**2.** The second task consists in finding a top 3 topics on Stack Overflow for a certain post from a given dataset.

This task could be solved using mainly **TfidfVectorizer** in order to find scores of apparition of every word in the dataset. First of all, there were selected the most relevant columns from the .csv file (the title of the post and the body of the post). These two columns were concatenated in Excel and they were read in the code from the .csv file.

df = pd.read\_csv('results-20190921-122938 modified.csv', delimiter=',')  
text = df['titlebody'].values.astype('U')

In order to clean the text (tokenization, lemmatizing and removing words with less than 3 letters), it was used the same function of word processing from the previous task:

def word\_processing(sentence):  
 results = ''  
  
 for token in sentence:  
 if len(token) > 3:  
 results = results + " " + str(token.lemma\_)  
 return results

The tokenization was performed using Spacy. The next steps consist on a data manipulation process. Most of all, in order to call the word\_processing function on the given data, it has to be the **nlp** type and in order to apply **nlp**, they have to be strings:

for i in text:  
 i = str(i)  
 str\_text.append(i)  
  
nlp = spacy.load('en\_core\_web\_sm')  
  
for i in str\_text:  
 doc = nlp(i)  
 doc\_list.append(doc)

Next, the word\_processing function is called, and the results are appended to a **words** list.

for i in doc\_list:  
 words.append(word\_processing(i))

Now, the **TfidfVectorizer** can be called and the **fit** and **transform** methods can be called on the processed data. Then, the found features could be returned using the **get\_feature\_name** method.

tv = TfidfVectorizer(stop\_words='english')  
tv.fit(words)  
filtered\_words = tv.transform(words)  
features = tv.get\_feature\_names()

Next, a **dataframe** of the **tfidf** score and the feature names (which will represent the topics) could be created, this way:

df\_tf\_idf = pd.DataFrame(filtered\_words.toarray()[n], index=features, columns=['TfIdf Score'])  
df\_tf\_idf.sort\_values(by=['TfIdf Score'])  
  
values = df\_tf\_idf['TfIdf Score'].nlargest(3)  
  
print(values)

The topics are sorted with respect to the score and then, the first 3 are chosen using the **nlargest** method. The **n** variable, which is an index for the **filtered\_words.toarray()[n]** is actually the number of the post from the dataset.

So, these are the main 3 topics of the 15th post of the dataset (n = 15), with their corresponding **tfidf** score:

**assembly 0.485438**

**sign 0.368265**

**built 0.330308**

For **n = 30**:

**memory 0.411179**

**heap 0.260550**

**observe 0.207165**

As a further improvement, the **hypernyms** of these features could be found and used in order to better generalize the names of the possible topics.

**3.** As a bonus task, I decided to analyze the top 3 topics regarding all the posts on the dataset. It could be seen also as a clustering application, but I tried to solve it also with NLP techniques. Besides the **CountVectorizer** class, the main entities which help the application find the topics are two classes from the **sklearn.decomposition** library: NMF (Non-Negative Matrix Factorization) and LatentDirichletAllocation.

The same word\_processing function was used for cleaning the data. This time, the returned value is a list.

def word\_processing(text):  
 results = []  
  
# results.append([token.lemma\_ for token in text if len(token) > 3])  
  
 for token in text:  
 if len(token) > 3:  
 results.append(token.lemma\_)  
  
 return results

The **CountVectorizer** model was used to find the frequency of apparition of every token:

df = pd.read\_csv('results-20190921-122938.csv', delimiter=',')  
text = str(df['title'][0:10].values.astype('U'))  
  
nlp = spacy.load('en\_core\_web\_sm')  
doc = nlp(text)  
  
words = word\_processing(doc)  
  
#print(words)  
  
cv = CountVectorizer(analyzer='word',lowercase=True, stop\_words='english')  
filtered\_words = cv.fit\_transform(words)  
features = str(cv.get\_feature\_names())

Then, the two mentioned models are trained through the **fit**  method.

lda = LatentDirichletAllocation(n\_components=n, max\_iter=20, learning\_method='online')  
lda\_data = lda.fit(filtered\_words)  
  
nmf\_model = NMF(n\_components=n)  
nmf\_Z = nmf\_model.fit(filtered\_words)

Here, the **n\_component** refers to the number of topics that needs to be found. But this number is not related to the number of topics we want to be found. It is more like the number of "clusters" which need to be selected. It needs to be optimal for the application and there are methods (like the Elbow method in case of K-Means) to find this optimal number of topics. A correct implementation would have taken into consideration this fact and after the first n topics were selected, the first 3 to be printed. In this case, I chose **n = 3**, but this value might not be the optimal one.

In order to print the topics, the function **print\_topics** was implemented:

def print\_topics(model, vectorizer, top\_n=10):  
 for idx, topic in enumerate(model.components\_):  
 print("Topic %d:" % idx)  
 print([(vectorizer.get\_feature\_names()[i], topic[i]) for i in topic.argsort()[:-top\_n - 1:-1]])

The next results were obtained:

**NMF model:**

**Topic 0:**

**[('multiple', 1.492940761649883), ('resource', 9.768671096132694e-07), ('standard', 5.51086078837544e-07), ('contain', 5.18335390537324e-07), ('dataitem', 4.983435865045754e-07), ('window', 4.866374698555849e-07), ('regular', 4.58627333068094e-07), ('upload', 4.062022652405716e-07), ('server', 3.8091917584697177e-07), ('project', 3.7580079346423423e-07)]**

**Topic 1:**

**[('file', 1.0934767471574738), ('net', 0.6282289620670124), ('asp', 0.6282289620670124), ('matching', 3.164234023651136e-06), ('custom', 3.0047795940458663e-06), ('expression', 2.5207908179255615e-06), ('create', 2.283079893280526e-06), ('javascript', 2.179480203246969e-06), ('iphone', 1.4614953649016342e-06), ('extra', 8.284532282663327e-07)]**

**Topic 2:**

**[('quote', 1.3628894822545294), ('asp', 1.7481464160782764e-05), ('net', 1.7481464160782764e-05), ('good', 2.490373622749675e-06), ('fuzzy', 2.3236604598644913e-06), ('python', 2.301423266324544e-06), ('membershipprovider', 2.165055116843714e-06), ('view', 1.7691261771737926e-06), ('reference', 1.7480521931962772e-06), ('property', 1.723729045046758e-06)]**

**LDA model:**

**Topic 0:**

**[('file', 2.2332953647202456), ('py2exe', 1.3099288818925052), ('create', 1.3081338556699489), ('matching', 1.3018177995986688), ('upload', 1.2971459937197245), ('listview', 1.2967266497800063), ('window', 1.296673475590235), ('escaped', 1.295231525679783), ('property', 1.2951398007512822), ('expression', 1.293607344291446)]**

**Topic 1:**

**[('multiple', 2.2150765945694233), ('net', 1.3168804186604377), ('python', 1.3099759404233149), ('asp', 1.3072941828240967), ('custom', 1.3022236108631469), ('validation', 1.2983420210836658), ('fuzzy', 1.2967326883191908), ('wrap', 1.295073829079107), ('pyinstaller', 1.2927713441193438), ('executabl', 1.292364948084633)]**

**Topic 2:**

**[('quote', 2.2295116605194916), ('search', 1.3028095312698045), ('extra', 1.3027335802686366), ('membershipprovider', 1.2987458206701203), ('good', 1.2985652546058635), ('iphone', 1.2979886876215765), ('solution', 1.2978050674283972), ('resource', 1.296718235576194), ('dataitem', 1.2957802367509352), ('standard', 1.2952191570696967)]**

It can be observed that the models do not return the topics themselves. They find some tokens relevant to the topics and the interpretation part belongs to us. Also, the function **print\_topics** is given as a parameter a top of words. This top refers to the number of relevant tokens that are found for every possible topic. In this case, this number was chosen to be 10.