Natural Language Processing Assignment - I

Natural Language Processing Assignment - 1 report submitted to Department of Computer Science and Engineering

Indian Institute of Technology Kharagpur

by

Shatansh Patnaik

(20MA20067)

Under the supervision of

Dr. Saptarshi Ghosh

Department of Computer Science and Engineering

Indian Institute of Technology Kharagpur

Spring Semester, 2024-25

February 4, 2024

Contents

Chapter 1

Implementation of Tasks

1.1 Task 1

1.1.1 Importing the required libraries and deleting the unnecessary columns

In the first step of any Machine Learning task, we import the essential libraries like *Pandas*Pandas, *spaCy* and *re*. Next as we import the datasets in the form of Data Frames as *queries* (data from *queries.csv*), *df* (data from *docs.csv*) and *qdrel* (data from *qdrel.csv*). We observe that an unnecessary column going by the name *"Unnamed"*. Therefore we delete it before proceeding further. The code for the same can be illustrated in Python as follows:

```
# Importing the essential libraries
import spacy
import re
import pandas as pd

# Importing the CSV files in the form of Dataframes
df = pd.read_csv("./Query_Doc/docs.csv")
queries = pd.read_csv("./Query_Doc/queries.csv")
qdrel = pd.read_csv("./Query_Doc/qdrel.csv")

print("\033[1m" + "Docs Data Frame: \n" + '\033[0m\n', df.head())
print("\033[1m" + "Queries Data Frame: \n" + '\033[0m\n', queries.head())
print("\033[1m" + "Relational Data Frame: \n" + '\033[0m\n', qdrel.head())

# Removal of the unnamed column from the dataframe
df = df.drop(df.columns[df.columns.str.contains('Unnamed', case=False)], axis=1)
```

```
queries = queries.drop(queries.columns[queries.columns.str.contains('Unnamed', case=False)], axis=1)
qdrel = qdrel.drop(qdrel.columns[qdrel.columns.str.contains('Unnamed', case=False)], axis=1)

print("\033[1m" + "Docs Data Frame: \n" + '\033[0m\n', df.head())
print("\033[1m" + "Queries Data Frame: \n" + '\033[0m\n', queries.head())
print("\033[1m" + "Relational Data Frame: \n" + '\033[0m\n', qdrel.head())
```

1.1.2 Text Pre-processing: Removal of characters other than AlphaNumerics and White Spaces

In this sub-task we shall use the Regex library (*re* package) to channel out all characters except for lowercase, uppercase alphabets, numbers and white spaces. For this task, we have substituted the alphanumerics and spaces with a single space. We have done this instead of substitution with an empty string because, we have strings in the database of the form *"word...word"*, and if we remove the dots then both the words get appended together, as a result of which the context of the entire sentence is lost. The code for the following can be illustrated as follows:

```
# Removal of characters other than alphabets (both uppercase and lowercase included), digits and numbers
def purify_docs(data):
    purified_doc = re.sub(r'[^A-Za-z0-9]+|\s+', ' ', data)
    return purified_doc

df['pure'] = (df['doc_text']).apply(purify_docs)
queries['pure'] = (queries['query_text']).apply(purify_docs)

print("\033[1m" + "Docs Data Frame: \n" + '\033[0m\n', df.head(), "\n")
print("\033[1m" + "Queries Data Frame: \n" + '\033[0m\n', queries.head(), "\n")
```

1.1.3 Correction of Spellings in the Dataset:

In this sub-task, we correct the sentences in queries dataframe and display the sentences which were wrong. We have used parts of NLP Pipeline, namely *NER* and *POS Tagging* for recognising proper nouns and names of entities. We are ignoring such words by not passing them through the spellChecker. We are using the builtin *SpellChecker* package for checking spellings of words. The faulty sentences are stored in a dictionary for the purpose of displaying them later. The entire flow can be displayed in Python as follows:

```
# Correction of spellings in the dataset
from spellchecker import SpellChecker

spell = SpellChecker()
nlp = spacy.load("en_core_web_sm")

corrections = {}
def correctSpellings(sentence):
    doc = nlp(sentence)
    newList = []
    flag = 0
    for token in doc:
        if token.pos_ == 'PROPN' or token.ent_type_ != '':
            newList.append(token.text)
            continue
        else:
            misspelt = spell.unknown([token.text])
            if misspelt != set():
                newList.append(str(spell.correction(token.text)))
                flag = 1
            else:
                newList.append(token.text)
    if(flag):
        corrections[sentence] = ' '.join(newList)
    return newList

def makeSentence(listOfWords):
    return ' '.join(listOfWords)

queries['correctList'] = queries['pure'].apply(correctSpellings)
queries['correctSentence'] = queries['correctList'].apply(makeSentence)

for key, value in corrections.items():
    print(f"\033[1mOriginal:\033[0m {key}\n\033[1mCorrection:\033[0m {value}\n")
```

1.1.4 Tokenization and Removal of most frequent and least frequent words from the Vocabulary

In the first part of the sub-task, we tokenize the sentences using the standard *en_core_web_sm* spaCy NLP pipeline. We define a function named *deriveTokens* for the standard tokenization of the sentences into tokens and then apply it to both the *data* and *queries* dataframes. This function returns a list of tokens for each cell in the dataframes. Since we are required to remove words with a frequency of more than 85% of the total number of documents and tokens with a frequency of

less than 5, we store all unique words in a set. Using this set, we construct a map with a mapping between each unique word (type) and its frequency, essentially forming the vocabulary for the entire task. Using List Comprehension, we filter out words that are not part of the vocabulary, obtaining the required vocabulary for the task. Finally, we convert the list of strings into space-separated sentences and use the TFIDF Vectorizer from the *sklearn* library to vectorize the sentences. This process is applied to both the *queries* and *df* dataframes.

We removed the most frequent tokens from the vocabulary, as these are often stopwords that occur very frequently in sentences and do not contribute much contextual value. Similarly, we removed the least frequent words, as they may be too specific to a small subset of documents, potentially causing issues like overfitting.

The standard Python code for the described process can be illustrated as follows:

```
# First of all we need to load the NLP spaCy pipeline
nlp = spacy.load("en_core_web_sm")

# Next in order to tokenize the words into a list, we use the mentioned below function for dividing the words into tokens, for each cell of the dataframe a list is returned
def deriveTokens(sentence):
    doc = nlp(sentence)
    tokensList = []

    for token in doc:
        word = token.text
        tokensList.append(word)

    return tokensList


# Now we apply the above mentioned function to the dataframes
df['tokensList'] = df['pure'].apply(deriveTokens)
queries['tokensList'] = queries['pure'].apply(deriveTokens)

# Next we shall generate the vocabulary and for that purpose we first need a collection of all tokens used in the docs database
def performFiltrationOfWords(df):
    collectionOfAllWords = []

    for tokens in df['tokensList']:
        for eachToken in tokens:
            collectionOfAllWords.append(eachToken)

    collectionSet = set(collectionOfAllWords)
    mapCollections = {}

```

```
    for token in collectionSet:
        mapCollections[token] = collectionOfAllWords.count(token)

    purifiedWords = []

    for token in collectionSet:
        if mapCollections[token] >= 5 and mapCollections[token]<= len(df) * 0.85:
            purifiedWords.append(token)

    return purifiedWords

filteredWords = performFiltrationOfWords(df)


# We generate a new column tokenList which has the words included in the given vocabulary and
    then we join those words to form sentences
df['tokensList'] = df['tokensList'].apply(lambda words: [word for word in words if word in
    filteredWords])
df['sentences'] = df['tokensList'].apply(lambda word: ' '.join(word))
queries['sentences'] = queries['tokensList'].apply(lambda word: ' '.join(word))

from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer()
tfidfVectorsForDocs = tfidf.fit_transform(df['sentences'])
tfidfVectorsForQueries = tfidf.transform(queries['sentences'])
```

1.1.5 Performing the Cosine Pairwise Similarity and then Generating a list of top K similar documents

In this sub-task, we apply the cosine-pairwise-similarity to obtain the matrix, which we shall sort column wise in ascending order (using *argsort* function) and then get the last elements in the reverse manner, according to cosine similarity values for all documents, for each query. We define two functions in order to obtain the Top-1, Top-5 and Top-10 most similar documents for given queries. Then we print them accordingly. The code for the following can be illustrated as follows:

```
# Importing the cosing similarity library from sklearn
from sklearn.metrics.pairwise import cosine_similarity

generateCosineSimilarityMatrix = cosine_similarity(tfidfVectorsForQueries, tfidfVectorsForDocs)

# In this step we get the top k similar documents
def obtainSimilarDocs(n):
    return generateCosineSimilarityMatrix.argsort(axis=1)[:, -n:][:, ::-1]

# Function for printing similar docs
```

```
def printSimilarDocs(n, documents):
    print(f"\nTop {n} Similar Docs: ")
    for i, indices in enumerate(documents, start=1):
        docText = df.iloc[indices]['doc_text']
        docIndex = df.iloc[indices]['doc_id']
        print(f"Doc ID: {docIndex} : {docText}")

# Obtaining the similarity indices and printing them subsequently
topOneSimilarInds = obtainSimilarDocs(1)
topFiveSimilarInds = obtainSimilarDocs(5)
topTenSimilarInds = obtainSimilarDocs(10)

# Running a for loop in order to print all the values:
for i, rows in queries.iterrows():
    query = rows['query_text']
    print(f"\nGiven Query: {query}")

    printSimilarDocs(1, topOneSimilarInds[i])
    printSimilarDocs(5, topFiveSimilarInds[i])
    printSimilarDocs(10, topTenSimilarInds[i])
```

1.1.6 Calculation of Precision@K Scores

As we are already given, the *qdrel* dataframe, which gives us the relations between the Document IDs and each Query ID , which helps us in obtaining the related documents for each query. So we can easily calculate the Precision@K scores by calculating the intersection between Top K documents and actually associated queries, dividing it by K to normalise it and then summing this over all the queries available to us. Lastly we divide this value by the number of queries in the dataset. Finally we return this score for $K = 1, K = 5, and\ K = 10$.

```
# Obtaining the P@K scores
def getPScores(n):
    pAtOneSum = 0.0
    pAtFiveSum = 0.0
    pAtTenSum = 0.0
    if n==1:
        for i, rows in queries.iterrows():
            # We find the actual number of related docs for each query and then we calculate
    the number of intersections
            getRelations = qdrel[qdrel['query_id'] == rows['query_id']]['doc_id']
            generateSetOfRelations = set(getRelations)
            topOneDocs = topOneSimilarInds[i]
            dataAtOne = set()
            for j, indices in enumerate(topOneDocs, start=1):
                dataAtOne.add(df.iloc[indices]['doc_id'])
```

```
15
16              pAtOne = len(generateSetOfRelations.intersection(dataAtOne)) / 1
17              pAtOneSum += pAtOne
18              queryLength = len(queries)
19          return pAtOneSum/queryLength
20
21      if n==5:
22          for i, rows in queries.iterrows():
23              getRelations = qdrel[qdrel['query_id'] == rows['query_id']]['doc_id']
24              generateSetOfRelations = set(getRelations)
25              topFiveDocs = topFiveSimilarInds[i]
26              dataAtFive = set()
27              for j, indices in enumerate(topFiveDocs, start=1):
28                  dataAtFive.add(df.iloc[indices]['doc_id'])
29
30              pAtFive = len(generateSetOfRelations.intersection(dataAtFive)) / 5
31              pAtFiveSum += pAtFive
32              queryLength = len(queries)
33          return pAtFiveSum/queryLength
34
35      if n==10:
36          for i, rows in queries.iterrows():
37              getRelations = qdrel[qdrel['query_id'] == rows['query_id']]['doc_id']
38              generateSetOfRelations = set(getRelations)
39              topTenDocs = topTenSimilarInds[i]
40              dataAtTen = set()
41              for j, indices in enumerate(topTenDocs, start=1):
42                  dataAtTen.add(df.iloc[indices]['doc_id'])
43
44              pAtTen = len(generateSetOfRelations.intersection(dataAtTen)) / 10
45              pAtTenSum += pAtTen
46              queryLength = len(queries)
47          return pAtTenSum/queryLength
48
49  pAOne = getPScores(1)
50  pAFive = getPScores(5)
51  pATen = getPScores(10)
52
53  print(f"The Avg Precision@1 Score is as follows: {pAOne:.5f}")
54  print(f"The Avg Precision@5 Score is as follows: {pAFive:.5f}")
55  print(f"The Avg Precision@10 Score is as follows: {pATen:.5f}")
```

1.1.7 Results of Task-1:

The result of Task-1 can be illustrated as follows:

```
The Avg Precision@1 Score is as follows: 0.59000
The Avg Precision@5 Score is as follows: 0.19000
The Avg Precision@10 Score is as follows: 0.10100
```

FIGURE 1.1: Precision@K scores for Task-1

1.2 Task-2

1.2.1 Performance of Stemming and its Inference

It was mentioned in the problem statement, we were required to use stemming function from *spaCy* library, but there isn't any such stemming library in *spaCy*. Therefore I have used *PorterStemmer()* function from *NLTK* library for stemming. Stemming is a text normalization technique used in NLP. It is used to reduce given words to their respective root form according to rules of the language. But it is an algorithmic library since it doesn't take care exact aspects of the language.

For instance, if we consider the word "eating", it will be reduced to its base form "eat", but on the other hand the word "ate" wouldn't be reduced to its base form "eat" if we perform stemming. In the pre-processing of the text, while tokenization of the words, we actually stem the words to their root form. After the performance of stemming, we perform the same tasks as before of using the TFIDF Vectorizer, cosine similarity and calculations of Precision Scores. The same thing can be illustrated (performance of stemming) in the code given below:

```
# Importing the NLTK library to perform stemming
from nltk.stem import PorterStemmer
st = PorterStemmer()

# Usage of Stemming
def performStemmingAndTokenize(sentence):
    doc = nlp(sentence)
    tokensList = []

    for token in doc:
        word = st.stem(token.text)
        tokensList.append(word.lower())

    return tokensList

df['stemmedTokens'] = df['pure'].apply(performStemmingAndTokenize)
queries['stemmedTokens'] = queries['pure'].apply(performStemmingAndTokenize)
```

1.2.2 Results of Task-2a (Stemming):

The result of Task-2a (Stemming) can be illustrated as follows:

```
The Avg Precision@1 Score is as follows: 0.69000
The Avg Precision@5 Score is as follows: 0.20000
The Avg Precision@10 Score is as follows: 0.10900
```

FIGURE 1.2: Precision@K scores for Task-2a

We can certainly observe that the score has improved compared to the previous case, this is because of the reduction of the words to their base forms which helps in reduction of dimensionality of the data (which wasn't the case in the previous task) Also this improvement might also be because of better text matching, because the probability of matching to similar words is increased when the words are reduced to their base forms.

1.2.3 Performance of Lemmatization and its Inference

For the performance of Lemmatization, I have added the Lemmatizer to the NLP Pipeline that I have used in the previous tasks. Lemmatization is a text normalization technique used in NLP. It is used to reduce given words to their respective root form according to rules of the language. Unlike stemming, this actually reduces the words to their base words strictly according to the rules of the given language.

For instance, if we consider the word "eating", the word will be reduced to its base form "eat" in both stemming and lemmatization, but on the other hand the word "ate" wouldn't be reduced to its base form "eat" in case of stemming but it will be reduced to "eat" in case of lemmatization. In the pre-processing of the text, while tokenization of the words, we actually lemmatize the words to their root form. After the performance of lemmatization, we perform the same tasks as before of using the TFIDF Vectorizer, cosine similarity and calculations of Precision Scores. The same thing (performance of lemmatization) can be illustrated in the code given below:

```
# Performance of Lemmatization
def performLemmatizationAndTokenize(sentence):
    doc = nlp(sentence)
    tokensList = []

    for token in doc:
        tokensList.append(token.lemma_.lower())

    return tokensList
```

```
10
11 df['lemmatizedTokens'] = df['pure'].apply(performLemmatizationAndTokenize)
12 queries['lemmatizedTokens'] = queries['pure'].apply(performLemmatizationAndTokenize)
```

1.2.4 Results of Task-2b (Lemmatization):

The result of Task-2b (Lemmatization) can be illustrated as follows:

```
The Avg Precision@1 Score is as follows: 0.71000
The Avg Precision@5 Score is as follows: 0.19600
The Avg Precision@10 Score is as follows: 0.10800
```

FIGURE 1.3: Precision@K scores for Task-2b

We can certainly observe that the score has improved a bit compared to the previous case, this is because of the reduction of the words to their actual base forms which helps in reduction of dimensionality of the data (which wasn't the case in the previous task) Also this improvement might also be because of better text matching, because the probability of matching to similar words is increased when the words are reduced to their base forms. Also rather than performance of the task, we are performing the task strictly according to rules of the language, as illustrated in the instance given above. This enables to have better matching than stemming. Therefore the performance in general is better than stemming.

1.3 Task-3

1.3.1 Improving the Performance by taking into account Named Entity Recognition (NER) and Parts-Of-Speech (POS) Tagging

As it was already mentioned in the problem statement, we need to multiply a factor of 2 to all the cells containing nouns and 4 to all cells containing names of named entities. For performing this task, we use the NER and POS Tagging features of the spaCy library, we add them to the NLP pipeline. After the calculation of TFIDF Vector of features, we have a function associated which gives the names of all columns of the TFIDF Matrix which goes by the name *.get_feature_names_out()*. So we iterate over this list and modify the cells of the tfidf matrix if we

come across a noun or the name of the entity accordingly. After this we re-apply the cosine similarity and then we obtain the Precision@K scores as previously used, the above can be illustrated by the following Python code:

```
# Usage of TFIDF Vectorizer to vectorize the given dataset
tfidf = TfidfVectorizer()
tfidfVectorsForDocs = tfidf.fit_transform(df['lemmatizedSentences'])
tfidfVectorsForQueries = tfidf.transform(queries['lemmatizedSentences'])

tfidf_tokens = tfidf.get_feature_names_out()
listOfWords = []

for word in tfidf_tokens:
    listOfWords.append(word)

listOfWords = list(nlp.pipe(listOfWords))

nounFactor = 2
nerFactor = 4

# Multiplication of a factor of 2 for proper nouns and 4 for named entities
for i in range(len(listOfWords)):
    word = listOfWords[i]
    for token in word:
        if token.pos_ == "NOUN":
            tfidfVectorsForDocs[:, i]*=nounFactor
            tfidfVectorsForQueries[:, i]*=nounFactor
        elif token.ent_type_ != '':
            tfidfVectorsForDocs[:, i]*=nerFactor
            tfidfVectorsForQueries[:, i]*=nerFactor

# Calculation of the cosine Similarity Scores
generateCosineSimilarityMatrix = cosine_similarity(tfidfVectorsForQueries, tfidfVectorsForDocs)

topOneSimilarInds = obtainSimilarDocs(1)
topFiveSimilarInds = obtainSimilarDocs(5)
topTenSimilarInds = obtainSimilarDocs(10)

# Calculation of Precision@K scores
pAOne = getPScores(1)
pAFive = getPScores(5)
pATen = getPScores(10)

print(f"The Avg Precision@1 Score is as follows: {pAOne:.5f}")
print(f"The Avg Precision@5 Score is as follows: {pAFive:.5f}")
print(f"The Avg Precision@10 Score is as follows: {pATen:.5f}")
```

1.3.2 Results of Task-3 (NER and POS Tagging):

The result of Task-3 (NER and POS Tagging) can be illustrated as follows:

```
The Avg Precision@1 Score is as follows: 0.71000
The Avg Precision@5 Score is as follows: 0.19000
The Avg Precision@10 Score is as follows: 0.10200
```

FIGURE 1.4: Presision@K scores for Task-3

We can observe that the score is better than the previous cases. This can be attributed because of the following reasons:

1. **Relevant Information is captured easily:** We know that Proper Nouns and Named Entities carry the most important information in the sentence that is relevant to the context of the sentence, by assigning more weightage, we focus on the importance of these terms which helps in improving the overall performance.

2. **Reduction of influence of Stopwords:** Stopwords are the most frequently used function words that are relevant to the overall grammatical stricture of the sentence but don't contribute much contextually. Therefore if we assign weights to important words, then we reduce the influence of such stop words.

1.4 Task-4

1.4.1 Improving the Performance by taking into account other factors relevant to the performance

We observe the following problems while fitting the data to the TFIDF Vectorizer:

1. **Removal of Important words from the Vocabulary:** If we print the vocabulary of the entire data set then we observe that most of the important words which are quite relevant to the context of the sentence, would definitely help in matching similar sentences together have frequency less than 5. Therefore as we are removing the tokens with frequency less than 5 these words are lost forever and hence this poses a problem in matching the sentences.

2. **Not taking care of stopwords having less than 85% frequency:** Stopwords are the most frequently used function words that are relevant to the overall grammatical stricture of

the sentence but don't contribute much contextually. But if we actually print the vocabulary, we can observe that not all stopwords have frequency ore than 85% in the data. Therefore there is requirement to remove them manually.

3. **Not taking care of case sensitivity:** Some of the words are in all uppercase and these are mapped differently from the ones in all lowercase. Contextually they mean the same for the sentence, so we need to ignore the case of the tokens, which has not been done in the original problem statement.

Solutions to the above problems can be stated as follows:

1. **Keeping the relevant words with low frequency:** Instead of generating a filter to remove words with frequency greater than 85% or less than 5, we ignore this step and directly use the given vocabulary that we can contruct from the original set of words.

2. **Removing Stopwords:** Since we have implemented the previous step, we need to remove the function words that don't contribute much to the overall context of the sentence, therefore while passing the sentences to a TFIDF Vectorizer, we set the parameters: analyzer='word' and stop_words= 'english'. This enables us to remove all the frequently used stopwords in the dataset, which can improve the overall performance.

3. **Switching the case to lowercase:** To avoid case related ambiguities, we keep everything in lowercase letters. ALl the relevant POS Tagging and NER is possible even if the tokens are in lowercase.

The following can be illustrated in the code below:

```
# First of all we need to load the NLP spaCy pipeline
nlp = spacy.load("en_core_web_sm")

# Next in order to tokenize the words into a list, we use the mentioned below function for
    dividing the words into tokens, for each cell of the dataframe a list is returned
def deriveNewTokens(sentence):
    doc = nlp(sentence)
    tokensList = []

    for token in doc:
        tokensList.append(token.lemma_.lower())

    return tokensList

# Now we apply the above mentioned function to the dataframes
```

```
df['tokensList'] = df['pure'].apply(deriveNewTokens)
df['newSentences'] = df['tokensList'].apply(lambda word: ' '.join(word))
queries['tokensList'] = queries['pure'].apply(deriveNewTokens)
queries['newSentences'] = queries['tokensList'].apply(lambda word: ' '.join(word))

# Usage of TFIDF Vectorizer to vectorize the given dataset
tfidf = TfidfVectorizer(analyzer='word',stop_words= 'english')
tfidfVectorsForDocs = tfidf.fit_transform(df['newSentences'])
tfidfVectorsForQueries = tfidf.transform(queries['newSentences'])

# Calculation of the cosine Similarity Scores
generateCosineSimilarityMatrix = cosine_similarity(tfidfVectorsForQueries, tfidfVectorsForDocs)

topOneSimilarInds = obtainSimilarDocs(1)
topFiveSimilarInds = obtainSimilarDocs(5)
topTenSimilarInds = obtainSimilarDocs(10)

# Calculation of Precision@K scores
pAOne = getPScores(1)
pAFive = getPScores(5)
pATen = getPScores(10)

print(f"The Avg Precision@1 Score is as follows: {pAOne:.5f}")
print(f"The Avg Precision@5 Score is as follows: {pAFive:.5f}")
print(f"The Avg Precision@10 Score is as follows: {pATen:.5f}")
```

1.4.2 Results of Task-4 (Customizations):

The result of Task-4 (Customizations) can be illustrated as follows:

```
The Avg Precision@1 Score is as follows: 0.89000
The Avg Precision@5 Score is as follows: 0.22000
The Avg Precision@10 Score is as follows: 0.11700
```

FIGURE 1.5: Precision@K scores for Task-4

Now because of the customizations we can clearly observe the boosts in the overall behaviour and performance of the task.

Chapter 2

Conclusion

In this assignment, our main goal was to group similar sentences together using various NLP methods as provided to us by the SpaCy library. We have used TFIDF Vectorizor to vectorize the given dataset. We have pre-processed the data, removed all characters apart from alphanumerics and whitespaces. Next we rectified the incorrectly spelt words in the dataset. We then tokenized and built a vocabulary for the task and then we used TFIDF Vectorizer to vectorize the sentences, so that we can use cosine similarity and calculate the Precision@K scores.

In the subsequent tasks, we performed stemming and lematization on the tokens and improved the overall performance. We also provided an additional weightage to important and relevant words in the dataset using NER and POS Tagging. Lastly we customized the data in a way that would improve the overall performance which we are quite successful at implementing.