

Extracting sentence, which is similar to most of the sentences using python.

Shaury Srivastav

25th October, 2021



Introduction

1. Identifying the problem.
2. Approach used.

Identifying the problem


While working with large texts, it becomes tedious for us to go through all the text, hence sometimes useful information which must be taken into account is often left out .

For humans to understand what's important and should be taken into account is easier, but for computers the same is a tough job . But for the same problem, if we are talking about a large amount of information, then human capabilities would be put to test. In order to solve this problem we can use NLP or natural language processing techniques to make our problem easier to interpret.

For example if we have sentences such as :

- 1) His name is SAM.
- 2) SAM is his name .
- 3) We call him by the name SAM .

All the above sentences have the same meaning, and any one of the sentences can be used to define the behavior of others as well. Here the name SAM is common in all and the pronouns used are his or him. For us to understand the similarities is easy but lets suppose we are given 1000 sentences and we have to find the alike sentences amongst them , then it would require a lot of time.



For the purpose of understanding, we will use the case of job description regarding the various job titles.

Lets suppose we have various job titles and we have the respective job descriptions for the same titles from different job portals such as naukri.com , timesjob.com , monster.com and indeed.com. Now we dont need the whole description but just the responsibilities for those job postings and out of those responsibilities we need the top responsibility for each of the job title.

Approach used

For this kind of problems we use the method of Chunking for filtering the sentences based upon certain grammar rules. So that we get only those sentences that are actually regarding the responsibility.

Chunking in python can be done by using the NLTK library, below is a code snippet for the same :

```
from nltk.tokenize import word_tokenize
from nltk.chunk import RegexpParser
from nltk import Tree
```

```

def chunkI(i):
    if(i=='None'):
        return i
    rules = r"""Chunk:
{<DT>?<JJ>*<V.+>+<.*>*|<JJ.??>*<IN.??>*<VBG.??>|<NN.??>*<RB.??>*<IN.??>*|<NNS.??>*<CC.??>*<NNS.??>*<N
N.??>*|<DT.??>*<NN.??>*<VBP.??>*|<VB.??>*<CC.??>*<VB.??>*<JJ.*>|<VB.??>*<JJ.??>*<NNS.??>*<VB.??>*<.*>|<
RB.??>*<VB.??>*<NN.??>*<NN.??>*<.*>}
"""
    chunk = RegexpParser(rules)
    x = ""
    for j in i.split("."):
        y = j+"."
        words = word_tokenize(y)
        flag = 1
        for ind in words:
            if(ind in respon):
                x = x + y;
                flag = 0
                break
        if(flag):
            postags = nltk.pos_tag(words)
            for child in chunk.parse(postags):
                if isinstance(child,Tree):
                    if (child.label() == 'Chunk'):
                        x = x+y+" "
                        break
    return x

```


The above code snippet shows how we can filter out sentences using chunking.

The RegexpParser is feeded with certain grammar rules :

Here <DT>?<JJ>*<V.+>+<.*>* is a grammar rule.

DT is determiner , JJ is adverb V is verb.

For better understanding of part of speech tagging refer to this [LINK](#).



The major problem here is some, since we are just using the grammar discrimination rule hence some non-useful sentences might also be included in our corpora, to avoid that we use the **re** library which is the regular expression library. We only take those sentences into account which don't have the skills, education or experience required for that job posting.


Hence, we used the code below to do so :

```
def noskill(x,job):
    for i in skilldict[job]:
        try:
            if(re.search(i,x)):
                return False
        except:
            continue
    return True
```

In the above snippet, the skilldict is a dictionary of skills required for that particular job title. We iterate over all the skills and see if that sentence has that skill word or not, if it's present we don't take that sentence into our corpora.

While making the skill dictionary we must take into account only those skills which are having the NNP(proper noun) tag, because if we see the skill set required for many of the job postings then most of the skills required are generalized verbs or adjectives which are present in the responsibility sentences as well, such as

Sales , application , communication ,etc.



For the education and experience columns we do the same as for skills.

After these steps we get the mostly the useful sentences which we can use for finding the top responsibility.

For finding the top responsibility for a job title we have our preprocessed text, now we just need to find the similar sentences and club them different clusters and select the sentence which is able to explain most of the sentences in that cluster or in easy language , we need to take the sentence which is similar to most of the sentences.

For finding the similarity in the sentences we use the Word Mover's approach.

Lets go back to the example above :

- 4) His name is SAM.
- 5) SAM is his name .
- 6) We call him by the name SAM .

In these sentences if we were to take into account the unique words then we would get a list like

```
['his', 'name', 'is' , 'sam', 'we', 'call', 'by',  
'the' , 'him']
```

While finding the similarity, we use the Word2Vec approach .

For this purpose we convert these sentences into numerical vectors.

The below code snippet is how we convert sentences into vectors.

```
from nltk.tokenize import PunktSentenceTokenizer
punkt = PunktSentenceTokenizer()
def sentence_maker(sentence):
    """
    Input : A string of whole description text.
    Return values :
        sentences : list of sentences in the for of tokenized words.
        full_Sent : list of sentences in the for of tokenized sentences from
the main text.
    """
    sentences = []
    full_sent = []
    for i in punkt.tokenize(sentence):
        if(len(i.split())>3):
            sentences.append(i.lower().split())
            full_sent.append(i)
    return sentences,full_sent
```

In the above code we use the pretrained sentence tokenizer PunktSentenceTokenizer for the purpose of tokenizing the sentences for our final corpus.

After this step we use the Word2Vec approach

```
from gensim.models import Word2Vec
W = Word2Vec(sentences,min_count = 1)
```

The word2vec instance takes the sentences array as input which we got from the above sentence_maker function.

Further we define a function to vectorize the arrays and use them for our final model.

The function below is for vectorization:

```

def vectorizer(sent,W):
    vec = []
    numw = 0
    for i in sent:
        try:
            if(numw==0):
                vec = W[i]
            else:
                vec = np.add(vec,W[i])
            numw+=1
        except:
            pass
    return np.asarray(vec)/numw
l = []
for sent in sentences:
    l.append(vectorizer(sent,W))
X = np.array(l)

```

Finally we move on to making our clustering model using the KMeans approach from sklearn library in python. We take the array X for training our model.

For general purpose we are taking the top 5 responsibilities for each job title, hence we take number of clusters as 5. We train the model and our model creates clusters, out of those clusters we chose the top three responsibilities from each of the cluster and club them together to get our top responsibility.

We use the sentence_transformer library to generate encodings and then based upon cosine similarity we find the alike sentences.

Below is the implementation code .


```

from sklearn.clusters import KMeans
from sklearn.metrics.pairwise import cosine_similarity #word mover distance
approach
from sentence_transformers import SentenceTransformer

kmodel = KMeans(n_clusters=5, init= 'k-means++')
kmodel.fit(X)
sentsdf = pd.DataFrame({"Sent":full_sent,"label":kmodel.predict(X)})
pd.DataFrame({"Sent":full_sent,"label":kmodel.predict(X)})

### similarity
model = SentenceTransformer(model_name_or_path="bert-base-nli-mean-tokens") #using
the bert model
top_resp = []
for lab in range(0,5):
    sentences = []
    for sent in sentsdf[sentsdf['label']==lab]['Sent']:
        sentences.append(sent)
    #defining the model
    sentence_vecs = model.encode(sentences)

    ## the final dictionary for the final data frame
    mydict = {}
    for i in sentences:
        mydict[i] = 0

    ## taking the sentences with significant similarity with other sentences
    cnt = 0
    for i in sentence_vecs:
        for j in cosine_similarity([i],sentence_vecs)[0]:
            if(j>0.65): #using 0.65 to get the most significant counts only
                mydict[sentences[cnt]] +=1
            cnt=cnt+1

    dftemp =
pd.DataFrame([mydict]).transpose().sort_values(0,ascending=False).reset_index()
    dftemp.columns = ['Sent','count']
    top_resp.append(dftemp['Sent'][0]+" ". +dftemp['Sent'][1]+" ".
"+dftemp['Sent'][2])

```

The above code gives us a final array top_resp which contains the top 5 responsibilities for a particular job posting.