Report Data Science Lab – Laboratory 5

The purpose of laboratory 5 is to analyze a dataset of 4000 documents, finding similarities and clustering them according to topics.

The second part consists in the visualization of the result via **wordclouds**.

The first step is reading all the documents.
All the documents are named with a progressive number, indicating their id.

```
def load_data(dirpath):
    files_names = os.listdir(dirpath)
    data = [''] * (len(files_names))

    for filename in files_names:
        sys.stdout.write("\r" + dirpath + '/' + filename)
        f = open(dirpath + '/' + filename, "r")
        data[int(filename)] = f.read()
        f.close()
    print("\n")
    return data
```

Then we have to extract the features from the documents.
This is done by applying a **tf-idf** algorithm to the set of documents.
The **tf-idf** algorithm is implemented by the *sklearn* library.
As argument I specified:

- The stopwords list, indicating the common terms to be ignored
- The tokenizer, useful to avoid different inflexions of the same term to be treated singularly.
- The max_tf and min_tf used to ignore words that are too common or too rare.
  Usually very common terms doesn't bring much information, while very rare terms are likely to be typos or non characterizing words

All the documents are converted in lowercase, without accents.

```
def feature_extraction(data, min_freq, max_freq):
    stop_words = sw.words('english')
    # extending stopwords
    stop_words.extend((
        'abov', 'ani', 'becaus', 'befor', 'could', 'doe', 'dure', 'ha', 'might', 'must', 'need',
          'onc', 'onli', 'ourselv', 'sha', 'themselv', 'veri', 'wa', 'whi', 'wo', 'would','yourselv'))

    # extraction of features
    tokenizer = LemmaTokenizer.LemmaTokenizer()
    vect = TfidfVectorizer(tokenizer=tokenizer, encoding='utf-8', strip_accents='unicode',
                           lowercase=True, stop_words=stop_words, max_df=max_freq, min_df=min_freq,
                               sublinear_tf=True)

    numerical = vect.fit_transform(data)
    return numerical
```

The list of stopwords is taken from *nltk* library and is extended with the words obtained after tokenizing the already present stopwords.

The matrix obtained consists in 4000 rows and a column for each distinct word present in the dataset that has not been ignored by previous filters.

The next step is reducing the dimensionality of the matrix.
This is done by pipelining two different methods:

- The first one is **PCA** (principal components analysis), used to lower dimensionality down to **50** dimensions.
- The second one is **TSNE** (t-distributed stochastic neighbour embedding) used to lower dimensionality down to **3** dimensions.

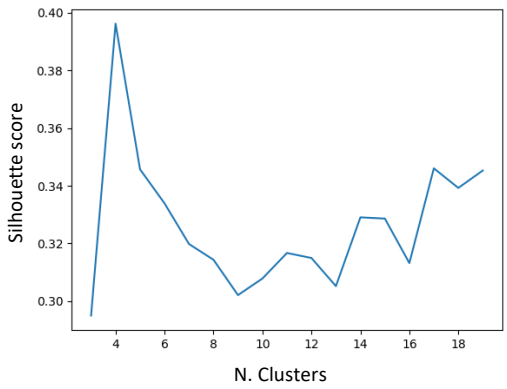
```
def reduce_dim(data, components):
    # reducing dimensions
    svd = TruncatedSVD(n_components=50, random_state=42)
    tsne = TSNE(n_components=components, perplexity=50, verbose=1, init='random', learning_rate=2000,
                n_iter=1000, early_exaggeration=12, method='exact')

    pipeline = make_pipeline(svd, tsne)
    red_X = pipeline.fit_transform(data)

    return np.array(red_X)
```

Now we have a 4000x3 matrix ready to be clustered.
The used algorithm is **MiniBatchKMeans**, a variant of the mainstream **KMeans**.

The number of topics is not known so the optimal number of clusters need to be chosen after different runs of **KMeans**.

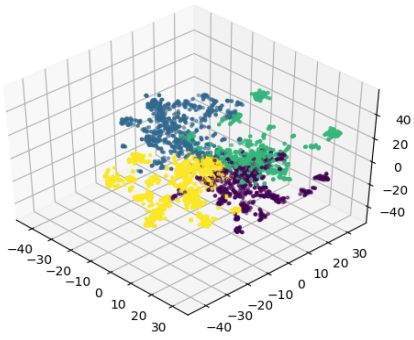
```
def clusterize(points, min, max):
    # clustering MiniBatchKMeans

    scores = []
    clusters = []
    for i in range(min, max):
        print("Trying with " + str(i) + " clusters")
        minib = MiniBatchKMeans(n_clusters=i, init_size=1024, batch_size=1024, random_state=20)
        clusters.append(minib.fit(points))
        scores.append(silhouette_score(points, clusters[i - min].labels_))

    plt.plot(range(min, max), scores)  # plotting silhouette values for all numbers of clusters
    plt.show()
    return clusters[np.argmax(scores)]  # I pick the best
```

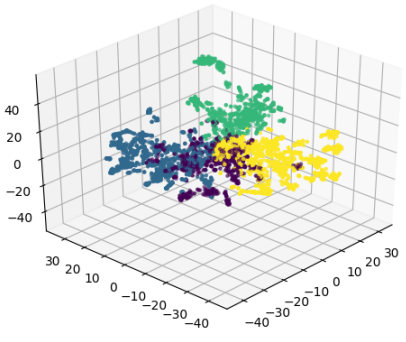
The metric used to evaluate the effectiveness of different number of clusters is the silhouette score.
The results can be plotted:



From the graph we see that the optimal cluster number is **4**, so we dump to file the clustering result with **4 clusters**.

The result can be plotted with a 3d scatter plot, every document is a point, colored according to the cluster.





Now we know how the documents should be grouped, but we still don't know the actual topic of them.

To do so, we can visualize the most common terms among documents of the same cluster and try to guess a possible topic.

The wordcloud library allows us to see them in a fancy way.
Firstly we have to concatenate the documents of each cluster, then we tokenize them in a similar way to the first part, also removing stopwords.
Then we count the frequency of each term with **FreqDist** function from *nltk*.

```
def text_to_freq(raw_text, stopWords):
    raw_text = raw_text.lower()  # lowercase
    raw_text = raw_text.replace("'", "")
    tokens = nltk.word_tokenize(raw_text)  # tokenize
    text = nltk.Text(tokens)

    text_content = [word for word in text if word not in stopWords]  # removing stopwords
    text_content = [s for s in text_content if len(s) != 0]
    WNL = nltk.WordNetLemmatizer()  # apply lemmatizing
    text_content = [WNL.lemmatize(t) for t in text_content]
    fdist = nltk.FreqDist(text_content)  # evaluate frequencies

    word_dict = {}

    for key in fdist:
        word_dict[key] = fdist[key]

    return word_dict
```

Before running wordcloud we delete from the dictionaries the most common words, this ensures that the visualized words will be significative for the topic.

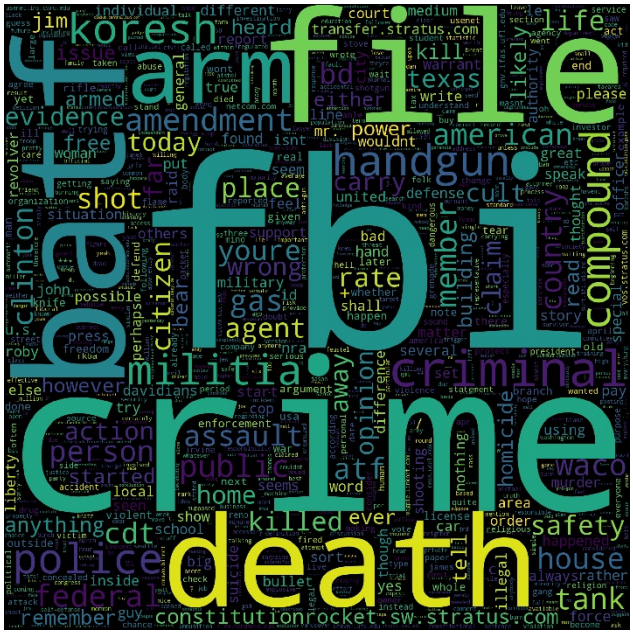
To do so, we concatenate all the documents and compute the total frequency term.
Then we remove from the cluster dictionaries the terms that have a high total frequency among all documents.

```
total_freq = text_to_freq("".join(clusters), stop_words)  # evaluating total term frequencies
max_tf = 500

# I remove all terms that appears too often through all the documents (in more than 500 documents for
instance)
```

```
i = 0
for key in total_freq.keys():
    if int(total_freq[key]) > max_tf:
        i += 1
        for cluster_freq in freq:
            if key in cluster_freq.keys():  # if a common term is present in the cluster term
                                              frequencies, remove it
                cluster_freq.pop(key)
print("Pruned " + str(i) + ' words')
```

The results are the following:

First cluster:



From the most frequent words we can say that the main topic of the document is **police/criminality**.

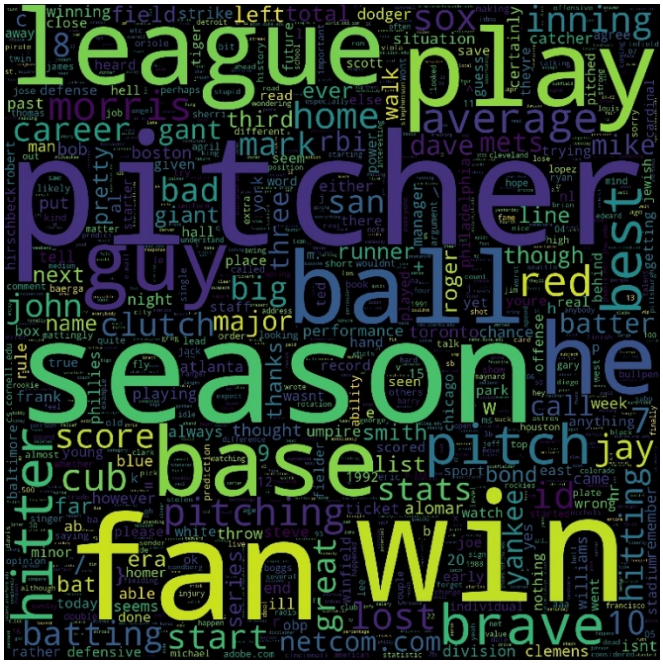
There are some mistakes, for example at the bottom we can see the domain rocket.sw.stratus.com that can be easily related to the 4th cluster.

Notes:
BATF is the Bureau of Alcohol, Tobacco, Firearms and Explosives.

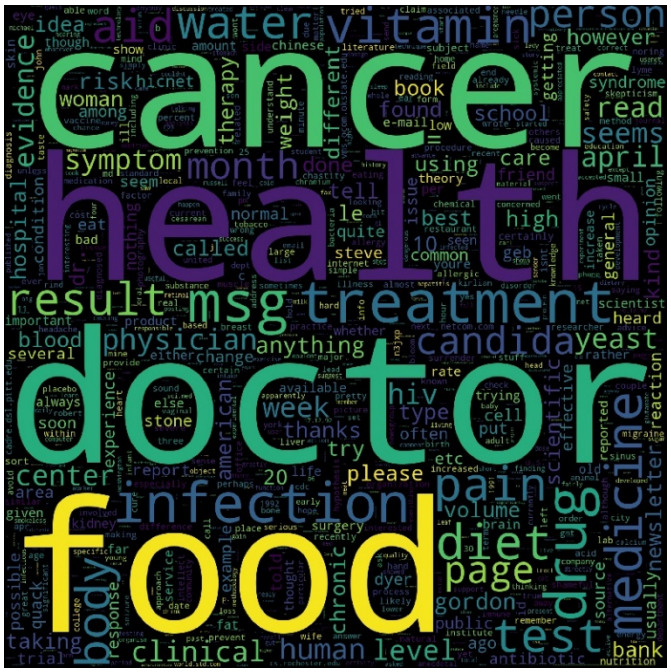
Second cluster:

From the most frequent words we can say that the main topic is **sport**, in particular **baseball**.

(In fact, the pitcher is a role in the sport of baseball)



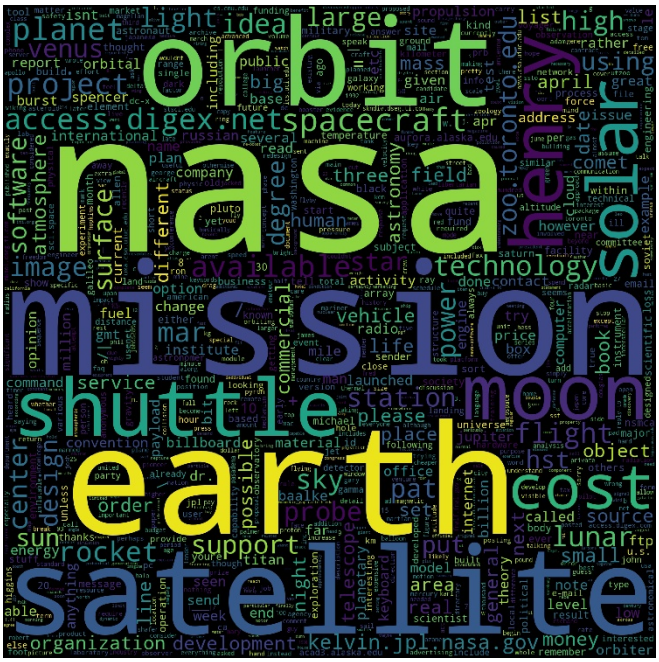
Third cluster:



From the most frequent words we can say that the main topic is **medical/health**

Fourth cluster:

From the most frequent words we can say that the main topic is **space missions**



A further analysis can be done by applying a frequent itemset algorithm to the set of words of each document in the cluster.
The used algorithm is **apriori** from *mlxtend* library, used with different values for **min_support** for every cluster.

```
for n, cluster in enumerate(clusters):
    set = text_to_set(cluster, stop_words, commons)
    te = TransactionEncoder()
    te_ary = te.fit(set).transform(set)

    set = pd.DataFrame(te_ary, columns=te.columns_)
    result = apriori(set, min_support=0.07, use_colnames=True)
    result = sorted(result.values, key=lambda x: -len(x[1]))
    print("Most significant sequences (Cluster"+str(n)+")")
    for res in result[:10]:
        print(res[1])
```

Cluster 1

With a min_support of 0.075 the longest subsets are not very significant for the topic, for instance: *{'company', 'today', 'cdt', 'special', 'speak', 'packet', 'investor', 'write'}*
With an higher min_support the longest are subsets of only 2 elements, but still more significant, for instance: *{'fbi', 'batf'}*

Cluster 2

With a min_support of 0.07 the subsets are short (2 elements) but characterizing, for instance:
{'season', 'game'}
With a lower min_support (0.04) the longest subsets became irrelevant to the topic:
{'mark', 'ms', 'singer'}

Cluster 3

With a min_support of 0.075 the longest subsets are not very significant for the topic, for instance:
{'chastity', 'geb', 'bank', 'intellect', 'gordon'}

Cluster 4

With a min_support of 0.07 some of the subsets are significant: *{'earth', 'orbit'}* while others are not: *{'date', 'gmt'}*
With a min_support of 0.06 some subsets are longer but less specific:
{'henry', 'zoology', 'toronto', 'spencer'}

Appendix

Here I report significant parts of code, not written above

text_to_freq, used to obtain term frequency from a text

```
def text_to_freq(raw_text, stopWords):
    raw_text = raw_text.lower()  # lowercase
    raw_text = raw_text.replace("'", "")
    tokens = nltk.word_tokenize(raw_text)  # tokenize
    text = nltk.Text(tokens)

    text_content = [word for word in text if word not in stopWords]  # removing
                                                                                stopwords
    text_content = [s for s in text_content if len(s) != 0]
    WNL = nltk.WordNetLemmatizer()  # apply lemmatizing
    text_content = [WNL.lemmatize(t) for t in text_content]
    fdist = nltk.FreqDist(text_content)  # evaluate frequencies

    word_dict = {}

    for key in fdist:
        word_dict[key] = fdist[key]

    return word_dict
```

text_to_set, used to get a set of the non distinct words of a document

```
def text_to_set(docs, stopwords, commons):
    sets = []
    for doc in docs:
        regex = re.compile("^[a-zA-Z0-9]+$")

        doc = doc.lower()  # lowercase
        doc = doc.replace("'", "")
        tokens = nltk.word_tokenize(doc)  # tokenize
        text = nltk.Text(tokens)

        text_content = [word for word in text if
                        (word not in stopwords and word not in commons and regex.match(word))]
                                                                    # removing stopwords
        text_content = [s for s in text_content if len(s) > 2]
        WNL = nltk.WordNetLemmatizer()  # apply lemmatizing
        text_content = [WNL.lemmatize(t) for t in text_content]
        sets.append(text_content)
    return sets
```

load_cluster, used to load in a row all the documents belonging to a cluster

```
def load_clusters(data, csv_path):
    ids = {
        "Id": [],
        "Predicted": []
    }
    with open(csv_path, mode='r') as file:
        csv_reader = csv.reader(file, delimiter=',')
        header = True
        for row in csv_reader:
            if header:
                header = False
                continue

            ids["Id"].append(row[0])
            ids["Predicted"].append(row[1])

    n_clusters = int(max(ids["Predicted"])) + 1  # eval total number of clusters
    clusters = [[] for i in range(n_clusters)]
    for id, cluster in zip(ids["Id"], ids["Predicted"]):
        clusters[int(cluster)].append(data[int(id)])  # appending all text belonging to the same
cluster

    return clusters
```