Headless Technologies Limited

* Problem Statement Description:

In today's environment, the way information is accessible may be greatly influenced by the use of frequently asked questions (FAQ) systems. Within the field of computer science, the field of FQA may be located at the point where information retrieval and natural language processing meet. The creation of intelligent computer systems that are able to react in natural language to questions posed by users is the focus of this area of study. FQA systems that are able to retrieve desired information accurately and quickly without having to skim through the entire corpus can serve as valuable assets in industry, academia, and our own personal lives. This can result in improved search engine results and more personalized conversational chatbots, for example. Law companies and human resources departments are two examples of real-world settings in which FQA systems have the potential to be useful. In the first scenario, specific information may be required (for citation in a current case) from the records of thousands of previous cases; nonetheless, adopting a FQA system may save a significant amount of time and labor when compared to other options. In the latter scenario, an employee may need to look up particular laws, such as those pertaining to vacations; employing a FQA system might assist them in this endeavor by providing the appropriate answers to queries of the type "How many job openings are available each year?"

* Solution Procedure:

For solving this FAQ problem statement, I have to consider a few preliminary steps before moving forward with the models. First of all, I have to use regular expression techniques for cleaning the datasets. After that, I need to use different approaches so that I can make it acceptable to others.

If I am going to do this NLP assignment, there are a few fundamental measures that I need to perform in order to assist the program in comprehending natural language:

1. Sentence Segmentation
2. Word Tokenization
3. Text Lemmatization
4. Stop Words
5. Dependency Parsing
6. Named Entity Recognition (NER)

1. Sentence Segmentation:

```
nlp = spacy.load("en_core_web_sm")
questionText = df["Answer"][8]
doc = nlp(questionText)
#to print sentences
for sent in doc.sents:
   print(sent)
```

2. Word Tokenization:

```
nltk_tokens = nltk.word_tokenize(questionText)
print (nltk_tokens)
```

3. Lemmatization:

```
tag_map = defaultdict(lambda : wn.NOUN)
tag_map['J'] = wn.ADJ
tag_map['V'] = wn.VERB
tag_map['R'] = wn.ADV

text = df["Answer"][8]
tokens = word_tokenize(text)
lemma_function = WordNetLemmatizer()
for token, tag in pos_tag(tokens):
   lemma = lemma_function.lemmatize(token, tag_map[tag[0]])
   print(token, "=>", lemma)
```

4. Stop Words:

```
data = df['Answer'][8]
stopWords = set(stopwords.words('english'))
words = word_tokenize(data)
wordsFiltered = []

for w in words:
   if w not in stopWords:
      wordsFiltered.append(w)

print(wordsFiltered)
```

Textual Information Processing:

The data needs to be pre-processed in order for us to get it all into a format that is consistent before we can begin any NLP project. Our data has to be cleaned up, tokenized, and matricified before we can use it. Let's develop a function that, when applied to the text columns, will carry out the following operations:

* Make text lowercase,
* Removes hyperlinks,
* Remove punctuation
* Removes numbers
* Tokenizes
* Removes stopwords

I have a frequently asked questions list available in CSV format in the following file: It discusses Albert Einstein and includes a lot of questions and answers about him. In the datasets, it has both of the following features:

1. Question
2. Answer

There were two different datasets, one for training and another for testing or validating the model's performance.

For solving this problem statement I have used bellow packages from python:

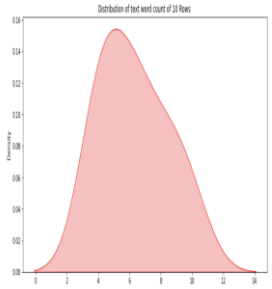
1. Fast Sentence Embeddings --> fse
2. Gensim --> gensim
3. Natural Language Toolkit --> nltk
4. Scikit-learn --> sklearn

5. Regular Expressions --> re
6. Uni Code Data --> unicodedata
7. Matplotlib
8. Seaborn
9. Pytroch --> torch
10. Transformers

Fast Sentence Embeddings" (FSE) :
import fse

There are several different embedding approaches that we may employ to successfully incorporate the meta data of our query. In my experience, I have made use of "Fast Sentence Embeddings," which is a Python package that may be thought of as an extension of Gensim. The purpose of this library is to make it as simple as possible to locate sentence vectors for big sets of sentences or texts by using vectors.

Distribution of text word count of 10 Rows



Data Pre-processing:
Datapoints that contain NAN values, if present in the question or answer columns, are dropped from the dataframe. I then read the question pattern for each datapoint and extracted the question text from that file in the dataframe. Newlines ("n") in the Queston column are replaced with "." fullstops.

Data Cleaning:
The content in the Question and Answer columns are cleaned up by deleting the [^\w\s\.\?] characters. The resulting text has its upper-case letters changed into lower-case ones.

Word Embedding
Embedding models are able to extract the meaning of words by making use of the contexts in which they appear. These models are founded on the concept of co-occurrence. I make use of the capabilities of such models and modify them so that they can be applied to a scenario of question-answering to serve as our baseline models. The following provides a detailed description of my implementation of embedding-based approaches:

Model for answering questions based on Word2Vec data:
Word2Vec is an application that learns word associations from a large text corpus by employing a neural network. This model, once trained, is able to recognize similarities within words. It can also be used to predict synonyms and measure the cosine similarity between different words. The machine is able to comprehend the semantics of the language because it is able to measure the degree to which words are similar to one another. Utilizing this in the construction of a FQA system is possible. On our dataset, I was able to modify word2vec embeddings in order to accommodate a question-and-answer format as follows:

```
# predict the most similar document
X = embedding.transform([question])
```

```
Q_id = retriever.kneighbors(X, return_distance=False)[0][0]
selected = df.iloc[Q_id]['Question']

# vectorize the document
transform_text(embedding, selected)
```

Using fundamental Python data manipulation techniques, the incoming data are first transformed into a list of lists. The word2vec model is updated with these new pieces of data. After that, the model is trained for a total of fifty epochs. The embedding size is maintained at its default value of 100, and the size of the context window is set to 8. After the model has been properly trained, I will start responding to inquiries. I broke down the question that we were asking into its individual words and input them into the Word2Vec system. After adding all of the created embeddings together, the results are then averaged. The question now has a context thanks to this embedding. After that, I go on to the relevant article content, which is intended to be the source from which the response is formed, and I break it up into individual phrases. After that, I utilize a method that is conceptually similar to locate embeddings for each sentence that is included in our article content. Once I obtain the embeddings for the question as well as for each phrase in the response text, I next use the cosine similarity method to determine the degree of similarity between the embeddings of the question and those of each article sentence. The prediction made by the model for the output of the given question is the phrase that has the greatest degree of similarity with the question.

The Word2Vec model works in this manner by selecting a phrase from the article text that has the greatest Word2Vec embedding similarity with the question that is being asked and then outputting that sentence as the response. This extremely basic question-answering system has a good chance of getting the right response right most of the time.

I have taken two parts from transformer:

1. BertTokenizer
2. BertForQuestionAnswering

NLP's Transformer is a novel design that seeks to handle problems sequence-to-sequence while readily addressing long-distance dependencies. Because of this, the use of two transformer modules is required in order to take use of the architecture. For the purpose of determining the input and output representations, we do not make use of sequence-aligned RNNs or convolutions. Instead, we rely only on paying attention to ourselves.

1. BertTokenizer

The BertTokenizer was yet another job that was finished for this problem statement. A tool that is referred to as a "word fragment tokenizer" is used by BERT. The process works by disassembling words into either their whole forms (for example, one word becomes one token) or into word parts, with the possibility that a single word may be broken up into several tokens. One scenario in which this may be helpful is one in which a word can be written in more than one way.

```
tokenizer = BertTokenizer.from_pretrained('bert-large-uncased-whole-word-masking-finetuned-squad')
```

2. BertForQuestionAnswering

I have developed a new model for the representation of querstion and answers called BERT, which stands for Bidirectional Encoder Representations from Transformers. BERT, in contrast to more contemporary models of language representation, is intended to pre-train deep bidirectional representations from unlabeled text. This is accomplished by simultaneously conditioning on both left and right context at all levels of the model. Therefore, the pre-trained BERT model can be fine-tuned with just one more output layer to make state-of-the-art models for a wide range of tasks, such as answering questions and making inferences about language, without having to make significant changes to the architecture for each task. This is possible because the pre-trained BERT model has already been trained.

```
model_bert = BertForQuestionAnswering.from_pretrained('bert-large-uncased-whole-word-masking-finetuned-squad')
```

I used cosine similarity to compare the vectors here.
Cosine Similarity

The degree of resemblance between two vectors in an inner product space may be measured using the cosine similarity. It checks if two vectors are heading nearly in the same direction by measuring the cosine of the angle formed by the two vectors and comparing the results. In text analysis, it is often used as a measurement tool for document similarity.

```
def cosine_similarity(X, Y=None, dense_output=True):

  X, Y = check_pairwise_arrays(X, Y)

  X_normalized = normalize(X, copy=True)
  if X is Y:
    Y_normalized = X_normalized
  else:
    Y_normalized = normalize(Y, copy=True)

  K = safe_sparse_dot(X_normalized, Y_normalized.T, dense_output=dense_output)

  return K
```

```
from wordcloud import WordCloud
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=[30, 15])
wordcloud1 = WordCloud( background_color='white',
                width=600,
                height=400).generate(" ".join(Question_text_clean))
ax1.imshow(wordcloud1)
ax1.axis('off')
ax1.set_title('Question text ',fontsize=40);

wordcloud2 = WordCloud( background_color='white',
                width=600,
                height=400).generate(" ".join(Answer_text_clean))
```

```
ax2.imshow(wordcloud2)
ax2.axis('off')
ax2.set_title('Answer text ',fontsize=40);
```

Wordcloud

A collection or cluster of words that are represented in varying sizes is called a word cloud. When a word is given a larger font size and bolder font style, it indicates that the supplied text emphasizes the significance of that term by referring to it more often than other words.

Jaccard index

The Jaccard similarity is a measurement that determines the degree of similarity between two different sets of data to determine which members of the sets are common and which members are unique. The Jaccard similarity is determined by taking the total number of observations from both sets and dividing that number by the total number of observations from either set.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

J = Jaccard distance

A = set 1

B = set 2

```
def jaccard(str1, str2):
   a = set(str1.lower().split())
   b = set(str2.lower().split())
   c = a.intersection(b)
   return float(len(c)) / (len(a) + len(b) - len(c))
```

```
Sentence_1 = df['Question'][0]
Sentence_2 = df['Answer'][0]
Sentence_3 = df['Answer'][1]
```

```
print(jaccard(Sentence_1,Sentence_2))
print(jaccard(Sentence_1,Sentence_3))
```

Result and Discussion

The Conversation, as Well as Some Closing Ideas Creating an effective question-and-answer system is a tough undertaking that one must undertake. This problem becomes much more difficult to solve when one considers the several types of results (yes/no, descriptive, spans), which a user may anticipate getting in response to a single question. For example, a user may expect to receive yes/no responses. After beginning with straightforward approaches such as Word2Vec and SIF embeddings, we progressed to attention-based state-of-the-art models such as BERT, which are more difficult to understand.

```
def getAnswerBert(question, context):

  # print('Query Context has {} tokens.'.format(len(tokenizer.encode(context))))
```

```
    context_list = get_split(context)
    ans = []

    for c in context_list:

        encoding = tokenizer.encode_plus(text=question,text_pair=c)

        inputs = encoding['input_ids']  #Token embeddings
        token_type_id = encoding['token_type_ids']  #Segment embeddings
        tokens = tokenizer.convert_ids_to_tokens(inputs) #input tokens

        output = model_bert(input_ids=torch.tensor([inputs]), token_type_ids=torch.tensor([token_type_id]))
        start_index = torch.argmax(output.start_logits)
        end_index = torch.argmax(output.end_logits)

        answer = ' '.join(tokens[start_index:end_index+1])

        ans.append(answer)
    print('Question: ', question)
    potentials = []
    for i in ans:
        if ('SEP' not in i) and ('CLS' not in i):
            potentials.append(re.sub('(#)+', '', i))
    answer = getBestAnswer(question, potentials)

   # print('Potential Answers: \n')
   # print(answer.head())
    return answer


print(getAnswerBert(df['Question'].iloc[9], df['Answer'].iloc[9]))
print("---------------------------------")
print('Actual Answer: ', df['Answer'].iloc[9])
```

Output:

```
Question:  for what did he receive the nobel prize

0    his discovery of the law of the photo ele ctric effect
Name: 0, dtype: object
---------------------------------
Actual Answer:  einstein was rewarded for his many contributions to theoretical physics and especially for his discovery of the law of the photoelectric effect
```

The word-to-vec paradigm worked well for us as a starting point in our work. We predicted that it would provide inaccurate results due to the fact that it is a pretty basic approach that is based on phrase similarity and was not intended for question-answering activities. This led us to believe that it was not designed for such jobs. Because it does not comprehend or understand the information, Word2Vec cannot be used to generate responses in natural language. This is because it does not. The most notable disadvantage it has is that it cannot deliver direct replies that are human-like; rather, it can only create phrases drawn from the answer text. This constraint prevents it from providing responses that are more natural. In spite of this, the model performed well, getting a high

score according to our one-of-a-kind criteria in spite of the intricacy of the dataset that we used. We were able to acquire the information we needed from a few of the replies it supplied, depending on how closely they linked to the question, and in general, this strategy was beneficial for us. We were able to get the information we needed from a few of the responses it offered.