

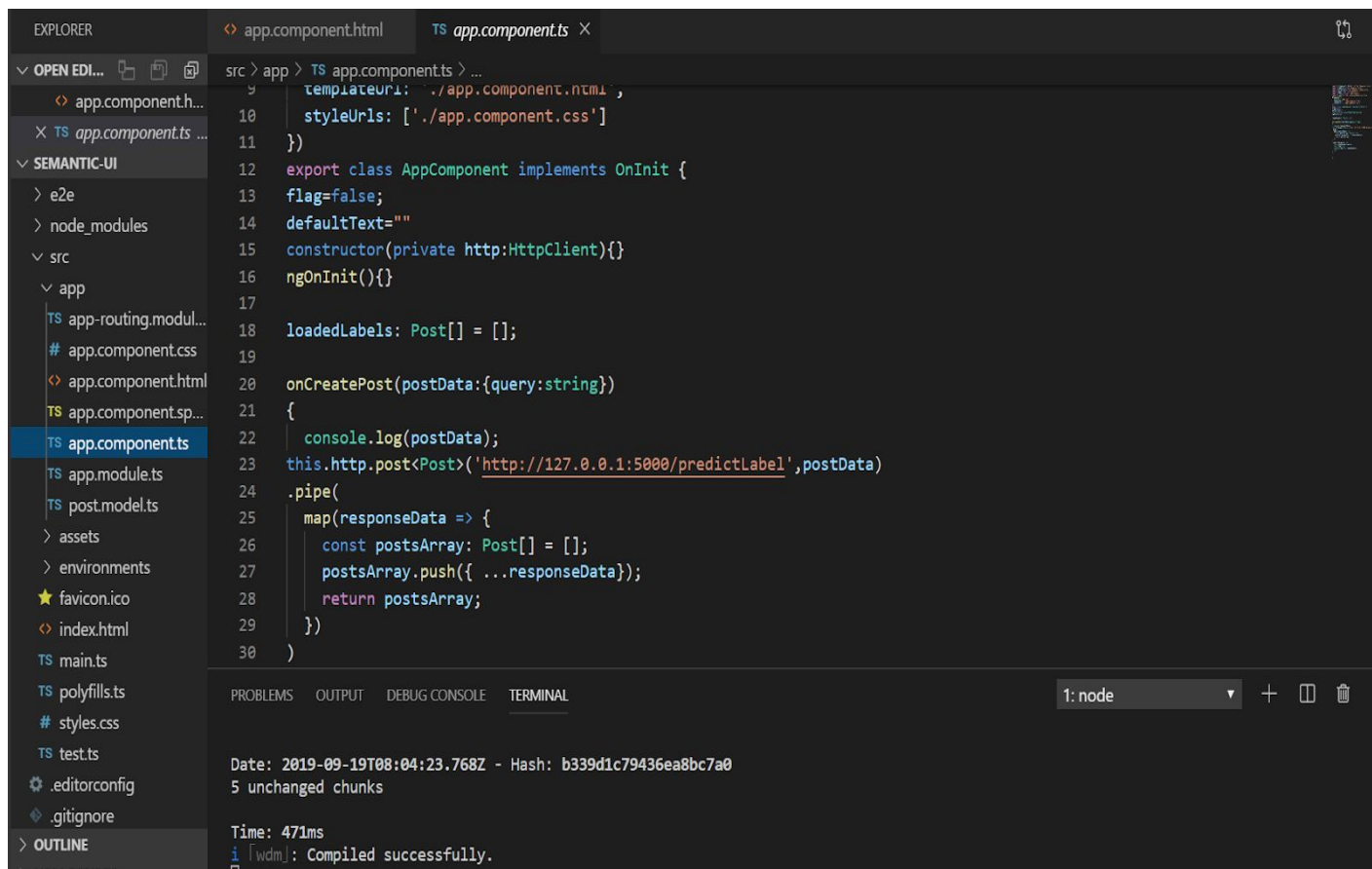
Semantics_Assignment

Description of files attached and how to run it:

1.GUI: I have used Angular 8 for building UI. To run the UI please open the “**semnatic-ui** folder” in visual studio or any other IDE which is preferable for you. The html and .ts files are present in the src folder. In app.component.ts I have specified the URL of the flask rest service which is running on portno:5000 for me. You can change it if its different for you. This is the URL:

['http://127.0.0.1:5000/predictLabel'](http://127.0.0.1:5000/predictLabel).

To run the UI you have to open the terminal and type **ng serve**. The angular app was running on <http://localhost:4200/> for me.



The screenshot displays the Visual Studio Code interface. On the left, the Explorer sidebar shows the project structure with the 'src/app' folder expanded, highlighting 'app.component.ts'. The main editor area shows the content of 'app.component.ts' with the following code:

```
9  src > app > TS app.component.ts > ...
10  templateUrl: './app.component.html',
11  styleUrls: ['./app.component.css']
12  })
13  export class AppComponent implements OnInit {
14    flag=false;
15    defaultText=""
16    constructor(private http:HttpClient){}
17    ngOnInit(){}
18
19    loadedLabels: Post[] = [];
20
21    onCreatePost(postData:{query:string})
22    {
23      console.log(postData);
24      this.http.post<Post>('http://127.0.0.1:5000/predictLabel',postData)
25      .pipe(
26        map(responseData => {
27          const postsArray: Post[] = [];
28          postsArray.push({ ...responseData});
29          return postsArray;
30        })
31      )
32    }
33  }
```

At the bottom, the TERMINAL panel shows the output of the 'ng serve' command:

```
1: node
Date: 2019-09-19T08:04:23.768Z - Hash: b339d1c79436ea8bc7a0
5 unchanged chunks
Time: 471ms
i [wdm]: Compiled successfully.
```

UI:

Semantic Analyser

Enter text

Apple already plans to buy back \$100 billion in shares, including \$16 billion worth last quarter. Icahn probably pounce

Get Label

computer-company

2.Rest Service and database :

I have used Flask library for building Rest services and Sqlite3 for the database.

In the “**backend** folder” I have stored all the files relating to rest service,models and database.

- **data.db:** This is the database file. It is created by running the create_table.py file.
It consists of one table “responses” which has two columns query and labels. The data type of both are string.
- **app.py:** contains the rest service. This python files loads **lstm_model.h5**. **tokenizer**,**word_to_vector_map**,**word2id**,**maxLen** pickle files to run the rest service.

You need to change the path depending on where you are saving the files.

- **maxLen.pickle:** This is the length of the input to be passed into the model. We need it to convert the input text into sequences of length=maxLen.
- **tokenizer.pickle:** It will help to convert the text which we are about to pass into model into sequences. It learned the vocabulary from the text we fed during training we have to store it
- **word2id.pickle:** This dictionary stored the mapping between words and their indices generated by the tokenizer.
- **word_to_vector_map.pickle:** This dictionary stores the mapping between words and their embedding vector generated from skipgram model.

3.Models:

There are two folders one for model1 and other for model2 :

Model1: Gave accuracy of 71% on the test data and 80% on the unseen data when I chose window size of 7 in the skip gram generator.

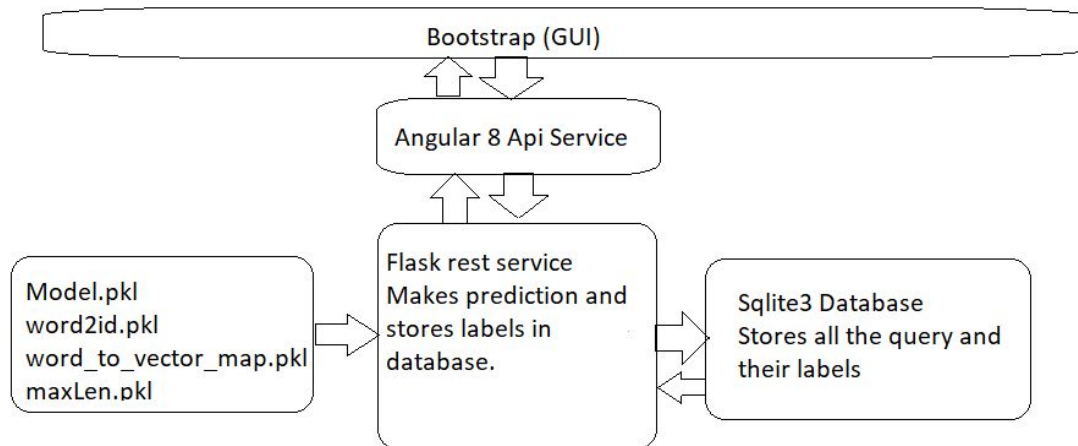
Model2: Gave accuracy of 82% on the test data and 69% on the unseen data when I chose window size of 8 in the skip gram generator.

- skipgram_model.h5: This is the model used to generate embedding vector for each word.
- skipgram_model.json
- lstm_model.h5: This is the classifier model which is used to train the input data corresponding to its labels.
- lstm_model.json

4.Deployment:

I have also attached the files required for deployment in the " **deployment files** folder". I used Heroku to deploy the rest service but it was

Architecture of GUI and API:



Input string is sent to backend rest service through angular app's POST HTTP service. Flask rest service preprocesses the input i.e clean the input string . Then first it check whether this particular string is already present in the database. If its there then it fetches the label from the database and sends it as a response to UI. Otherwise it first calls the method `sentence_to_indices` to convert the string to sequence of indices which is then fed into the model. The model then predicts the label for the string and sends it in response.

Reason for choosing skip gram model for this problem:

This problem is a text classification problem which classifies whether the sentence in the input belong to an apple fruit or Apple company. We need to represent words in form of vectors so for that I am using Skip gram algorithm which will generate embedding for each word in the sentence which can then be passed into the classifier model and train data corresponding to their labels.

- **What is Skip gram?**

Skipgram is one of the algorithm of Word2Vec. In this model, we take a **centre** word and a window of **context (neighbor)** words and we try to predict context words out to some window size for each centre word. So, our model is going to define a probability distribution i.e. probability of a word appearing in context given a centre word and we are going to choose our vector representations to maximize this probability.

Instead of defining the complete probability distribution over words, the model learns to differentiate between the correct training pairs retrieved from the corpus and the incorrect, randomly generated pairs. For each correct pair the model draws m negative ones — with m being a hyperparameter.

The new objective of the model is to maximise the probability of the correct samples coming from the corpus and minimise the corpus probability for the negative samples.

- **How Skip gram algorithm is helpful?**

It is helpful because it learns the context of the words in which it is used. So even

if your training set is small and it relates only a few words to a particular label, word embedding algorithm will be able to generalize and associate words in the test set to the same label even if those words don't even appear in the training set. This allows to build an accurate classifier mapping from sentences to labels, even using a small training set. This is a huge advantage of word embedding approaches over older popular TF/IDF (bag of words) approaches. Also the space used to store the word embeddings is small compared to TF/IDF.

Steps followed to build the skip gram and classifier model:

1. Built the **corpus vocabulary** by extracting unique words from the corpus and assigned unique identifiers to it. For this I used the tokenizer method available in keras.
2. Built the **skip gram generator** which will give pair of words and their relevance.
[(target,context),relevancy]. For this I used skip gram function of `keras.preprocessing.sequence`.
3. Built the **skip gram model** architecture which will take target word and context or random word pair as input. Each of which are passed to an embedding layer which is initialized with random weights. Once embeddings are obtained for target word and context word we pass it to the merge layer where dot product of these two vectors are calculated.
Then we finally pass this dot product to the Dense layer which predicts 0 or 1 depending on if the pair is contextually relevant or just random words. We match this with the actual relevance, compute the loss and backpropagate to update the embedding layer.

Summary of the model:

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	(None, 1)	0	
input_4 (InputLayer)	(None, 1)	0	
embedding (Embedding)	(None, 1, 100)	358600	input_3[0][0] input_4[0][0]
reshape_3 (Reshape)	(None, 100, 1)	0	embedding[0][0]
reshape_4 (Reshape)	(None, 100, 1)	0	embedding[1][0]
dot_2 (Dot)	(None, 1, 1)	0	reshape_3[0][0] reshape_4[0][0]
reshape_5 (Reshape)	(None, 1)	0	dot_2[0][0]
dense_1 (Dense)	(None, 1)	2	reshape_5[0][0]
Total params: 358,602			
Trainable params: 358,602			
Non-trainable params: 0			

I used **keras functional API** to build this model since here we have to

merge two layers and in sequential API it can't be done.

4. **Embedding vectors:** At last we get the word embedding by extracting the weights from the embedding layer. We use this embedding to generate a dictionary **word_to_vector_map** which maps each word to its embedding vector.
5. **Classifier model:** After getting the embeddings we will use them as features for the model which is developed to classify the sentence into one of the two categories (Apple-company or apple-fruit)

Here I have used **keras sequential API** to build the model.

Input layer defines the shape of the input (sentence indices) passed to the model. Then these sentence indices are passed to the embedding layer to get the embedding vectors corresponding to each index. These embeddings are passed to the LSTM model which has 128 hidden states. Dropout of 0.5 is applied on the output of previous layer. Then again it is passed to an LSTM with 128 hidden states and again Dropout of 0.5 is applied. Then the output is passed through the

Dense layer with softmax as the activation function. While compiling the model we are using binary_crossentropy as the loss function, adam optimizer for optimization and accuracy as the metrics.

Loss function:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i=0}^N (y * \log(\hat{y}_i) + (1 - y) * \log(1 - \hat{y}_i))$$

Summary of the model.

Layer (type)	Output Shape	Param #
=====	=====	=====
input_33 (InputLayer)	(None, 33)	0
embedding_18 (Embedding)	(None, 33, 100)	358600
lstm_17 (LSTM)	(None, 33, 128)	117248
dropout_17 (Dropout)	(None, 33, 128)	0
lstm_18 (LSTM)	(None, 128)	131584
dropout_18 (Dropout)	(None, 128)	0
dense_18 (Dense)	(None, 2)	258
activation_9 (Activation)	(None, 2)	0
=====	=====	=====
Total params: 607,690		
Trainable params: 249,090		
Non-trainable params: 358,600		

6. **Train Test split:** I have split the data such that 80 percent is the training data and 20 percent is the test. Also I have converted the data into sequences of indices of fixed length=maxLen. And label is either 0 or 1.
7. **Results:** Finally after running the model on the train data got an accuracy of 71% on the test data and 80% on the unseen data when I chose window size of 7 in the skip gram generator.

Improvements that can be done:

- Online available pretrained GloVe Vectors can be used to increase the accuracy since it has embedding for a large no of words.
- Size of training data is also an important factor when building your own embedding vector model.