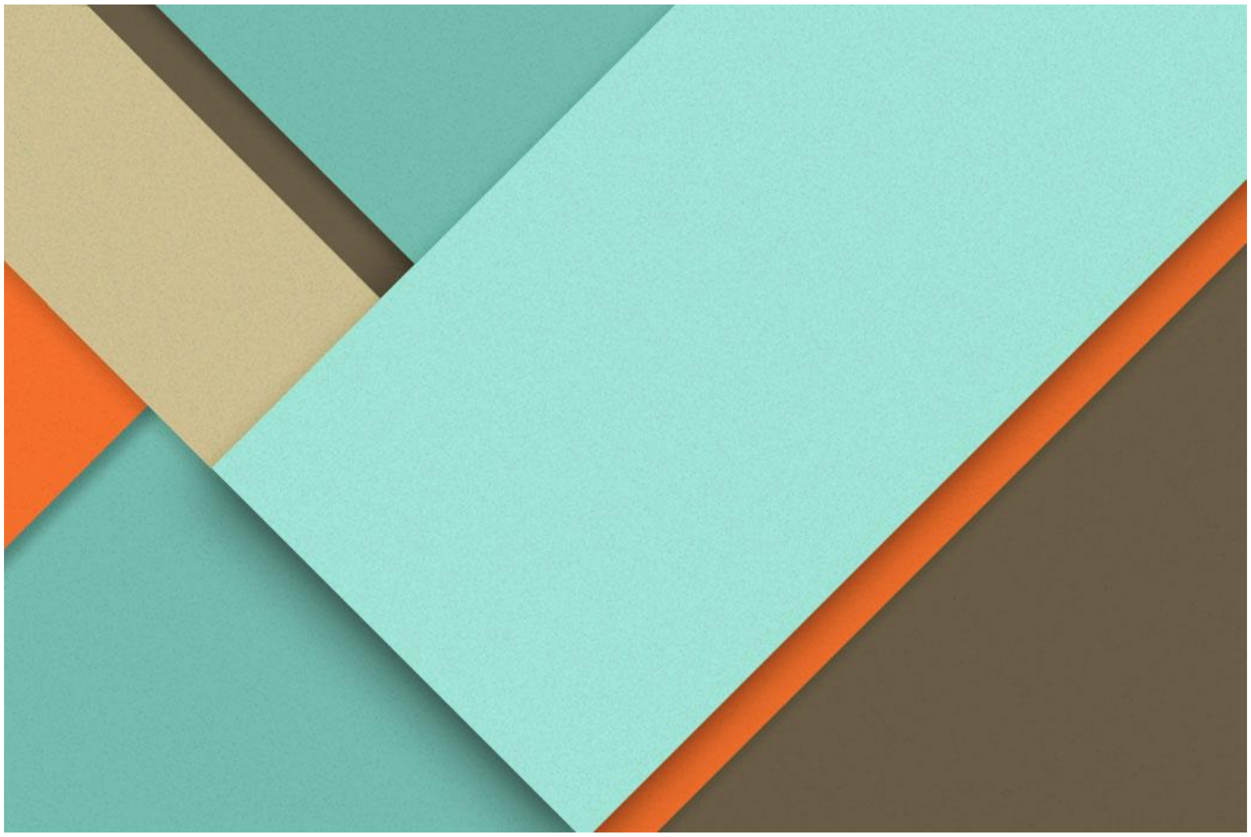


# Pdana Task 1:Sentiment Analysis Using RNNs

20.09.2023

—

Siyabulela Mathe

## A reference to the data set source.

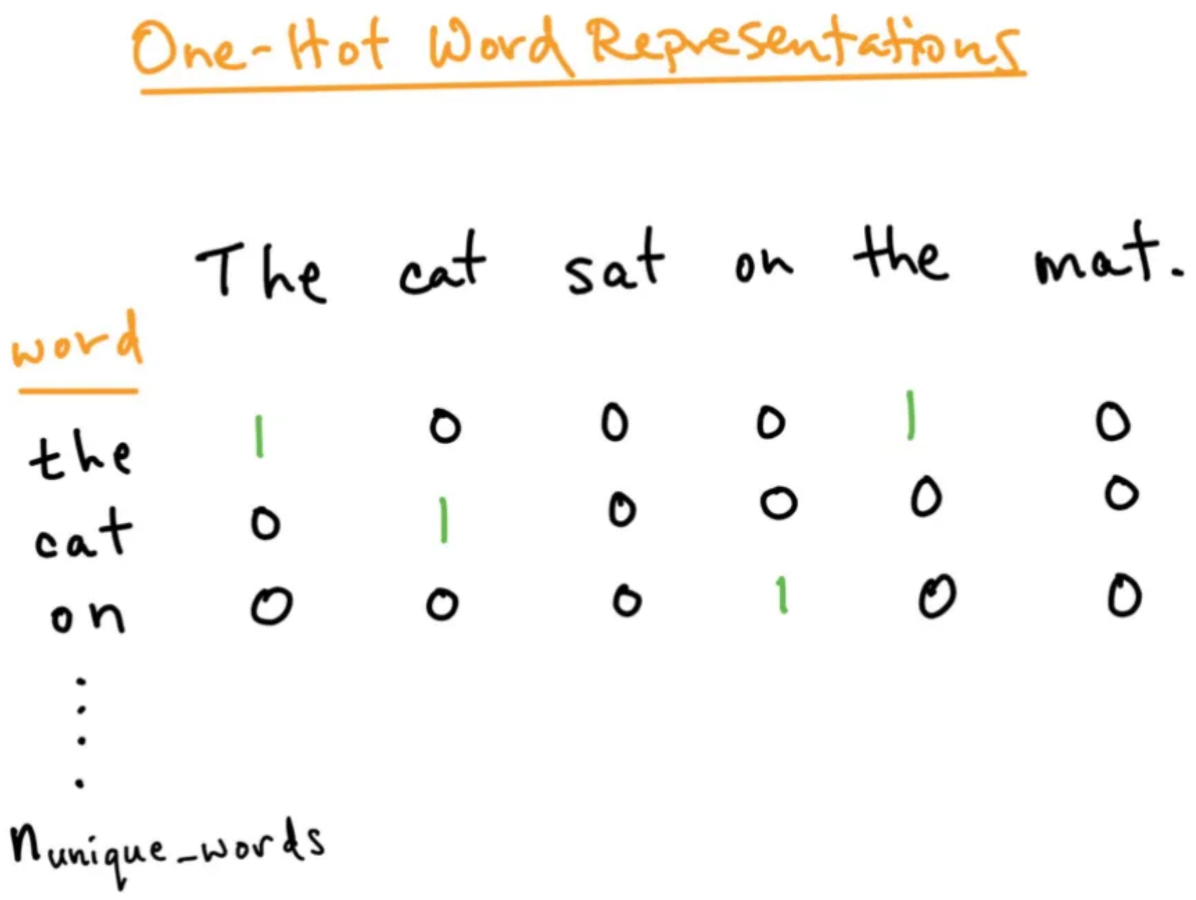
Justifying recommendations using distantly-labeled reviews and fined-grained aspects

Jianmo Ni, Jiacheng Li, Julian McAuley

Empirical Methods in Natural Language Processing (EMNLP), 2019

Amazon product review text data based on general appliances. The data card was sourced as gzipped json file(json.gz) then extracted using the windows tar command.

## An explanation of why the data set is appropriate for processing using a recurrent neural network.



One-Hot Word Representations

| word | The | cat | sat | on | the | mat. |
|---|---|---|---|---|---|---|
| the | 1 | 0 | 0 | 0 | 1 | 0 |
| cat | 0 | 1 | 0 | 0 | 0 | 0 |
| on | 0 | 0 | 0 | 1 | 0 | 0 |
| ⋮ | | | | | | |
| $n_{unique\_words}$ | | | | | | |

One of the most exciting advances in machine learning is the development of ways to teach machines to understand human language. This field of machine learning is called Natural Language Processing (NLP).

NLP is a broad field that encompasses many different tasks, such as machine translation, text summarization, and question answering. NLP systems are able to learn the patterns of human language and use that knowledge to perform these tasks.

However, computers aren't able to comprehend and read language the same way we do. Computers need to transform these words into numbers using mathematical frameworks to make sense of the information fed to these models.

The traditional transformation techniques aren't suitable for RNNs because they have no emphasis on order, as RNNs thrive on sequential data.

A product review data set is appropriate for processing using a recurrent neural network (RNN) because RNNs are well-suited for modeling sequential data, such as text. Product reviews are typically sequential data, with each sentence in the review building on the previous sentences. RNNs are able to learn long-term dependencies in sequential data, which is important for understanding the meaning of product reviews.

In addition, RNNs are able to learn the relationships between words in a sentence. This is important for understanding the sentiment of a product review, as the sentiment of a review is often conveyed by the relationships between the words in the review.

Here are some specific examples of how RNNs can be used to process product review data:

- Sentiment analysis: RNNs can be used to predict the sentiment of a product review (positive, negative, or neutral). This information can be used by businesses to improve their products and services, and by consumers to make informed decisions about which products to purchase.
- Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) that are well-suited for sentiment analysis of product reviews. LSTMs are able to learn long-term dependencies in sequential data, which is important for understanding the sentiment of a product review, as the sentiment of a review is often conveyed by the relationships between the words in the review.

## Data Analysis Steps

### 1. Defining the problem

#### I. Why is the dataset suitable for sentiment analysis

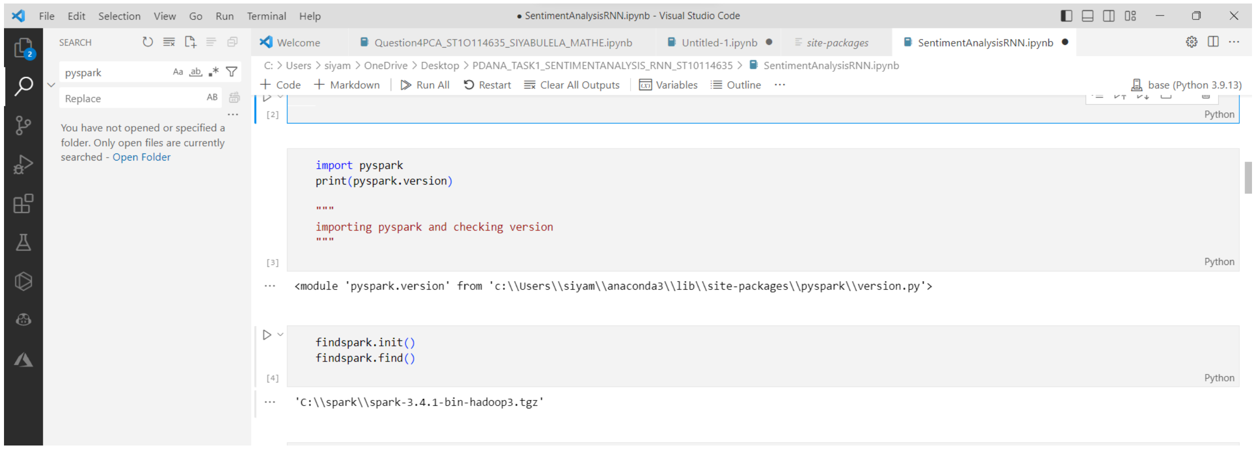
Product reviews can be ultimately used for sentiment analysis to give businesses better insight on which products are performing well on the market, and which aren't. Especially those that aren't, businesses can get a deeper insight on what customers think about those products and how they should be improved. Product review sentiment analysis is the process of extracting and analyzing the sentiment of customer reviews of a product(Olsson,2009;Nabi,2008). By understanding how customers feel about a product, businesses can identify areas where they can improve the product or customer experience. Product review sentiment analysis can also help businesses to identify product issues that customers are experiencing(Olsson,2009;Nabi,2008).. This information can then be used to fix the issues and improve the product. Product review sentiment analysis can also be used to improve marketing campaigns. For example, if a business finds that customers who are expressing positive sentiment about a product are also likely to be interested in other products, they can target those customers with marketing campaigns for those products(Olsson,2009;Nabi,2008).. Product review sentiment analysis can also help businesses to make better business decisions. For example, if a business finds that customers are expressing negative sentiment about a particular feature, they may decide to redesign or remove that feature. Ultimately, product review sentiment analysis is a pivotal tool for businesses that want to **improve their products, customer experience, and marketing campaigns.**

## 2. Data Exploration.

The data was loaded using pyspark which is a programming interface that allows developers to interact and make usage of Apache Spark. Apache Spark is an open source analytics engine used for big data workloads. It can handle both batches as well as real-time analytics and data processing workloads. It’s used for processed and analysis large amounts of data. Spark use cases in Computer Software and Information Technology and Services takes about 32% and 14% respectively in the global market. Apache Spark is designed for interactive queries on large datasets.

Essentially a Spark session was created and an SQL context was used to read the json file.

## DISCLAIMER

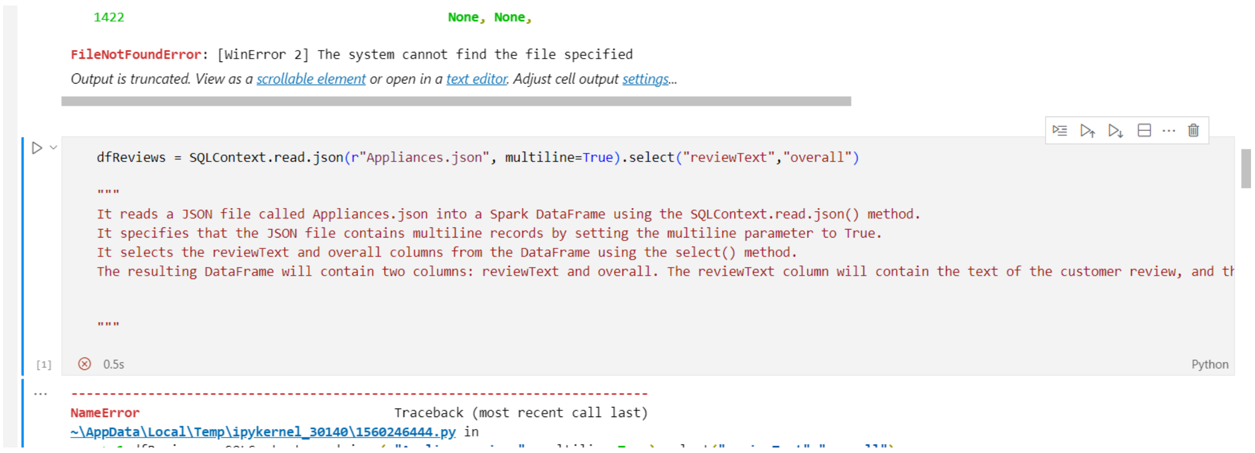


```
import pyspark
print(pyspark.version)

"""
importing pyspark and checking version
"""
```

```
<module 'pyspark.version' from 'c:\\Users\\siyam\\anaconda3\\lib\\site-packages\\pyspark\\version.py'>
```

```
findspark.init()
findspark.find()
```

```
'C:\\spark\\spark-3.4.1-bin-hadoop3.tgz'
```



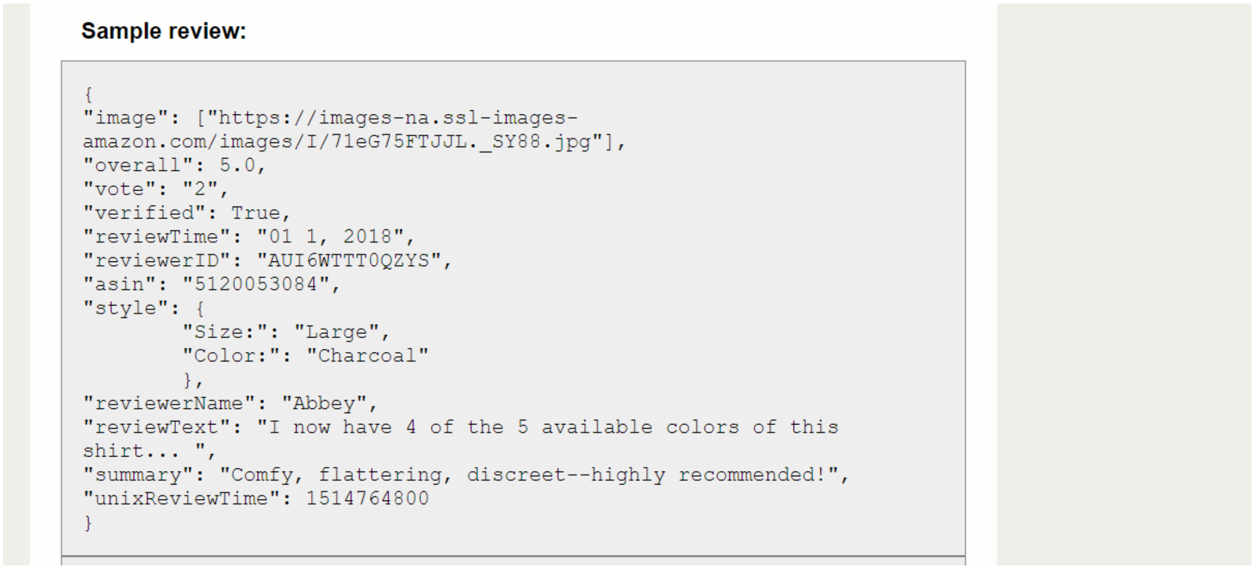
```
    1422                                          None, None,

FileNotFoundError: [WinError 2] The system cannot find the file specified
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
dfReviews = SQLContext.read.json(r"Appliances.json", multiline=True).select("reviewText","overall")

"""
It reads a JSON file called Appliances.json into a Spark DataFrame using the SQLContext.read.json() method.
It specifies that the JSON file contains multiline records by setting the multiline parameter to True.
It selects the reviewText and overall columns from the DataFrame using the select() method.
The resulting DataFrame will contain two columns: reviewText and overall. The reviewText column will contain the text of the customer review, and th


"""
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_30140\1560246444.py in
```

***Unfortunately this was the downfall of the project as,  i installed the wrong version and according the the documentation for the version i have installed on my machine(3.4.1) ,it's unable to read json files directly. However version 3.5 and up supports the approach I wanted to take. Which will be implemented in the POE. Apart that everything should be good.  The point of data exploration is essentially to get a feel for our data. Check the number of rows and columns, which features are most likely to be important in the development of our models and ultimately if there are missing records/null values which need to be filled.***



**Sample review:**

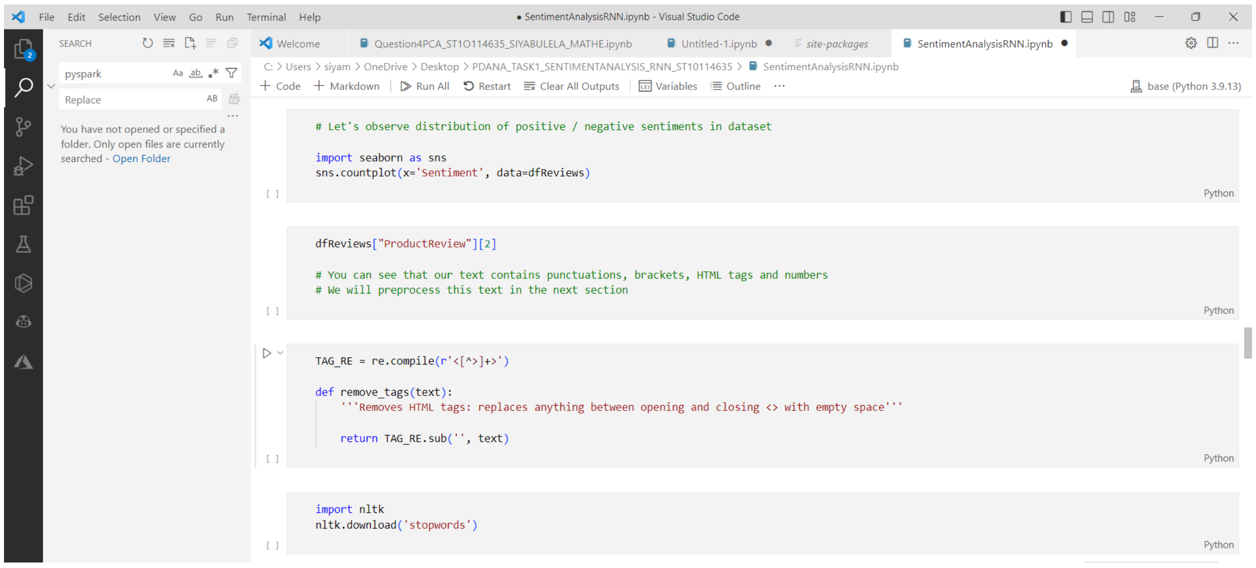
```
{
"image": ["https://images-na.ssl-images-
amazon.com/images/I/71eG75FTJJL._SY88.jpg"],
"overall": 5.0,
"vote": "2",
"verified": True,
"reviewTime": "01 1, 2018",
"reviewerID": "AUI6WTTT0QZYS",
"asin": "5120053084",
"style": {
        "Size:": "Large",
        "Color:": "Charcoal"
        },
"reviewerName": "Abbey",
"reviewText": "I now have 4 of the 5 available colors of this
shirt... ",
"summary": "Comfy, flattering, discreet--highly recommended!",
"unixReviewTime": 1514764800
}
```

## 3. Data Processing and Preparation.

Text processing in machine learning is the process of transforming unstructured text data into structured data that can be used by machine learning models. The term text processing refers to the ***automation of analyzing electronic text***. This allows machine learning models to get structured information about the text to use for analysis, manipulation of the text, or to generate new text. (Olsson,2009;Nabi,2008).

This involves a variety of tasks, such as:

- Data cleaning: This involves removing noise and errors from the text data. For example, removing punctuation,removing html tags,  removing stop words, correcting spelling mistakes, and handling inconsistencies in formatting.
- Tokenization/Transformation/Word Embedding: This involves breaking the text data into individual words or tokens. This is necessary because machine learning models can only understand numbers, so the text data needs to be converted into a numerical format.
- Feature extraction: This involves identifying the important features in the text data that will be used by the machine learning model. For example, the frequency of each word, the position of each word in the sentence, or the part of speech of each word.


File Edit Selection View Go Run Terminal Help
● SentimentAnalysisRNN.ipynb - Visual Studio Code
SEARCH
pyspark
Replace
Welcome
Question4PCA_ST10114635_SIYABULELA_MATHE.ipynb
Untitled-1.ipynb
site-packages
SentimentAnalysisRNN.ipynb
C: > Users > siyam > OneDrive > Desktop > PDANA_TASK1_SENTIMENTANALYSIS_RNN_ST10114635 > SentimentAnalysisRNN.ipynb
+ Code + Markdown Run All Restart Clear All Outputs Variables Outline
base (Python 3.9.13)

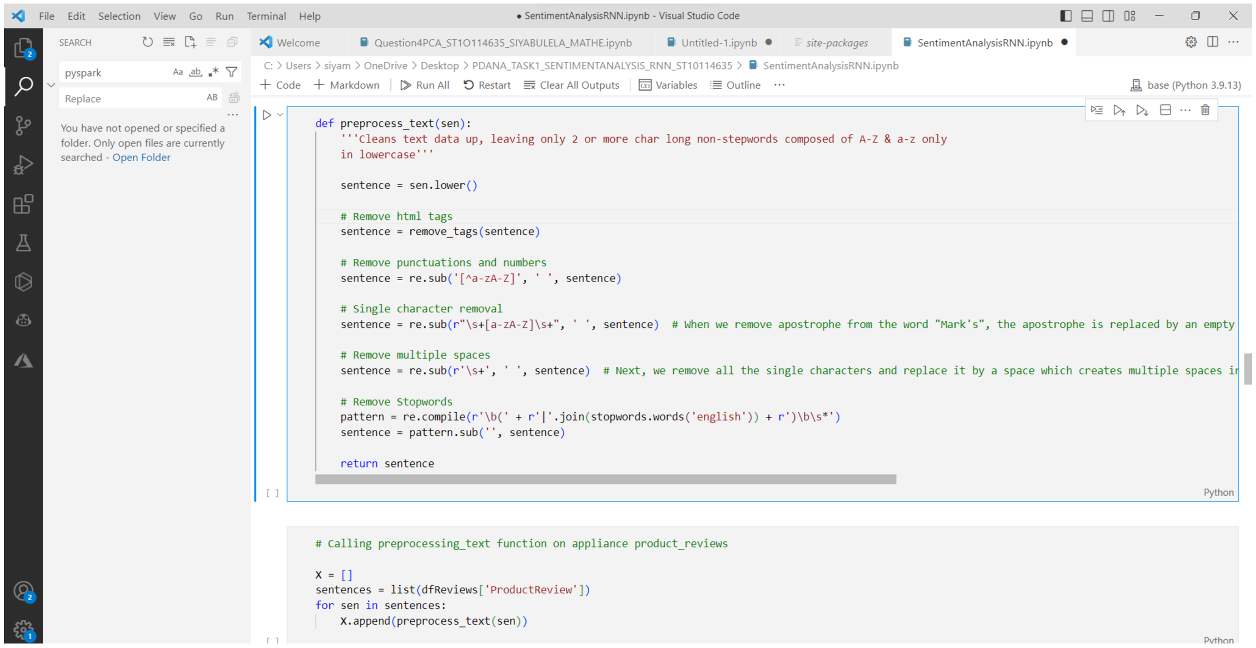
You have not opened or specified a folder. Only open files are currently searched - Open Folder

```
# Let's observe distribution of positive / negative sentiments in dataset

import seaborn as sns
sns.countplot(x='Sentiment', data=dfReviews)
```

```
dfReviews["ProductReview"][2]

# You can see that our text contains punctuations, brackets, HTML tags and numbers
# We will preprocess this text in the next section
```

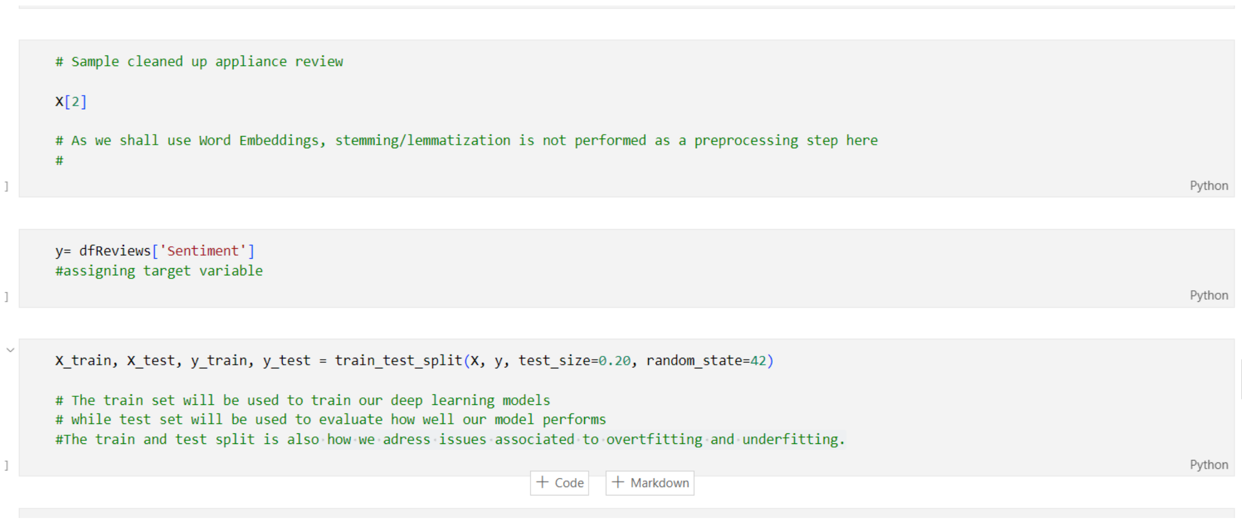
```
TAG_RE = re.compile(r'<[^>]+>')

def remove_tags(text):
    '''Removes HTML tags: replaces anything between opening and closing <> with empty space'''

    return TAG_RE.sub('', text)
```

```
import nltk
nltk.download('stopwords')
```



```
def preprocess_text(sen):
    '''Cleans text data up, leaving only 2 or more char long non-stepwords composed of A-Z & a-z only
    in lowercase'''

    sentence = sen.lower()

    # Remove html tags
    sentence = remove_tags(sentence)

    # Remove punctuations and numbers
    sentence = re.sub('[^a-zA-Z]', ' ', sentence)

    # Single character removal
    sentence = re.sub(r"\s+[a-zA-Z]\s+", ' ', sentence)  # When we remove apostrophe from the word "Mark's", the apostrophe is replaced by an empty

    # Remove multiple spaces
    sentence = re.sub(r'\s+', ' ', sentence)  # Next, we remove all the single characters and replace it by a space which creates multiple spaces ir

    # Remove Stopwords
    pattern = re.compile(r'\b(' + r'|'.join(stopwords.words('english')) + r')\b\s*')
    sentence = pattern.sub('', sentence)

    return sentence
```

```
# Calling preprocessing_text function on appliance product_reviews

X = []
sentences = list(dfReviews['ProductReview'])
for sen in sentences:
    X.append(preprocess_text(sen))
```

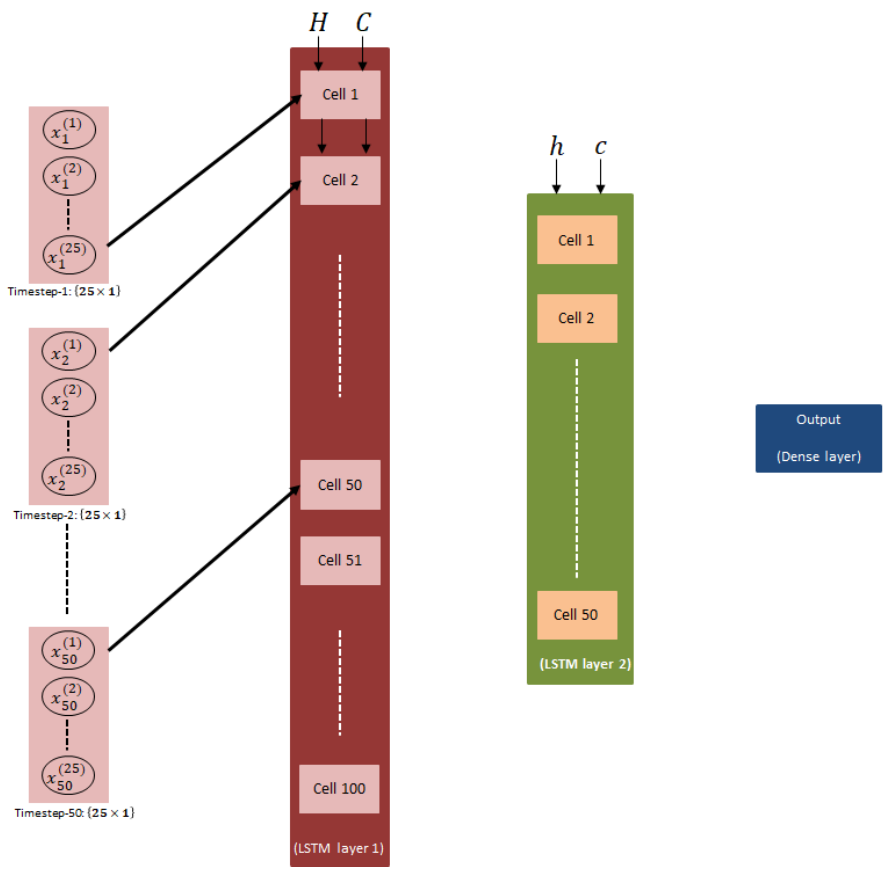


```
# Sample cleaned up appliance review

X[2]

# As we shall use Word Embeddings, stemming/lemmatization is not performed as a preprocessing step here
#
```

```
y= dfReviews['Sentiment']
#assigning target variable
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)

# The train set will be used to train our deep learning models
# while test set will be used to evaluate how well our model performs
#The train and test split is also how we adress issues associated to overfitting and underfitting.
```

## 4. Model Development.



The target variable which is the sentiment(positive or negative) , one or zero was set. The independent variable which was the review text was also set after the data had been cleaned and prepared. The data was eventually split 80/20 to address issues associated with overfitting and underfitting.

So what is overfitting and underfitting?

Overfitting and underfitting are two common problems that can occur when training machine learning models, including LSTM models used for sentiment analysis.

Overfitting occurs when a model learns the training data too well and is unable to generalize to new data. This can happen when the model is too complex or when the training data is too small.

Underfitting occurs when a model does not learn enough from the training data and is unable to make accurate predictions on new data. This can happen when the model is too simple or when the training data is too large.

This was avoided through the input of clean(non noisy data), efficient feature selection and the usage of a validation set that we can use to feed the model new data.

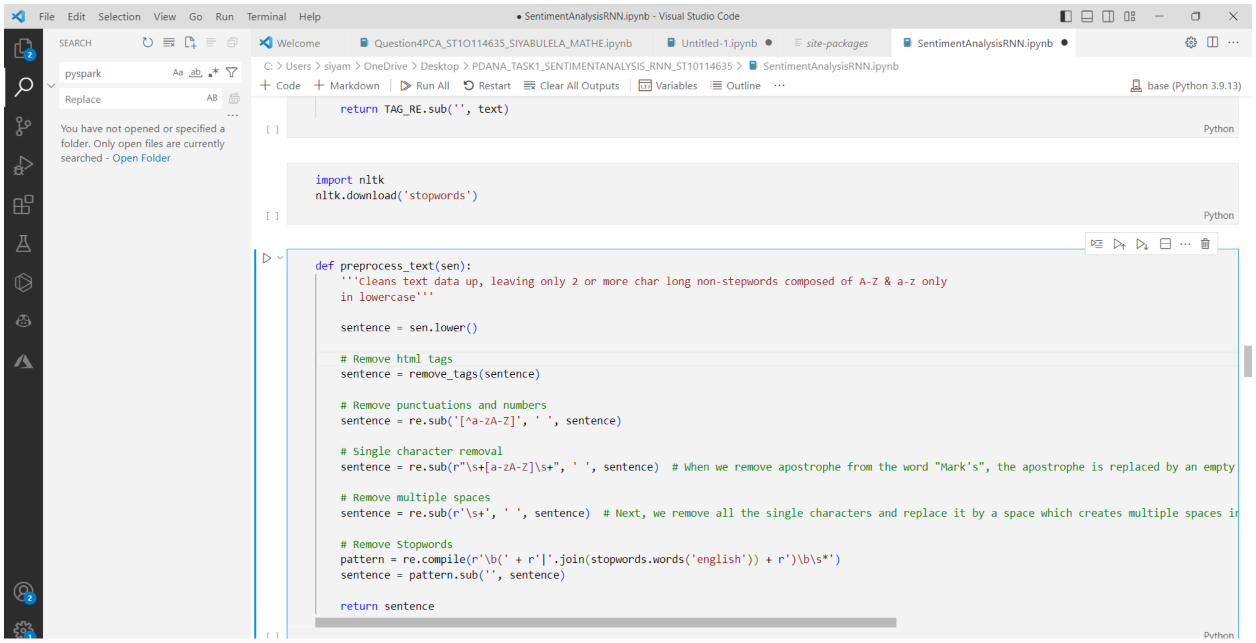
The first layer( word embedding layer)  is initialized with the pre-trained GloVe word embeddings that are stored in the embedding_matrix variable.

So what is GloVe?

GloVe, coined from Global Vectors, is a model for distributed word representation. The model is an unsupervised learning algorithm for obtaining vector representations for words. This is achieved by mapping words into a meaningful space where the distance between words is related to semantic similarity. GloVe was also used as the word representation framework for the online and offline systems designed to detect psychological distress in patient interviews.

The trainable=False parameter prevents the embedding matrix from being updated during training because we are using a pre trained word embedding library.

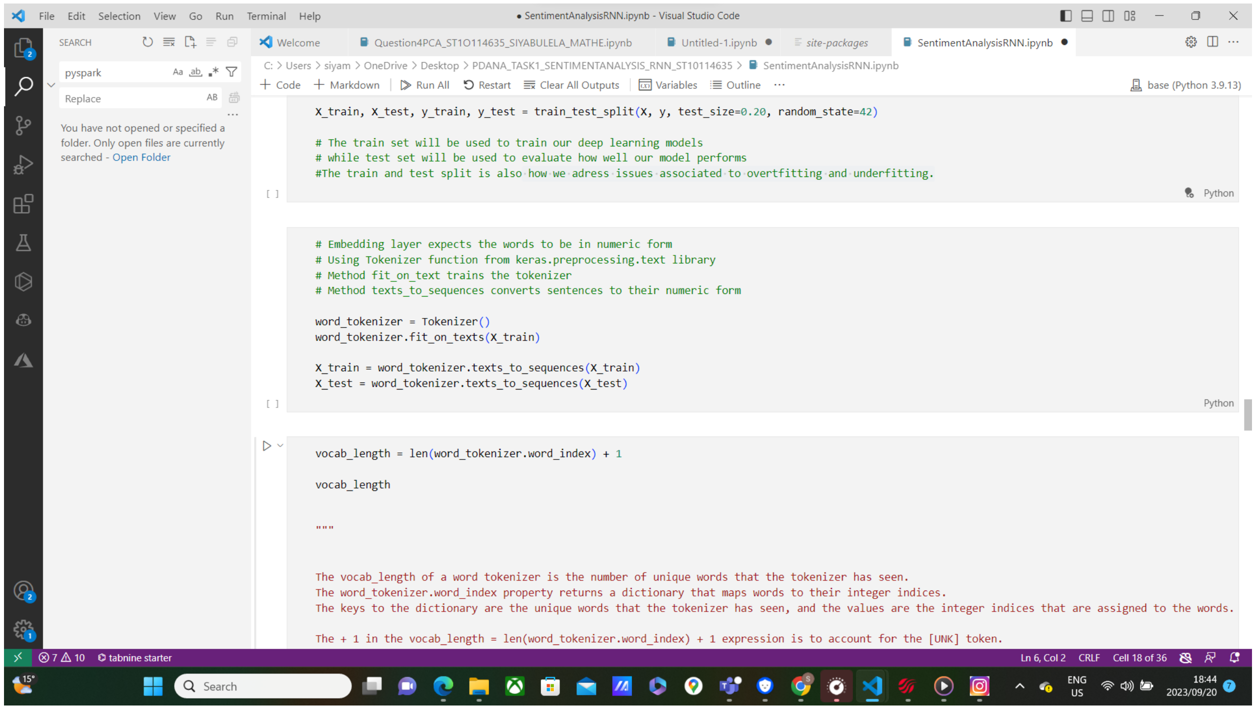
The embedding layer will convert each word in the input text to a dense vector of real numbers. The dense vector of real numbers will then be used by the model to predict the output.



```
    return TAG_RE.sub('', text)
```

```
import nltk
nltk.download('stopwords')
```

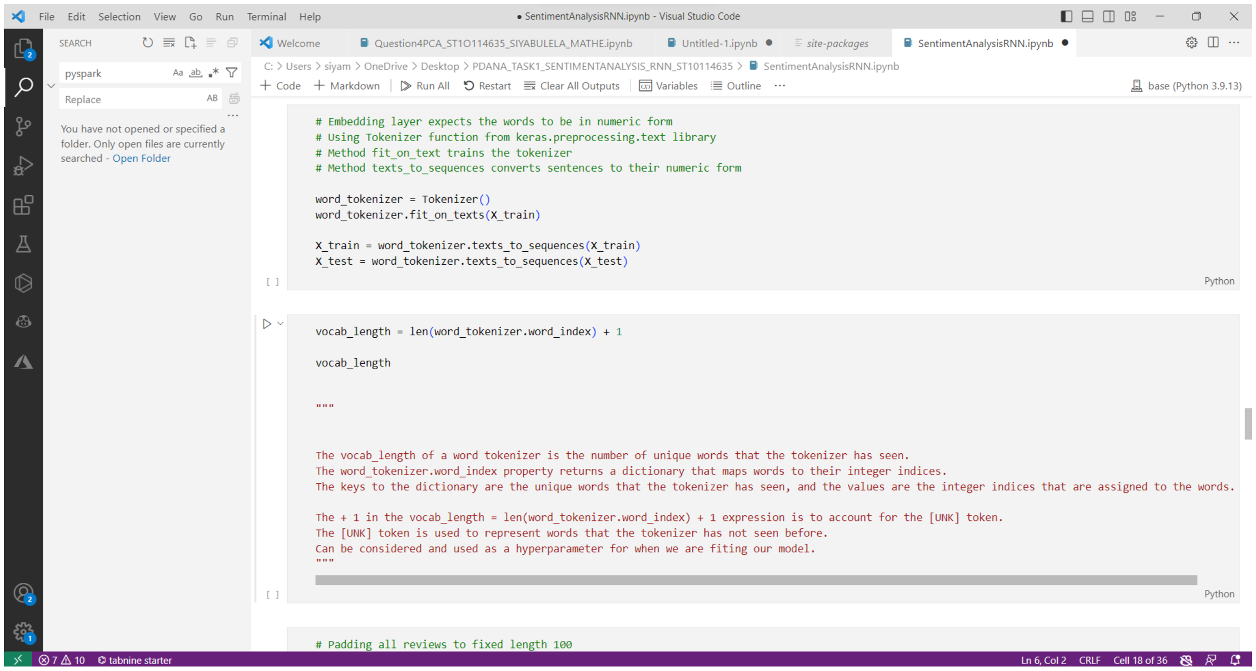
```
def preprocess_text(sen):
    '''Cleans text data up, leaving only 2 or more char long non-stepwords composed of A-Z & a-z only
    in lowercase'''

    sentence = sen.lower()

    # Remove html tags
    sentence = remove_tags(sentence)

    # Remove punctuations and numbers
    sentence = re.sub('[^a-zA-Z]', ' ', sentence)

    # Single character removal
    sentence = re.sub(r"\s+[a-zA-Z]\s+", ' ', sentence)  # When we remove apostrophe from the word "Mark's", the apostrophe is replaced by an empty

    # Remove multiple spaces
    sentence = re.sub(r'\s+', ' ', sentence)  # Next, we remove all the single characters and replace it by a space which creates multiple spaces in

    # Remove Stopwords
    pattern = re.compile(r'\b(' + r'|'.join(stopwords.words('english')) + r')\b\s*')
    sentence = pattern.sub('', sentence)

    return sentence
```



```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)

# The train set will be used to train our deep learning models
# while test set will be used to evaluate how well our model performs
#The train and test split is also how we adress issues associated to overtfitting and underfitting.
```

```
# Embedding layer expects the words to be in numeric form
# Using Tokenizer function from keras.preprocessing.text library
# Method fit_on_text trains the tokenizer
# Method texts_to_sequences converts sentences to their numeric form

word_tokenizer = Tokenizer()
word_tokenizer.fit_on_texts(X_train)

X_train = word_tokenizer.texts_to_sequences(X_train)
X_test = word_tokenizer.texts_to_sequences(X_test)
```

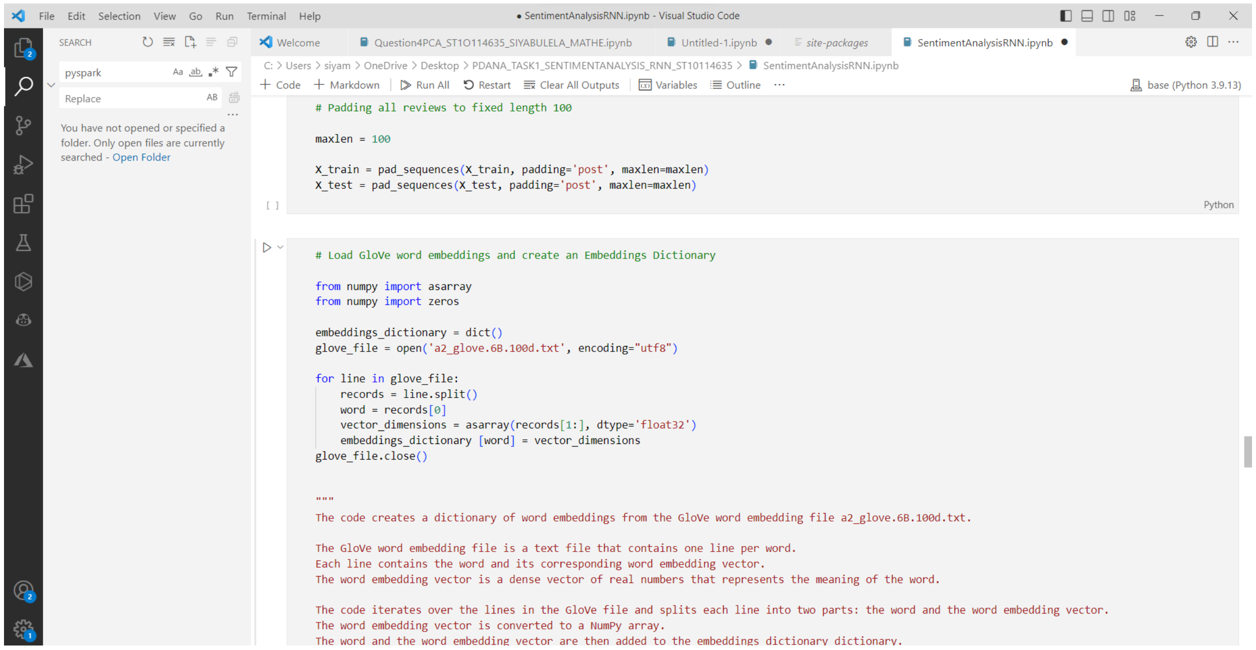
```
vocab_length = len(word_tokenizer.word_index) + 1

vocab_length


"""


The vocab_length of a word tokenizer is the number of unique words that the tokenizer has seen.
The word_tokenizer.word_index property returns a dictionary that maps words to their integer indices.
The keys to the dictionary are the unique words that the tokenizer has seen, and the values are the integer indices that are assigned to the words.

The + 1 in the vocab_length = len(word_tokenizer.word_index) + 1 expression is to account for the [UNK] token.
```



```
# Embedding layer expects the words to be in numeric form
# Using Tokenizer function from keras.preprocessing.text library
# Method fit_on_text trains the tokenizer
# Method texts_to_sequences converts sentences to their numeric form

word_tokenizer = Tokenizer()
word_tokenizer.fit_on_texts(X_train)

X_train = word_tokenizer.texts_to_sequences(X_train)
X_test = word_tokenizer.texts_to_sequences(X_test)
```

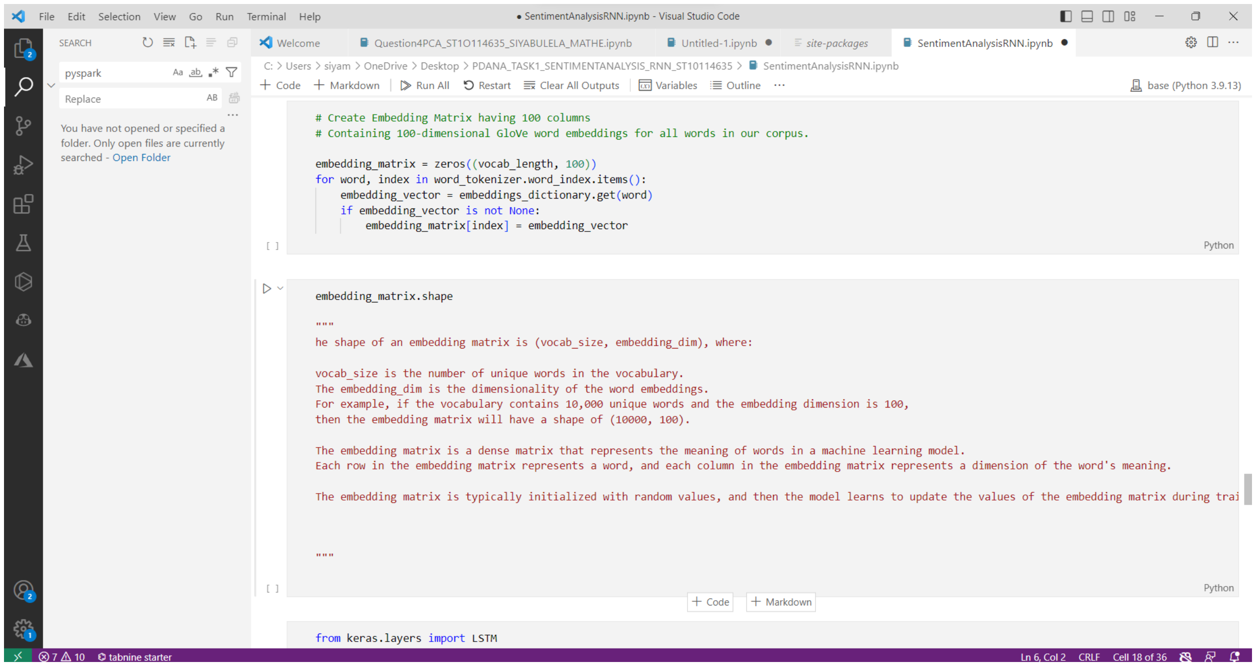
```
vocab_length = len(word_tokenizer.word_index) + 1

vocab_length


"""


The vocab_length of a word tokenizer is the number of unique words that the tokenizer has seen.
The word_tokenizer.word_index property returns a dictionary that maps words to their integer indices.
The keys to the dictionary are the unique words that the tokenizer has seen, and the values are the integer indices that are assigned to the words.

The + 1 in the vocab_length = len(word_tokenizer.word_index) + 1 expression is to account for the [UNK] token.
The [UNK] token is used to represent words that the tokenizer has not seen before.
Can be considered and used as a hyperparameter for when we are fiting our model.
"""
```

```
# Padding all reviews to fixed length 100
```



```
# Padding all reviews to fixed length 100

maxlen = 100

X_train = pad_sequences(X_train, padding='post', maxlen=maxlen)
X_test = pad_sequences(X_test, padding='post', maxlen=maxlen)
```

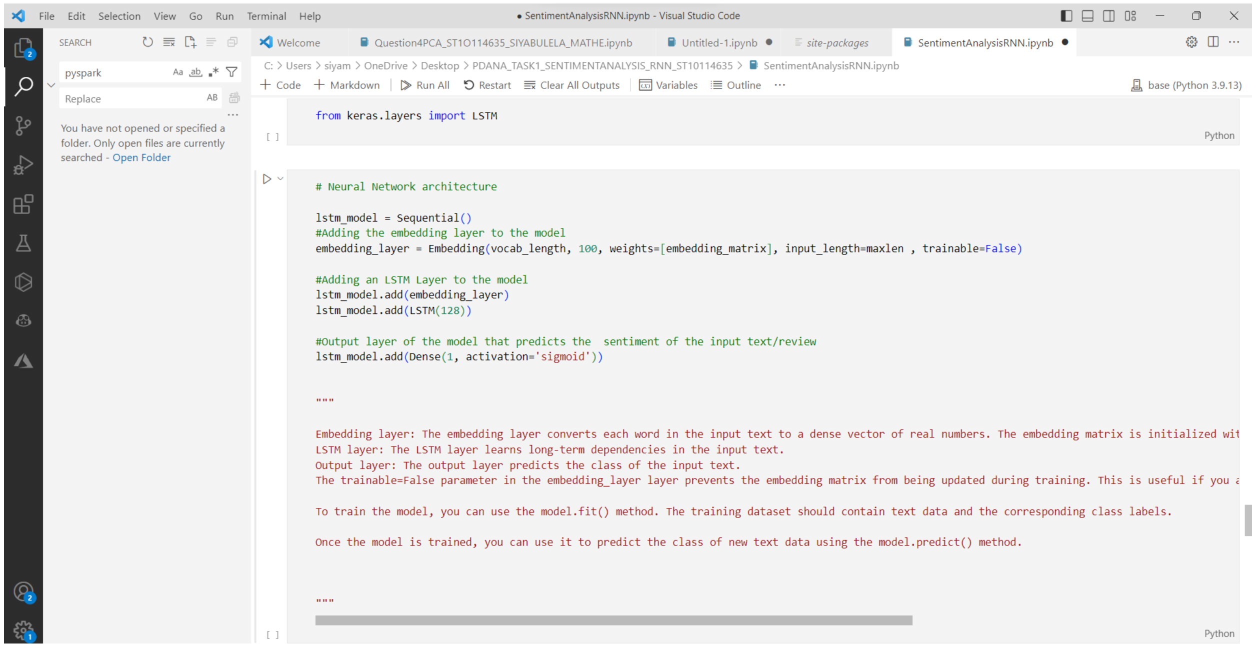
```
# Load GloVe word embeddings and create an Embeddings Dictionary

from numpy import asarray
from numpy import zeros

embeddings_dictionary = dict()
glove_file = open('a2_glove.6B.100d.txt', encoding="utf8")

for line in glove_file:
    records = line.split()
    word = records[0]
    vector_dimensions = asarray(records[1:], dtype='float32')
    embeddings_dictionary [word] = vector_dimensions
glove_file.close()


"""
The code creates a dictionary of word embeddings from the GloVe word embedding file a2_glove.6B.100d.txt.

The GloVe word embedding file is a text file that contains one line per word.
Each line contains the word and its corresponding word embedding vector.
The word embedding vector is a dense vector of real numbers that represents the meaning of the word.

The code iterates over the lines in the GloVe file and splits each line into two parts: the word and the word embedding vector.
The word embedding vector is converted to a NumPy array.
The word and the word embedding vector are then added to the embeddings dictionary dictionary.
```



```
# Create Embedding Matrix having 100 columns
# Containing 100-dimensional GloVe word embeddings for all words in our corpus.

embedding_matrix = zeros((vocab_length, 100))
for word, index in word_tokenizer.word_index.items():
    embedding_vector = embeddings_dictionary.get(word)
    if embedding_vector is not None:
        embedding_matrix[index] = embedding_vector
```

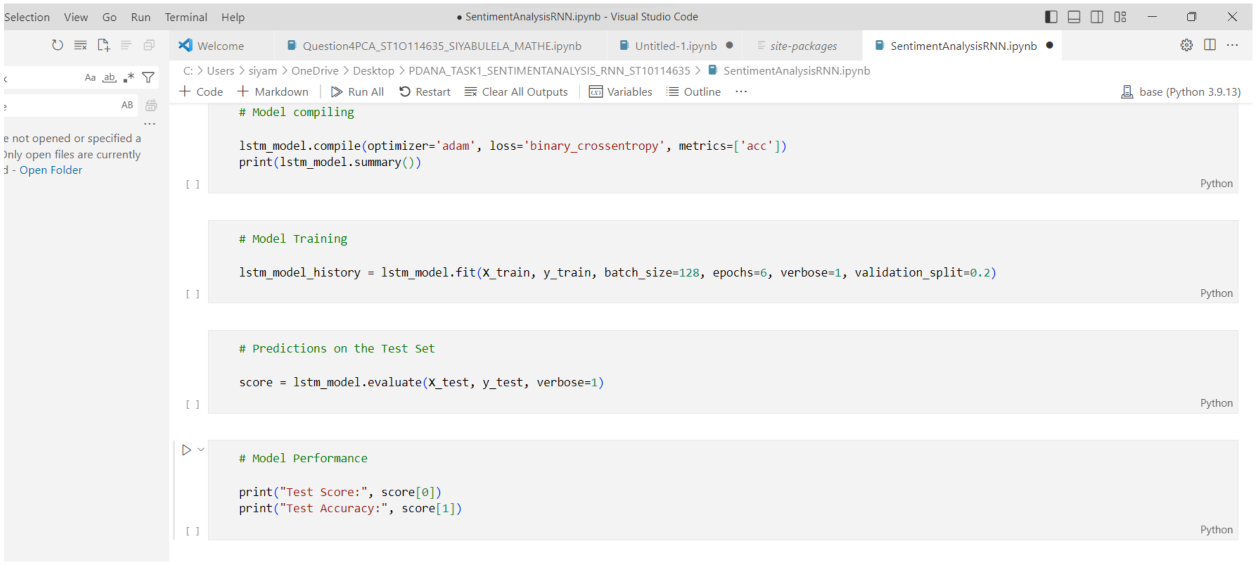
```
embedding_matrix.shape

"""
he shape of an embedding matrix is (vocab_size, embedding_dim), where:

vocab_size is the number of unique words in the vocabulary.
The embedding_dim is the dimensionality of the word embeddings.
For example, if the vocabulary contains 10,000 unique words and the embedding dimension is 100,
then the embedding matrix will have a shape of (10000, 100).

The embedding matrix is a dense matrix that represents the meaning of words in a machine learning model.
Each row in the embedding matrix represents a word, and each column in the embedding matrix represents a dimension of the word's meaning.

The embedding matrix is typically initialized with random values, and then the model learns to update the values of the embedding matrix during trai


"""
```

```
from keras.layers import LSTM
```



```
from keras.layers import LSTM
```

```
# Neural Network architecture

lstm_model = Sequential()
#Adding the embedding layer to the model
embedding_layer = Embedding(vocab_length, 100, weights=[embedding_matrix], input_length=maxlen , trainable=False)

#Adding an LSTM Layer to the model
lstm_model.add(embedding_layer)
lstm_model.add(LSTM(128))

#Output layer of the model that predicts the  sentiment of the input text/review
lstm_model.add(Dense(1, activation='sigmoid'))


"""

Embedding layer: The embedding layer converts each word in the input text to a dense vector of real numbers. The embedding matrix is initialized wit
LSTM layer: The LSTM layer learns long-term dependencies in the input text.
Output layer: The output layer predicts the class of the input text.
The trainable=False parameter in the embedding_layer layer prevents the embedding matrix from being updated during training. This is useful if you a

To train the model, you can use the model.fit() method. The training dataset should contain text data and the corresponding class labels.

Once the model is trained, you can use it to predict the class of new text data using the model.predict() method.


"""
```



```
# Model compiling

lstm_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
print(lstm_model.summary())
```

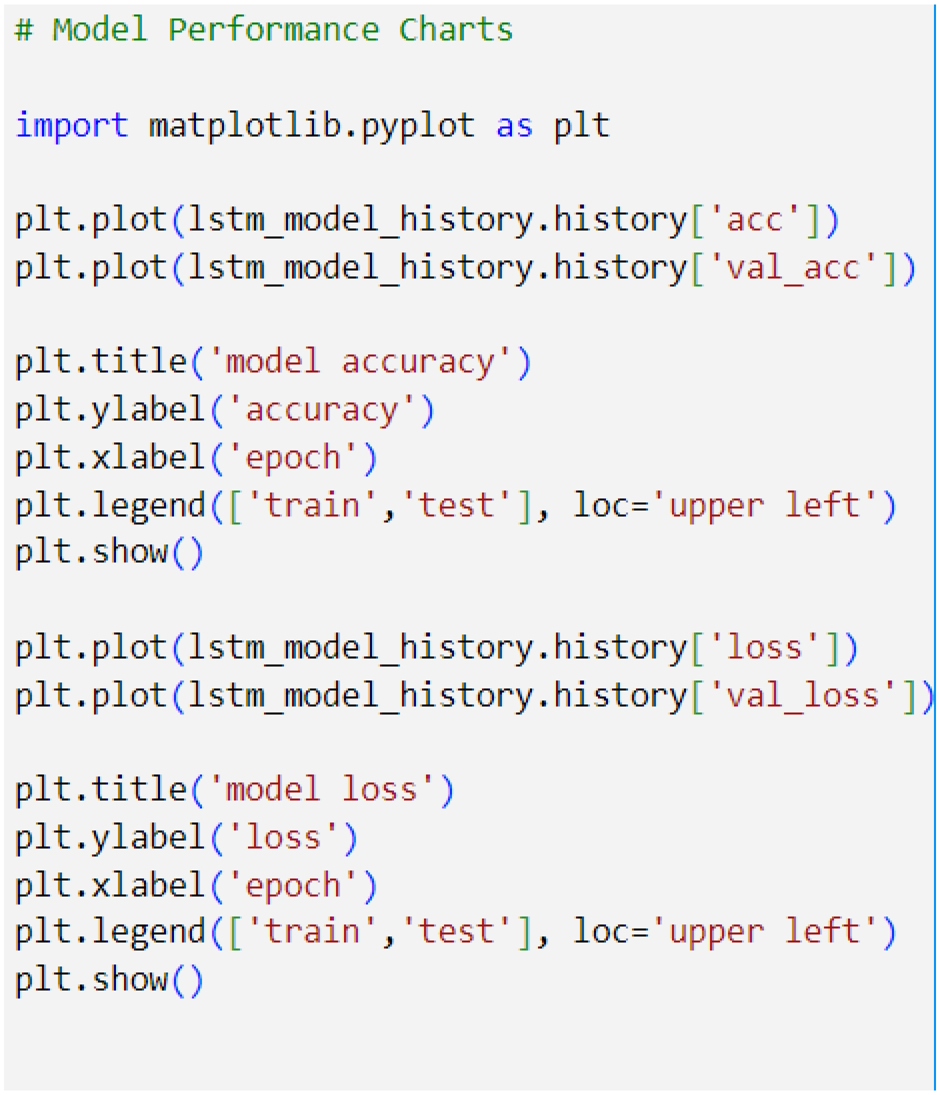
```
# Model Training

lstm_model_history = lstm_model.fit(X_train, y_train, batch_size=128, epochs=6, verbose=1, validation_split=0.2)
```

```
# Predictions on the Test Set

score = lstm_model.evaluate(X_test, y_test, verbose=1)
```

```
# Model Performance

print("Test Score:", score[0])
print("Test Accuracy:", score[1])
```

## 5. Model Evaluation.

Apart from the obvious metrics such as accuracy, f1 score, precision and recall the models were evaluated graphically. Using our built in model metrics.



```python
# Model Performance Charts

import matplotlib.pyplot as plt

plt.plot(lstm_model_history.history['acc'])
plt.plot(lstm_model_history.history['val_acc'])

plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train','test'], loc='upper left')
plt.show()

plt.plot(lstm_model_history.history['loss'])
plt.plot(lstm_model_history.history['val_loss'])

plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train','test'], loc='upper left')
plt.show()
```

The lstm_model_history.history variable contains a dictionary of metrics that were recorded during training.

The acc key in the dictionary contains the accuracy of the model on the training dataset, and the val_acc key contains the accuracy of the model on the test dataset.

The plt.plot() function is used to plot the accuracy of the model on the training and test datasets. The plt.title(), plt.ylabel(), and plt.xlabel() functions are used to set the title, y-axis label, and x-axis label of the plot, respectively.

The plt.legend() function is used to add a legend to the plot.

The plt.show() function is used to display the plot.

The loss key in the lstm_model_history.history dictionary contains the loss of the model on the training dataset, and the val_loss key contains the loss of the model on the test dataset.

The plt.plot(), plt.title(), plt.ylabel(), plt.xlabel(), and plt.legend() functions are used to create a plot of the loss of the model on the training and test datasets, similar to the plot of the accuracy of the model.

The plt.show() function is used to display the plot.

These plots can be used to evaluate the performance of the LSTM model. **The accuracy and loss** of the model on the training and test datasets can be used to determine whether the model is overfitting or underfitting the training data.

If the accuracy of the model on the test dataset is significantly lower than the accuracy of the model on the training dataset, then the model is overfitting the training data. Overfitting occurs when the model learns the specific features of the training data too well, and is unable to generalize to new data.

If the accuracy of the model on the test dataset is significantly higher than the accuracy of the model on the training dataset, then the model is underfitting the training data. Underfitting occurs when the model does not learn enough features from the training data, and is unable to make accurate predictions on new data.

The accuracy and loss of the model on the training and test datasets can be used to tune the hyperparameters of the LSTM model, such as the number of epochs to train the model for and the learning rate. The goal of tuning the hyperparameters is to improve the accuracy of the model on the test dataset without overfitting the training data.

## Model Optimization

The model can be optimized by using the hyperparameters of the embedding layer, adjusting the parameters by increasing/decreasing weights,vocab length, embedding dimensionality and the maximum length of the input text sequences. Additionally the model can be optimized by removing unnecessary layers/nodes which may overfit the model.

Here are the parameters of the embedding layer:

- vocab_length: The number of unique words in the vocabulary.
- embedding_dim: The dimensionality of the word embeddings.
- weights: A list of NumPy arrays that contain the pre-trained word embeddings.
- input_length: The maximum length of the input text sequences.

## Bibliography

1. Nabi, J. 2018 Machine learning — text Processing, Towards Data Science. Available at: https://towardsdatascience.com/machine-learning-text-processing-1d5a2d638958 (Accessed: July 1, 2023).
2. Olsson, F., 2009. A literature survey of active machine learning in the context of natural language processing
3. Nigam, V. 2019 Natural language processing: From basics to using RNN and LSTM, Analytics Vidhya. Available at: https://medium.com/analytics-vidhya/natural-language-processing-from-basics-to-using-rnn-and-lstm-ef6779e4ae66 (Accessed: September 20, 2023).

4.

5.