SCIA 2023

NLP 3
LAB 01

Alexandre Lemonnier - Sarah Gutierez
Victor Simonin - Eliott Bouhana
Enguerrand de Gentile Duquesne

Promotion 2023

Contents

Part 1. Keywords Extraction

1.1 Preprocessing data

Question 1.1 Preprocessing data in a meaningful way

First we will discuss about the additional stop words that we have added to the list of stop words. They are relevant because they are common words that may not be useful for our specific natural language processing task. Stop words are common words that are generally not useful for NLP tasks because they do not convey significant meaning and are often too frequent to be informative.

```
    new_words = ["fig","figure","image","sample","using",
             "show", "result", "large",
             "also", "one", "two", "three",
             "four", "five", "seven","eight","nine"]
```

For example, words like "fig," "figure," "image," and "sample" are likely to be used frequently in a dataset containing scientific papers, but they may not provide much information about the content of the papers. Similarly, words like "using," "show," "result," and "large" may be common words that are not very meaningful in the context of our task. By adding these words to the list of stop words, you can filter them out of your dataset and focus on the more meaningful words. This can help improve the performance of our NLP model by reducing the amount of noise in the input data.

The next code defines a function called preprocess that takes in a piece of text as input and applies a series of preprocessing steps to it. The first step is to remove any punctuation from the text using a regular expression. The text is then converted to lowercase and any tags are removed. Special characters and digits are also removed using another regular expression.

The text is then converted into a list of words and lemmatized using the WordNet lemmatizer. Finally, the list of lemmatized words is converted back into a string and returned as output. The purpose of this function is to clean and normalize the input text, making it more suitable for further processing or analysis.

```
    def pre_process(text):
        # Remove punctuation
```

```python
    text = re.sub('[^a-zA-Z]', ' ', text)

    # Convert to lowercase
    text = text.lower()

    # Remove tags
    text=re.sub("&lt;/?.*?&gt;"," &lt;&gt; ",text)

    # Remove special characters and digits
    text=re.sub("(\\d|\\W)+"," ",text)

    ##Convert to list from string
    text = text.split()

    #Lemmatisation
    lem = WordNetLemmatizer()
    text = [lem.lemmatize(word) for word in text if not word in
            stop_words]
    text = " ".join(text)
    return text
```

Question 1.2 Build a top N words based on occurence

```python
def get_counter(txt_preproc, N=10):
    # Use nltk to split the text
    tokens = nltk.word_tokenize(txt_preproc)

    # Create a counter
    counter = Counter(tokens)

    # Get the top N words
    top_N = counter.most_common(N)

    return top_N
```

This code defines a function called "get_counter" that takes in a piece of preprocessed text and an optional parameter "N" (which is set to 10 by default). The function first uses the Natural Language Toolkit (nltk) to split the text into individual words (also known as tokens). It then creates a counter object using the "Counter" class from the Python collections module. This counter object keeps track of the frequency of each word in the text.

Next, the function uses the "most_common" method of the counter object to get the top N most common words in the text. The value of N is specified by the function's "N" parameter. The function then returns a list of tuples, where each tuple consists of a word and its frequency.

Finally, the code applies the "get_counter" function to the "preproc_text" column of a DataFrame called "df" and creates a new column called "Top N" containing the results. This allows you to see the top N most common words in each row of the DataFrame.

Question 1.3 What are some of the limits of raw counts? How could we improve the approach through preprocessing?

There are several limitations of using raw counts for natural language processing tasks:

- Raw counts do not take into account the length of the text: If a document is longer, it will tend to have more words and higher raw counts, even if the content is similar to a shorter document. This can make it difficult to compare the importance of words across different documents.
- Raw counts do not account for word frequency: Some words may occur very frequently in the language (e.g., "the," "a," "and"), but they do not convey much meaning and can dominate the raw counts. This can make it difficult to identify important words in the text.
- Raw counts do not consider the context of words: The meaning of a word can vary depending on the words that surround it. Raw counts do not take this context into account, so important words may be overlooked.

To address these limitations, it is often necessary to preprocess the text data before using raw counts. Preprocessing steps may include:

- Normalizing the text: This can involve converting all text to lowercase, removing punctuation, and removing common words (also known as stop words) that do not convey much meaning.
- Stemming or lemmatizing the words: This can help reduce the dimensionality of the data by reducing words to their base form, which can make it easier to identify important words.
- Weighting the words: This can involve assigning higher weights to words that occur less frequently or that occur in important contexts. This can help identify important words that may be overlooked using raw counts.

By applying these preprocessing steps, it is possible to improve the effectiveness of raw counts for NLP tasks.

Question 1.4 How can you find an optimal max_df? Why are we using a sparse matrix instead of a regular matrix?

There is no definitive answer for this question, as it depends on the data and the desired outcome. However, one method to find an optimal max_df would be to experiment with different values and see which one results in the best performance for the specific task at hand. Another method would be to use a grid search to exhaustively search for the best max_df value.

A sparse matrix is used in the TF-IDF algorithm because it is more efficient than a regular matrix when dealing with large amounts of data. A sparse matrix is a matrix that has a small number of non-zero elements. This means that the algorithm can run faster and use less memory.

Question 1.5 Find an example where there is a noticeable difference between tf-idf and raw counts? Justify which method you would choose yourself (there is no bad and good answer here)

There is no one-size-fits-all answer to this question, as the best method to use will depend on the specific data and task at hand. However, in general, we would tend to favor using tf-idf over raw counts, as tf-idf can help to downweight common words that are less informative for a particular task. This can be especially helpful when working with large data sets, as it can help to reduce the dimensionality of the data and improve the performance of machine learning models.

1.2 KeyBERT

Question 2.1 Apply KeyBERT to the a sample of the dataset

```python
# Apply KeyBERT to the a sample of the dataset
df_["Top_N_KeyBERT"] = df_["preproc_text"].apply(kw_model.extract_keywords)

df_.sample(1)
```

This code applies a KeyBERT model to the "preproc_text" column of the DataFrame and creates a new column called "Top_N_KeyBERT" containing the results. The KeyBERT model is used to extract keywords from the preprocessed text.

The "apply" method of the DataFrame is used to apply the KeyBERT model to each row of the "preproc_text" column. The resulting keywords are stored in the new "Top_N_KeyBERT" column.

Question 2.2 Comparison of multiple techniques

// TODO

1. Draw a table of the solution, the quality score that you defined and the time taken to find keywords across a sample of 1000 of the original dataset.
2. Can you think of tweaks to reduce time to compute? If yes, add an additional column to the above table with your proposed tweaks.
3. Based on the above table and lecture 1, what do you think is the most appropriate solution for keywords extraction? Why?

Part 2. Word Vectors

Question 2.1 Implement distinct_words

```python
def distinct_words(corpus):
    """ Determine a list of distinct words for the corpus.
        Params:
            corpus (list of list of strings): corpus of documents - eg [["
hey", "I", "am", "toto"], ["hey", "I", "am", "tata"]]
        Return:
            corpus_words (list of strings): sorted list of distinct words
across the corpus
            num_corpus_words (integer): number of distinct words across the
corpus
    """
    corpus_words = []
    num_corpus_words = -1

    distinct_words = sorted(list({word for doc in corpus for word in doc}))
    num_distinct_words = len(distinct_words)
    return distinct_words, num_distinct_words
```

This code defines a function called "distinct_words" that takes in a corpus of documents as input and returns a sorted list of distinct words across the corpus as well as the number of distinct words.

The corpus is a list of lists of strings, where each inner list represents a document and consists of the individual words in that document.

Next, the function uses a set comprehension to create a set of all the words in the corpus. A set is used to remove any duplicates, so that only distinct words are included in the set. The set is then converted back into a list and sorted. The number of distinct words is calculated by finding the length of the list.

Finally, the function returns the sorted list of distinct words and the number of distinct words as output. This function can be used to identify the unique words in a corpus and to understand the vocabulary size of the corpus.

Question 2.2 Implement compute_co_occurrence_matrix

```python
def compute_co_occurrence_matrix(corpus, window_size=4):
    """ Compute co-occurrence matrix for the given corpus and window_size (default of 4).

        Note: Each word in a document should be at the center of a window. Words near edges will have a smaller
              number of co-occurring words.

              For example, if we take the document "<START> All that glitters is not gold <END>" with window size of 4,
              "All" will co-occur with "<START>", "that", "glitters", "is", and "not".

        Params:
            corpus (list of list of strings): corpus of documents
            window_size (int): size of context window
        Return:
            M (a symmetric numpy matrix of shape (number of unique words in the corpus , number of unique words in the corpus)):
                Co-occurence matrix of word counts.
                The ordering of the words in the rows/columns should be the same as the ordering of the words given by the distinct_words function.
            word2ind (dict): dictionary that maps word to index (i.e. row/column number) for matrix M.
    """
    words, num_words = distinct_words(corpus)
    M = None
    word2ind = {}

    M = np.zeros((num_words, num_words))
    word2ind = {word: i for i, word in enumerate(words)}
    for doc in corpus:
        for i, word in enumerate(doc):
            for j in range(max(0, i - window_size), min(len(doc), i + window_size + 1)):
                if i != j:
                    M[word2ind[word], word2ind[doc[j]]] += 1

    return M, word2ind
```

This code defines a function called "compute_co_occurrence_matrix" that takes in a corpus of documents and a window size.

The co-occurrence matrix is a numerical representation of the relationships between words in the corpus. It is a symmetric matrix with one row and column for each unique word in the corpus. The value at each position (i, j) in the matrix represents the number of times that the word corresponding to row i co-occurs with the word corresponding to column j within the specified window size.

To create the co-occurrence matrix, the function first uses the "distinct_words" function to get the list of unique words in the corpus and the number of unique words. It then initializes an empty matrix "M" and an empty dictionary "word2ind."

Next, the function fills in the matrix "M" and the dictionary "word2ind" by iterating over the documents in the corpus and the words in each document. For each word, the function counts the number of times that the word co-occurs with other words within the specified window size and updates the corresponding entries in the matrix. The dictionary is updated to map each word to its index in the matrix.

Question 2.3 Implement reduce_to_k_dim

```python
def reduce_to_k_dim(M, k=2):
    """ Reduce a co-occurence count matrix of dimensionality (num_corpus_words, num_corpus_words)
        to a matrix of dimensionality (num_corpus_words, k) using the following SVD function from Scikit-Learn:
            - http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html

        Params:
            M (numpy matrix of shape (number of unique words in the corpus , number of unique words in the corpus)): co-occurence matrix of word counts
            k (int): embedding size of each word after dimension reduction
        Return:
            M_reduced (numpy matrix of shape (number of corpus words, k)): matrix of k-dimensioal word embeddings.
                    In terms of the SVD from math class, this actually returns U * S
    """
    n_iters = 10     # Use this parameter in your call to `TruncatedSVD`
    M_reduced = None
    print("Running Truncated SVD over %i words..." % (M.shape[0]))

    svd = TruncatedSVD(n_components=k, n_iter=n_iters)
    M_reduced = svd.fit_transform(M)

    print("Done.")
    return M_reduced
```

The function uses the Truncated Singular Value Decomposition (SVD) algorithm from the scikit-learn library to reduce the dimensionality of the co-occurrence matrix. SVD is a mathematical technique that decomposes a matrix into its singular values and singular vectors, which can be used to approximate the original matrix with lower dimensionality.

To reduce the dimensionality of the co-occurrence matrix, the function first initializes an empty matrix "M_reduced" and creates an instance of the TruncatedSVD class. It then uses the "fit_transform" method of the TruncatedSVD instance to fit the co-occurrence matrix and transform it into a lower-dimensional space. The number of dimensions is specified by the "k" parameter.

Finally, the function returns the reduced matrix "M_reduced" as output. This matrix can be used as a lower-dimensional representation of the co-occurrence matrix, which may be useful for downstream tasks such as clustering or visualization.

Question 2.4 Implement plot_embeddings

```python
def plot_embeddings(M_reduced, word2ind, words):
    """ Plot in a scatterplot the embeddings of the words specified in the list "words".
        NOTE: do not plot all the words listed in M_reduced / word2ind.
        Include a label next to each point.

        Params:
            M_reduced (numpy matrix of shape (number of unique words in the corpus , 2)): matrix of 2-dimensioal word embeddings
            word2ind (dict): dictionary that maps word to indices for matrix M
            words (list of strings): words whose embeddings we want to visualize
    """

    # Create a scatter plot and plot embeddings of each word
    fig, ax = plt.subplots(figsize=(15, 10))
    for word in words:
        idx = word2ind[word]
        x, y = M_reduced[idx, :]
        ax.scatter(y, x, marker='o', color='purple', s=400)
        ax.annotate(word, (y, x), fontsize=12)

    plt.show()
```

The function first creates a scatterplot using the matplotlib library and plots the embeddings of each word in the list "words." It does this by iterating over the words in the list and using the "word2ind" dictionary to get the index of each word in the reduced matrix. It then uses this index to get the x and y coordinates of the word’s embedding from the matrix and plots these coordinates as a point on the scatterplot. The function also includes an annotation with the word next to each point.

Finally, the function displays the scatterplot using the "show" method of the matplotlib library. This plot can be used to visualize the relationships between the words in the reduced matrix and understand how the words are distributed in the lower-dimensional space.

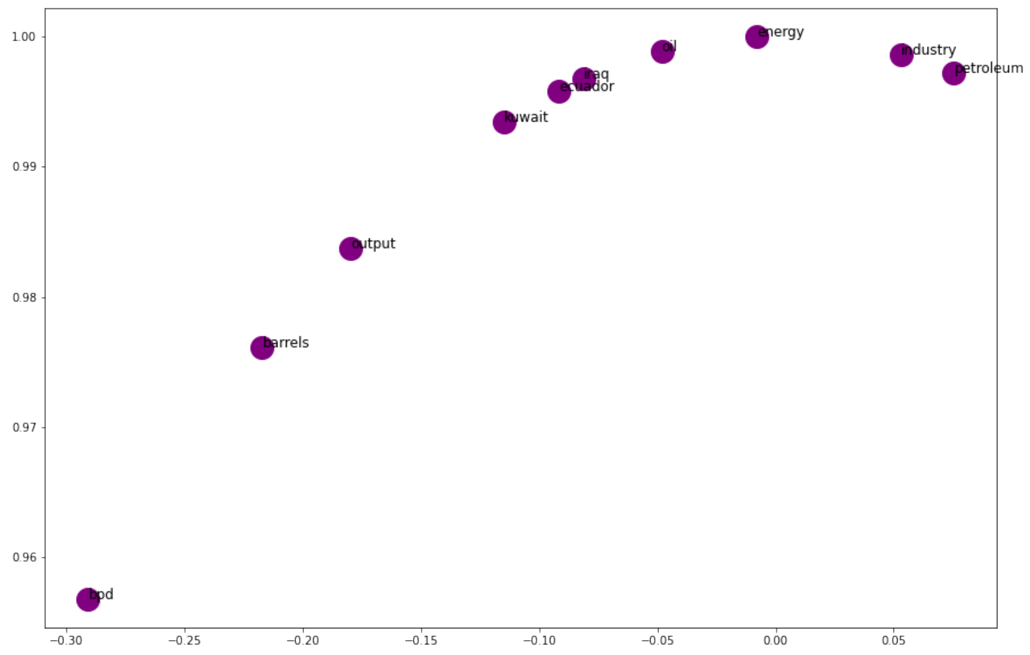
Question 2.5 Co-Occurrence Plot Analysis

Here we had to performs several steps to create a scatterplot of word embeddings.

First, we had to read in a corpus of documents and compute a co-occurrence matrix and a dictionary mapping words to their indices in the matrix.

Next, we reduce the dimensionality of the co-occurrence matrix to 2 and normalize the rows of the resulting matrix to make them each of unit length.

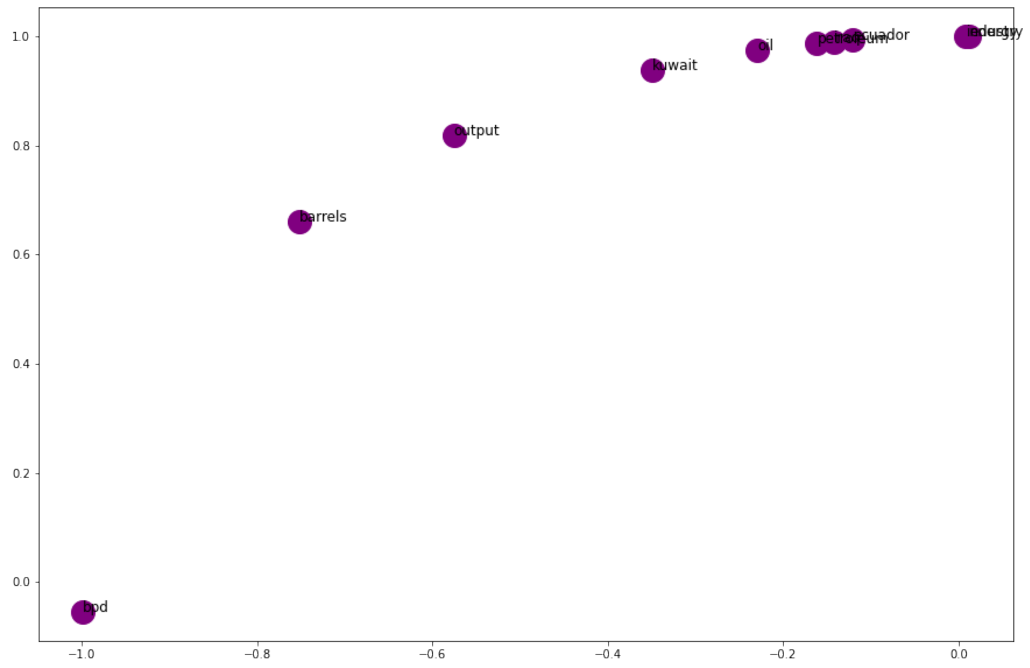
Finally, the code defines a list of words "words" and uses the "plot_embeddings" function to create a scatterplot of the embeddings of these words. The scatterplot is created using the normalized reduced matrix and the word-to-index dictionary.



Part 3. Prediction-based word vectors

Question 3.1 Co-Occurrence Plot Analysis

In a 2D embedding space, words that cluster together are words that have similar contexts, i.e. words that have a common meaning or are related.



The words "barrels" and "petroleum" could have been grouped together as they are related but are not. This may be due to the fact that the two words are not frequent enough in the corpus.

The plot is different from the one generated earlier from the co-occurrence matrix because the words are not grouped together in the same way, now there are word clusters. This is because the co-occurrence matrix is a symmetric matrix, whereas the word2vec model is not and the representation of words are not the same.

Question 3.2 Words with Multiple Meanings

TODO

Question 3.3 Synonyms & Antonyms

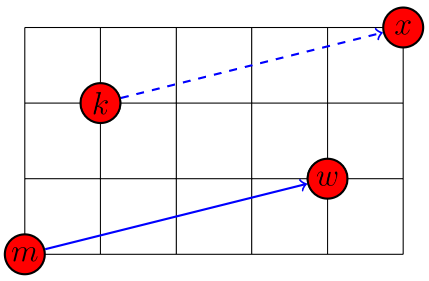
One possible explanation is that the word vectors of synonyms and antonyms are not perfectly orthogonal to each other in high-dimensional space. In other words, the vectors of synonyms and antonyms may not be perfectly correlated and may overlap.

```python
def condition_between_synonyms_antonyms(word1, word2, word3):
   syn_dist =  wv_from_bin.distance(word1, word2)
   ant_dist =  wv_from_bin.distance(word1, word3)

   return ant_dist < syn_dist

condition_between_synonyms_antonyms("hot", "warm", "cold")
```

For example, in the case of "hot" and "cold", the word vectors of these two words may not be perfectly orthogonal, even if they are antonyms. This may be because the two words are often used to describe temperature, and the vectors of these words may have common elements that capture this meaning.

Question 3.4 Analogies with Word Vectors

The expression in which we maximise the similarity of the cosine with x is $w - m + k$.



The cosine similarity between two vectors is maximized when the two vectors are equal. Then the solution is to translate the vector $\overrightarrow{mw} = w - m$ to k.

Question 3.5 Finding Analogies

One example of an analogy that holds according to these vectors could be: "Apple: Fruit :: Potatoes: Vegetables"

```python
def find_analogies(x, y, a):
     return pprint.pprint(wv_from_bin.most_similar(positive=[a, y], negative
    =[x]))

find_analogies("apple", "fruit", "potatoes")
```

```
[('vegetables', 0.7133320569992065),
 ('fruits', 0.6467835903167725),
 ('beans', 0.6283371448516846),
 ('cooked', 0.6031776666641235),
 ('vegetable', 0.5949727892875671),
 ('berries', 0.5880725383758545),
 ('meat', 0.5720727443695068),
 ('mashed', 0.5708385705947876),
 ('bananas', 0.5619319081306458),
 ('lentils', 0.5608645081520081)]
```

Question 3.6 Incorrect Analogy

One example of analogy that does not hold according to these vectors couls be : "Plant : Water :: Human : Food"

```
find_analogies("plant", "water", "human")
```

```
[('rights', 0.5145905613899231),
 ('beings', 0.473871111869812),
 ('blood', 0.47122642397880554),
 ('environment', 0.459567129611969),
 ('humanity', 0.45369186997413635),
 ('earth', 0.4464641511440277),
 ('humanitarian', 0.4438990354537964),
 ('humans', 0.4417664408683777),
 ('basic', 0.44106465578079224),
 ('life', 0.44028201699256897)]
```

Question 3.7 Guided Analysis of Bias in Word Vectors

The terms are most similar to "woman" and "worker" and most dissimilar to "man" are :

- employee
- workers
- nurse
- pregnant
- mother
- employer
- teacher
- child
- homemaker
- nurses

The terms are most similar to "man" and "worker" and most dissimilar to "woman" are :

- workers

- employee
- working
- laborer
- unemployed
- job
- work
- mechanic
- worked
- factory

The list of words associated with women is very much related to parenting and the home. While the men's list is related to factory work. This highlights the gender bias because today men and women do the same jobs.

Question 3.8 Independent Analysis of Bias in Word Vectors

TODO

Question 3.9 Independent Analysis of Bias in Word Vectors

One explanation of how bias gets into the word vectors is that the word vectors are trained on a corpus of text that is biased. For example, if the corpus is a collection of news articles, the word vectors will be biased towards words that are used in news articles. This is because the word vectors are trained on the co-occurrence matrix of the corpus, which is a matrix that counts the number of times each word occurs in the context of each other word. If the corpus is biased, the word vectors will also be biased.

Part 4. Prediction-based sentence vectors

Question 4.1 How would you represent a sentence with Glove? What are the limits of your proposed implementation?

We concatenate the vectors (obtained with GloVe) of the words in the sentence into a single long vector.

A limitation of concatenating all GloVe vectors to represent a sentence is that the resulting vector will be very long, potentially hundreds or thousands of dimensions depending on the number of words in the sentence. This can make it difficult to work with the vector, as it can require a lot of computing resources to process and store. .

Finally, vector concatenation does not take into account the order and relationships between words in the sentence. This can be a problem for tasks that require a deeper understanding of meaning and sentence structure, such as translation or text classification.

Question 4.2 Evaluate clustering quality of SentenceBERT. What makes it good at clustering sentences? Which method of the two below would you go for?

In both cases the SentenceBERT classification is excellent: the sentences are always well grouped.

SentenceBERT is able to handle long-range dependencies between words, allowing it to capture the context and relationships between different parts of a sentence. This allows it to understand the meaning of a sentence and group it with similar sentences.

We would choose the standard method as it seems to be better.

Question 4.3 SentenceBERT Plot Analysis

TODO

Question 4.4 Independent Analysis of Bias in Word Vectors

TODO