    else:
        return compute_signature_similarity(self.signature_matrix, doc1, doc2, permutations)
```

Figure 5: MinHashing implementation.

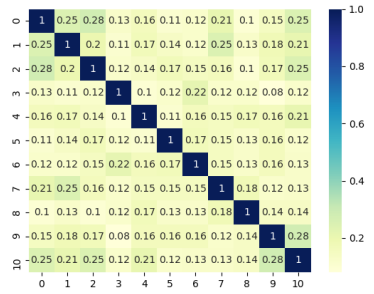

Figure 6: MinHasing corpus heat-map.

Finally, to avoid computing all the similarities of all documents, we performed a Locally-Sensitive Hashing analysis, which splits the Signature matrix into 20 bands, each band containing 5 minhash function (rows). The method compares the documents within each band and assigns similar documents of this band to a bucket of similar documents if the minhash functions between documents are similar. After, assigning documents to a bucket we compute only the similarity of the pair of documents for the same bucket. If the similarity surpasses a certain threshold (in our case 20%) the documents compared are considered similar. Our implementation (figure 7) guarantees that we don't recompute the similarities of pair if they were computed beforehand. Lastly, we present the similarities of the documents by a boolean heat-map (see figure 8), where 1 means that the documents are similar.

```python
# ----- CLASS CompareSignatures that estimates similarity of two integer vectors - minhash signatures ----- #
_ = set_comparator.lshashing(permutations, files, bands=20)
```

```python
def lshashing(self, permutations, files, bands=10, n_buckets=20):
    heatmap_matrix = np.zeros((self.signature_matrix.shape[1], self.signature_matrix.shape[1]))
    candidates = list()
    rows = int(self.signature_matrix.shape[0] / bands)
    # threshold = pow((1 / bands), (1 / rows))
    threshold = 0.2

    for b in range(bands):
        buckets = {k: [] for k in range(n_buckets)}
        band = self.signature_matrix[rows * b:rows * (b + 1)]
        hashes = [signature_hashing(band.loc[:, files[i]]) for i in range(band.shape[1])]
        modules = np.array(hashes) % n_buckets
        for idx, elem in enumerate(modules):
            if files[idx] not in buckets[elem]:
                buckets[elem].append(files[idx])
        for elem in list(buckets.values()):
            if len(elem) > 1:
                candidates.append(elem)

    for candidate in candidates:
        combinations = list(itertools.combinations(candidate, 2))
        for combination in combinations:
            i = self.signature_matrix.columns.get_loc(combination[0])
            j = self.signature_matrix.columns.get_loc(combination[1])
            if heatmap_matrix[i, j] == 0:
                val = compute_signature_similarity(self.signature_matrix, combination[0], combination[1],
                                                   permutations)
                heatmap_matrix[i, j] = val
                heatmap_matrix[j, i] = val

    heatmap_matrix = (heatmap_matrix > threshold) * 1

    sns.heatmap(heatmap_matrix, annot=True, cmap="YlGnBu")
    plt.show()

    return heatmap_matrix
```

Figure 7: Locality-Sensitive Hashing implementation.

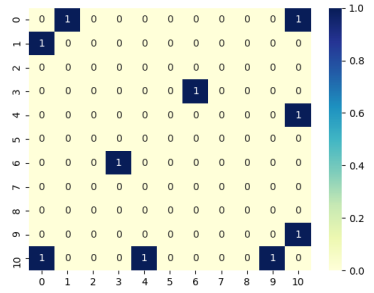

Figure 8: Locality-Sensitive Hashing corpus heat-map.

Figure 8 demonstrates that the pair of articles that are similar are: (0, 1), (0, 10), (3, 6), (4, 10), and (9, 10), which makes sense when reading them. For instance, the titles of these articles are:

- article 0: Spanish experts and authorities remain divided on home lockdown;
- article 1: Spain sets new weekend record for coronavirus cases, fatalities in the second wave;
- article 3: The Spanish youths who cleaned up after vandals protesting coronavirus restrictions;

- article 4: Spain has recorded third-highest coronavirus fatality rate in Europe since July;
- article 6: Police make 32 arrests in Madrid, after second night of disturbances across Spain;
- article 9: "State of alarm: All the latest coronavirus restrictions in Spain, region by region;
- article 10: Spain's new state of alarm: more regions close their borders.

3 Conclusions

By means of this assignment, it has been possible to see how text data can be analyzed in different ways in order to find similarity patterns between documents.

Nevertheless, the results obtained are not as good as should be expected. Our hypothesis about this lies in the following facts:

- Not cleaning the data (e.g. stemming the words) highly reduces the ratio of similarity. Stop words, special characters, and different terminations of words might suggest that the similarity index is lower than the real one.
- Added to the previous point, the fact of generating shingles employing a single-character sliding window, without previously cleaning the data, can increase a lot the ratio of difference, i.e. the same sequence of characters, but with a comma in between, generates many shingles containing this comma, that will be considered as different. For instance, comparing the phrases "Hello, Gustavo" and "Hello Gustavo" with a shingle length of 5 produces a Jaccard similarity of only 36%.

Therefore, our hypothesis states that if text cleaning and word stemming were performed, the similarity ratios will be different, and probably more realistic.

To reproduce our results, run the following code once unzipping the file *Assignment_1.zip*:

- `PYTHONHASHSEED=42 python FindSimilarItems.py`

References

[1] "English letter frequency counts: Mayzner revisited or etaoin srhldcu." `http://norvig.com/mayzner.html`.