Chatbot for Intent Recognition

Report by

Adina Dingankar

Rehapriadarsini Manikandasamy

Venkata Gangadhar Naveen Palaka

The George Washington University

Author Note

This report was prepared for DATS 6312, taught by Professor Amir Jafari

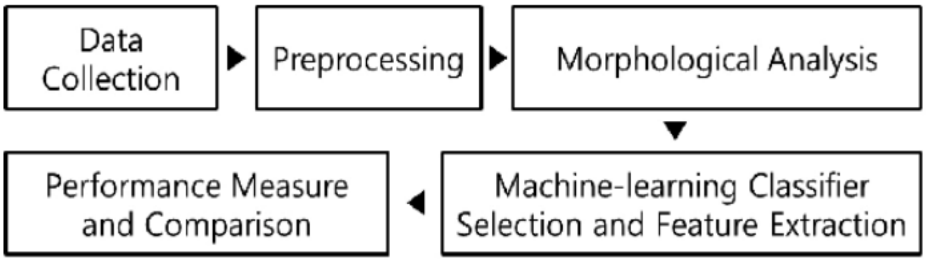
TABLE OF CONTENTS

INTRODUCTION

Intent recognition is sometimes called as intent classification is the task of taking a written or spoken input, and classifying it based on what the user wants to achieve. Intent recognition forms an essential component of chatbots and finds use in sales conversions, customer support, and many other areas. Intent recognition is a form of natural language processing (NLP), a subfield of artificial intelligence. NLP is concerned with computers processing and analyzing natural language, i.e., any language that has developed naturally, rather than artificially, such as with computer coding languages. Intent recognition works through the process of providing examples of text alongside their intents to a machine learning (ML) model. This is known as using training data to train a model.

Chatbots use natural language processing (NLP) to understand the user's intent which means recognizing user's aim in starting any conversation. Intent recognition is a critical feature in chatbot architecture that determines if a chatbot will succeed at fulfilling the user's needs in sales, marketing or customer service. The quantity of the chatbot's training data is key to maintaining a good conversation with the users. However, the data quality determines the bot's ability to detect the right intent and generate the correct response. Natural language processing (NLP) allows the chatbot to understand the user's message, and machine learning classification algorithms to classify this message based on the training data, and deliver the correct response. The chatbots' intent detection component helps identify what general task or goal the user is trying to accomplish to handle the conversation with different strategies. Once the goal is known, the bot must manage a dialogue to achieve that goal, ensuring that follow-up questions are handled correctly. The intent detector uses a text classifier to classify the input sentence into one of several classes.

One such major application of intent recognition is Google's Dialog Flow which is been used as Malaysian Airlines, Dominos etc. in order to understand their customers and improve their sales.


Data Collection
Preprocessing
Morphological Analysis
Machine-learning Classifier Selection and Feature Extraction
Performance Measure and Comparison

DESCRIPTION OF THE DATASET

The motivation for the Intent Recognition dataset is been drawn from the Natural Language Understanding Benchmark incorporating Few-Shot-Detection. Few-Shot-Intent-Detection is a repository designed for few-shot intent detection with/without Out-of-Scope (OOS) intents. It includes popular challenging intent detection datasets and baselines. Here is the basic understanding of OOD-OOS and ID-OOS intents.

OOD-OOS: i.e., out-of-domain OOS. General out-of-scope queries which are not supported by the dialog systems, also called out-of-domain OOS. For instance, requesting an online NBA/TV show service in a banking system.

ID-OOS: i.e., in-domain OOS. Out-of-scope queries which are more related to the in-scope intents, which makes the intent detection task more challenging. For instance, requesting a banking service that is not supported by the banking system.

The scope of our dataset has been inherited from 'CLINC150' intent dataset and since our purpose was to built a chatbot for intent recognition we have developed a custom-built dataset.

Features of the dataset are described as:

1. Tags: Intents
2. Patterns: User Input
3. Responses: Bot Responses

The above-described features are viewed as:

{"intents":

[{"tag": "greeting",

"patterns": ["Hi", "How are you", "Is anyone there?", "Hello", "Good day"],

"responses": ["Hello, thanks for visiting", "Good to see you again", "Hi there, how can I help?"],

]}}

DESCRIPTION OF ALGORITHMS

LSTM:

Long short-term memory networks, usually called LSTM – are a special kind of RNN. They were introduced to avoid the long-term dependency problem. In regular RNN, the problem frequently occurs when connecting previous information to new information. If RNN could do this, they'd be very useful. This problem is called long-term dependency.

How does LSTM work?

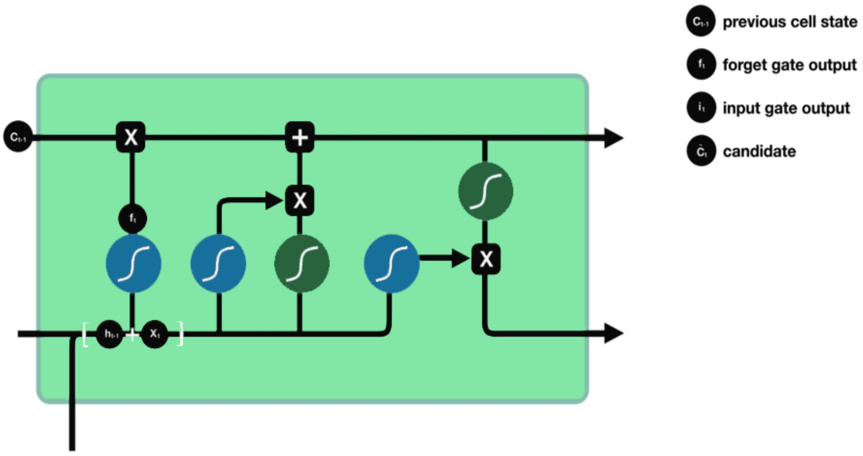
LSTM has 3 main gates.

1. FORGET Gate
2. INPUT Gate
3. OUTPUT Gate

Let's have a quick look at them one by one.

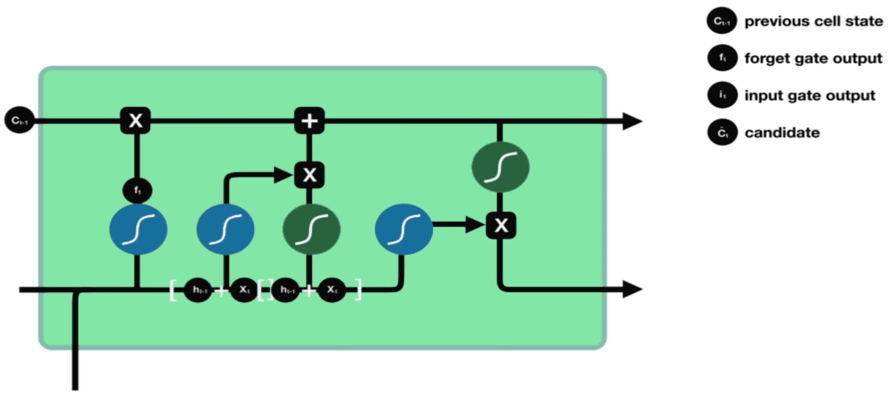
FORGET Gate

This gate is responsible for deciding which information is kept for calculating the cell state and which is not relevant and can be discarded. The ht-1 is the information from the previous hidden state (previous cell) and xt is the information from the current cell. These are the 2 inputs given to the Forget gate. They are passed through a sigmoid function and the ones tending towards 0 are discarded, and others are passed further to calculate the cell state.



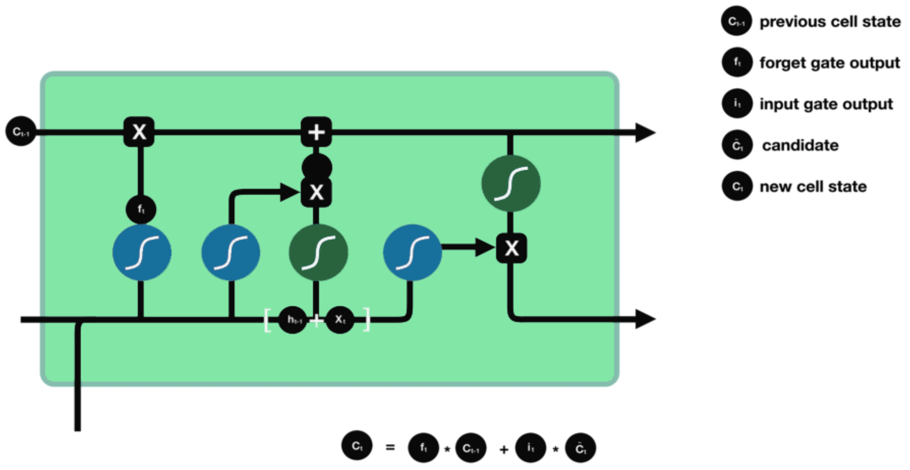
INPUT Gate

Input Gate updates the cell state and decides which information is important and which is not. As forget gate helps to discard the information, the input gate helps to find out important information and store certain data in the memory that relevant. ht-1 and xt are the inputs that are both passed through sigmoid and tanh functions respectively. tanh function regulates the network and reduces bias.



Cell State

All the information gained is then used to calculate the new cell state. The cell state is first multiplied with the output of the forget gate. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then a pointwise addition with the output from the input gate updates the cell state to new values that the neural network finds relevant.



OUTPUT Gate

The last gate which is the Output gate decides what the next hidden state should be. ht-1 and xt are passed to a sigmoid function. Then the newly modified cell state is passed through the tanh function and is multiplied with the sigmoid output to decide what information the hidden state should carry.

Why LSTM?

Traditional neural networks suffer from short-term memory. Also, a big drawback is the vanishing gradient problem. (While backpropagating the gradient becomes so small that it tends to 0 and such a neuron is of no use in further processing.) LSTMs efficiently improve performance by memorizing the relevant information that is important and finding the pattern.

LSTM for Multi-Class Text Classification

There are many classic classification algorithms like Decision trees, RFR, SVM, that can fairly do a good job, then why to use LSTM for classification?

One good reason to use LSTM is that it is effective in memorizing important information. If we look and other non-neural network classification techniques they are trained on multiple word as separate inputs that are just word having no actual meaning as a sentence, and while predicting the class it will give the output according to statistics and not according to meaning. That means, every single word is classified into one of the categories.

This is not the same in LSTM. In LSTM we can use a multiple word string to find out the class to which it belongs. This is very helpful while working with Natural language processing. If we use appropriate layers of embedding and encoding in LSTM, the model will be able to find out the actual meaning in input string and will give the most accurate output class.

GLOVE:

Glove stands for Global Vectors for word representation. It is an unsupervised learning algorithm developed by researchers at Stanford University aiming to generate word embeddings by aggregating global word co-occurrence matrices from a given corpus. The basic idea behind the Glove word embedding is to derive the relationship between the words from statistics. Unlike the occurrence matrix, the co-occurrence matrix tells you how often a particular word pair occurs together. Each value in the co-occurrence matrix represents a pair of words occurring together.

This is the idea behind the Glove pre-trained word embeddings, and it is expressed as;

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

BERT:

BERT stands for Bidirectional Encoder Representations from Transformers.
Google introduced the transformer architecture in the paper “Attention is All you need”. Transformer uses a self-attention mechanism, which is suitable for language understanding. The transformer has an encoder-decoder architecture. They are composed of modules that contain feed-forward and attention layers.

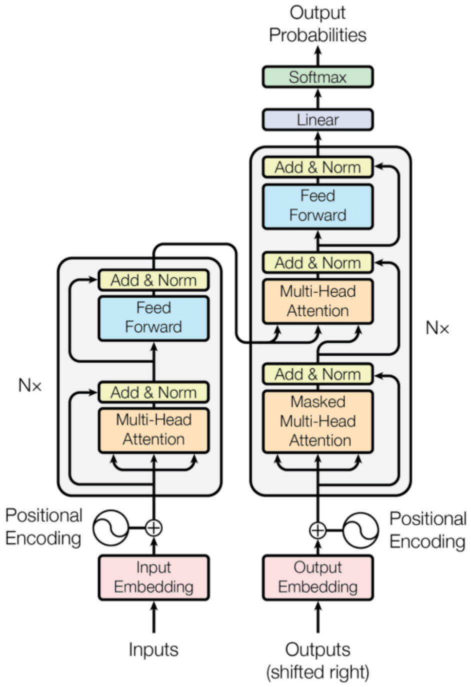


Figure 1: The Transformer - model architecture.

What is the need for BERT?

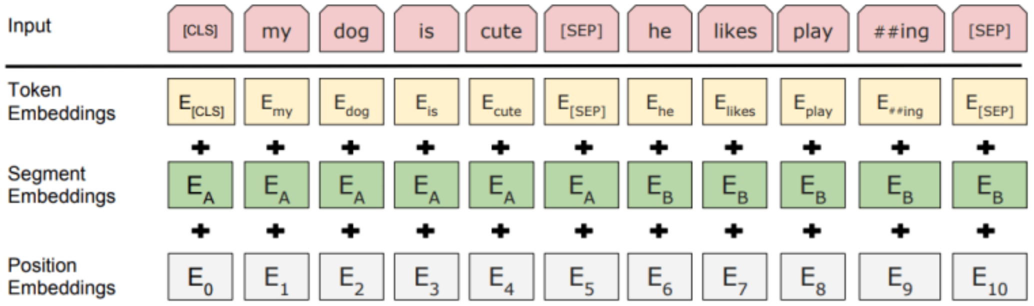
Generally, language models read the input sequence in one direction: either left to right or right to left. This kind of one-directional training works well when the aim is to predict/generate the next word. But in order to have a deeper sense of language context, BERT uses bidirectional training. Sometimes, it's also referred to as "non-directional". So, it takes both the previous and next tokens into account simultaneously. BERT applies the bidirectional training of Transformer to language modeling, learns the text representations. BERT is just an encoder. It does not have a decoder. The encoder is responsible for reading text input and processing. The decoder is responsible for producing a prediction for the task.

Architecture of BERT

BERT is a multi-layered encoder. In that paper, two models were introduced, BERT base and BERT large. The BERT large has double the layers compared to the base model. By layers, we indicate transformer blocks. BERT-base was trained on 4 cloud-based TPUs for 4 days and BERT-large was trained on 16 TPUs for 4 days.

BERT base – 12 layers, 12 attention heads, and 110 million parameters.

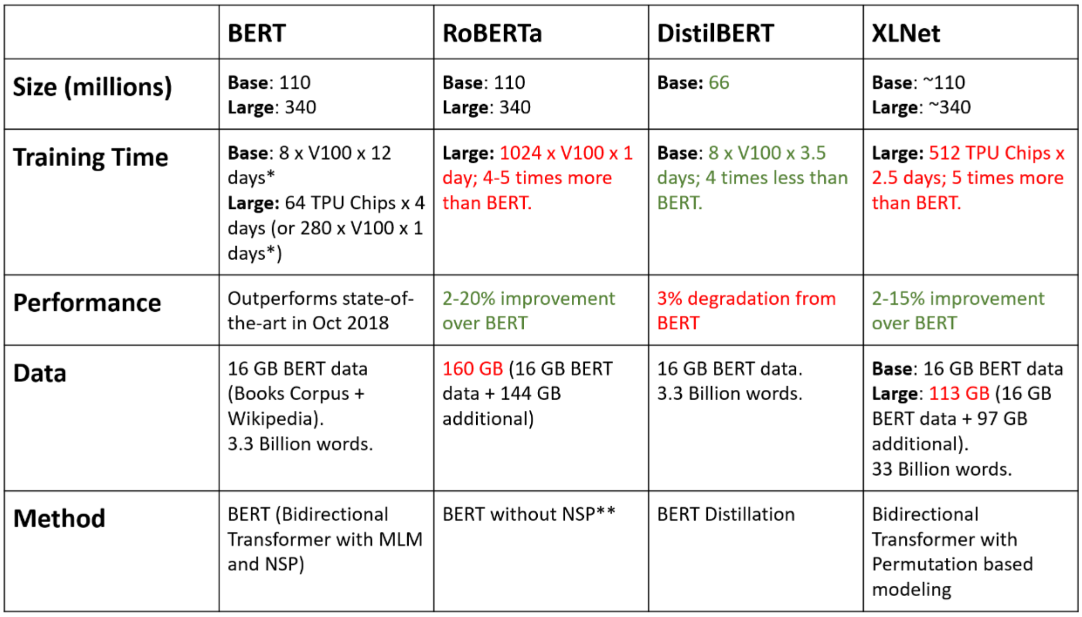
BERT Large – 24 layers, 16 attention heads and, 340 million parameters.


Input
[CLS]
my
dog
is
cute
[SEP]
he
likes
play
##ing
[SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} E_{cute} $E_{[SEP]}$ E_{he} E_{likes} E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10}

BERT was trained with the masked language modeling (MLM) and next sentence prediction (NSP) objectives. It is efficient at predicting masked tokens and at NLU in general, but is not optimal for text generation.

Distil BERT:

As Transfer Learning from large-scale pre-trained models becomes more prevalent in Natural Language Processing (NLP), operating these large models in on-the-edge and/or under constrained computational training or inference budgets remains challenging. Hence a method has been proposed to pre-train a smaller general-purpose language representation model, called Distil BERT, which can then be fine-tuned with good performances on a wide range of tasks like its larger counterparts. While most prior work investigated the use of distillation for building task-specific models, we leverage knowledge distillation during the pre-training phase and show that it is possible to reduce the size of a BERT model by 40%, while retaining 97% of its language understanding capabilities and being 60% faster. To leverage the inductive biases learned by larger models during pre-training, we introduce a triple loss combining language modeling, distillation and cosine-distance losses. Distill BERT is smaller, faster and lighter model is cheaper to pre-train and demonstrates its capabilities for on-device computations in a proof-of-concept experiment and a comparative on-device study.



	BERT	**RoBERTa**	**DistilBERT**	**XLNet**
Size (millions)	**Base**: 110 **Large**: 340	**Base**: 110 **Large**: 340	**Base:** 66	**Base**: ~110 **Large**: ~340
Training Time	**Base**: 8 x V100 x 12 days* **Large:** 64 TPU Chips x 4 days (or 280 x V100 x 1 days*)	**Large:** 1024 x V100 x 1 day; 4-5 times more than BERT.	**Base:** 8 x V100 x 3.5 days; 4 times less than BERT.	**Large:** 512 TPU Chips x 2.5 days; 5 times more than BERT.
Performance	Outperforms state-of-the-art in Oct 2018	2-20% improvement over BERT	3% degradation from BERT	2-15% improvement over BERT
Data	16 GB BERT data (Books Corpus + Wikipedia). 3.3 Billion words.	160 GB (16 GB BERT data + 144 GB additional)	16 GB BERT data. 3.3 Billion words.	**Base**: 16 GB BERT data **Large**: 113 GB (16 GB BERT data + 97 GB additional). 33 Billion words.
Method	BERT (Bidirectional Transformer with MLM and NSP)	BERT without NSP**	BERT Distillation	Bidirectional Transformer with Permutation based modeling

EXPERIMENTAL SETUP

AWS_SETUP

AWS instance setup:

1. Log in to AWS account and launch console.
2. Launch virtual machine
3. Select AMI if already exists
4. GPU – g3.4xlarge
5. Generate a key pair and download the key in .pem version
6. Launch the instance

GCP instance setup:

1. Launch Google cloud console
2. Launch compute engine
3. Create a project and click create intents
4. Select the following options in the instance creation tab:
5. Zone – US central-a-zone
6. Series- N1
7. Machine type – n1-standard-8(30GB)
8. GPU – nvidia-teslaT4/P4
9. Boot disk – ubuntu, version – 20.04, size – 300
10. Add the public key generated using Puttygen software

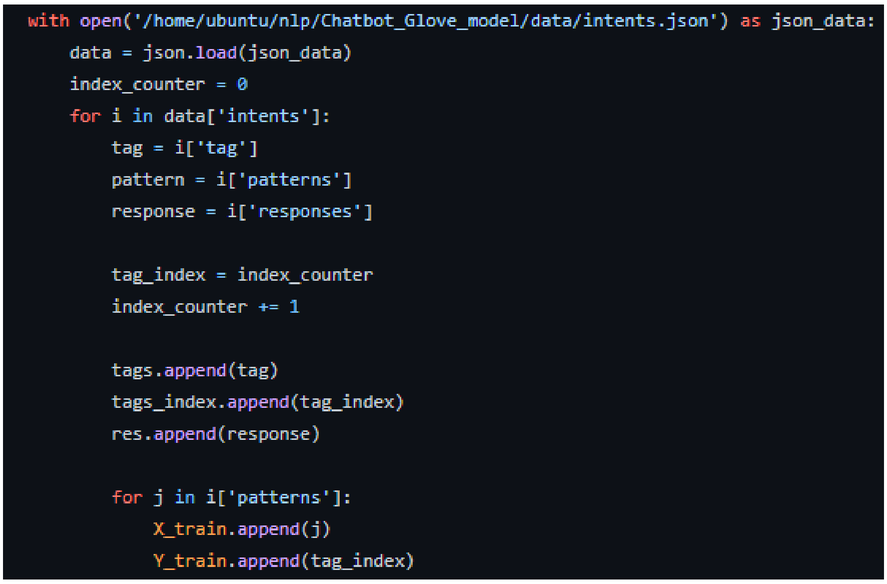
For windows,

1. Create a SSH session in mobaxterm
2. Host: External IP from any one of the instances
3. Name: ubuntu
4. Add the private key downloaded from the instances
5. Click ok.
6. Create a folder in the cloud
7. PyCharm integration:
8. Create a project
9. Tools -> Deployment -> Configuration -> Add SFTP
10. SSH configuration: Host- IP address from instances, name: ubuntu, authorize using the private key pair.
11. Test the connection
12. In mappings, map the remote local path and the cloud folder’s path
13. Deployment -> Configuration -> Automatic upload
14. Setup the Interpreter
15. Python interpreter -> SSH interpreter -> Existing server configuration -> Choose the cloud -> click ok.

DATASET

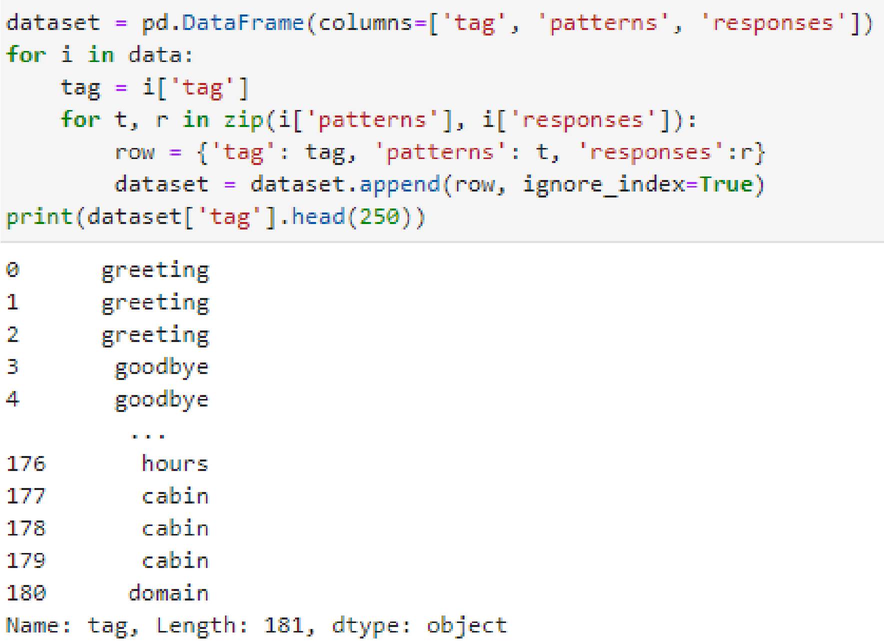
Since the dataset is custom built for our use, we don't have test data we have implemented .json file and converted into the csv file. The actual size of the train data is 868 rows and 3 attributes we are using the same data and mark it as test data to test the performance of the model. Hence before feeding the data to the models.

For LSTM: we are using the train data and dividing into X_train and Y_train where X_train contains the patterns and Y_train contains the tag_index along with the associated responses.



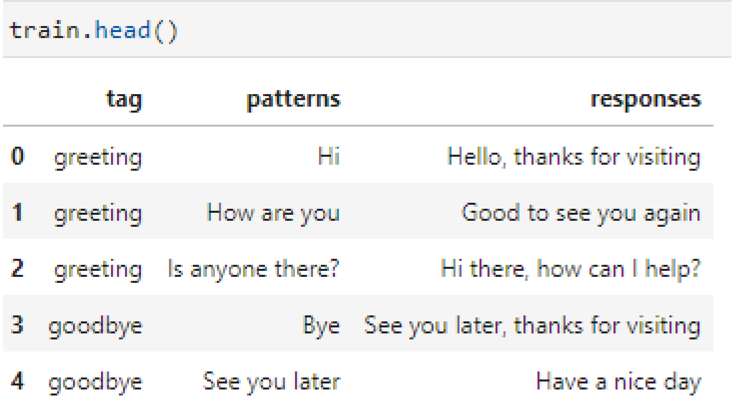
```
with open('/home/ubuntu/nlp/Chatbot_Glove_model/data/intents.json') as json_data:
    data = json.load(json_data)
    index_counter = 0
    for i in data['intents']:
        tag = i['tag']
        pattern = i['patterns']
        response = i['responses']

        tag_index = index_counter
        index_counter += 1

        tags.append(tag)
        tags_index.append(tag_index)
        res.append(response)

        for j in i['patterns']:
            X_train.append(j)
            Y_train.append(tag_index)
```

For BERT: we are replicating the train data to make a test data and have feed into the model where in the text column is "patterns" and labels "tag".



```
dataset = pd.DataFrame(columns=['tag', 'patterns', 'responses'])
for i in data:
    tag = i['tag']
    for t, r in zip(i['patterns'], i['responses']):
        row = {'tag': tag, 'patterns': t, 'responses':r}
        dataset = dataset.append(row, ignore_index=True)
print(dataset['tag'].head(250))
```

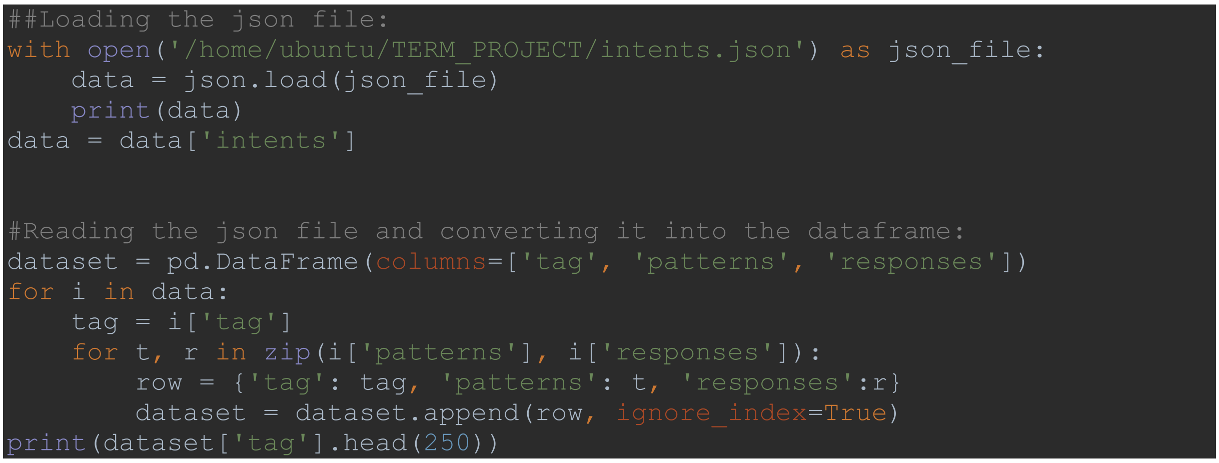
```
0      greeting
1      greeting
2      greeting
3       goodbye
4       goodbye
         ...
176       hours
177       cabin
178       cabin
179       cabin
180      domain
Name: tag, Length: 181, dtype: object
```



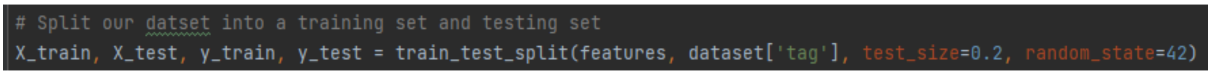
```
train.head()
```

	tag	patterns	responses
0	greeting	Hi	Hello, thanks for visiting
1	greeting	How are you	Good to see you again
2	greeting	Is anyone there?	Hi there, how can I help?
3	goodbye	Bye	See you later, thanks for visiting
4	goodbye	See you later	Have a nice day

For DISTIL BERT: we are using the entire train dataset and then by using train -test-split dividing the data into train and test with the test size of 0.2.



```
##Loading the json file:
with open('/home/ubuntu/TERM_PROJECT/intents.json') as json_file:
    data = json.load(json_file)
    print(data)
data = data['intents']


#Reading the json file and converting it into the dataframe:
dataset = pd.DataFrame(columns=['tag', 'patterns', 'responses'])
for i in data:
    tag = i['tag']
    for t, r in zip(i['patterns'], i['responses']):
        row = {'tag': tag, 'patterns': t, 'responses':r}
        dataset = dataset.append(row, ignore_index=True)
print(dataset['tag'].head(250))
```



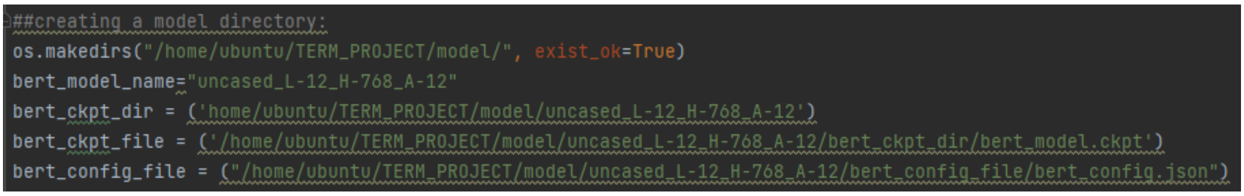
```
# Split our datset into a training set and testing set
X_train, X_test, y_train, y_test = train_test_split(features, dataset['tag'], test_size=0.2, random_state=42)
```

MODEL IMPLEMENTATION

Explain how you will implement the model in the chosen framework and how you will judge the performance.

BERT

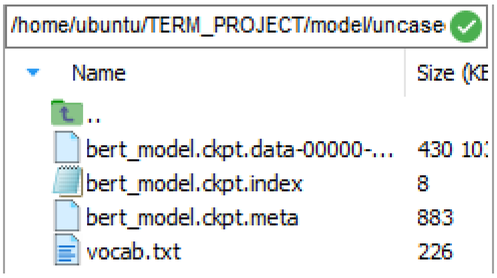
For BERT implementation the packages which are required are Bert-for-tf2, BertModelLayer and tqdm of the specific versions mentioned in the requirement.txt. After installing all the necessary libraries we have mounted our project path to the model which contains the uncased-bert-pretrained-model which is further sub-divided into the bert-config.json , bert-data file , bert.meta and bert.index. The bert-config.json consist of the configurations of the model on the top of which we will be embedding our keras layers.



```
##creating a model directory:
os.makedirs("/home/ubuntu/TERM_PROJECT/model/", exist_ok=True)
bert_model_name="uncased_L-12_H-768_A-12"
bert_ckpt_dir = ('home/ubuntu/TERM_PROJECT/model/uncased_L-12_H-768_A-12')
bert_ckpt_file = ('/home/ubuntu/TERM_PROJECT/model/uncased_L-12_H-768_A-12/bert_ckpt_dir/bert_model.ckpt')
bert_config_file = ("/home/ubuntu/TERM_PROJECT/model/uncased_L-12_H-768_A-12/bert_config_file/bert_config.json")
```

Bert_config.json looks like :

```
{
  "attention_probs_dropout_prob": 0.1,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "max_position_embeddings": 512,
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "type_vocab_size": 2,
  "vocab_size": 30522
}
```



/home/ubuntu/TERM_PROJECT/model/uncase

Name	Size (KE
..	
bert_model.ckpt.data-00000-...	430 10:
bert_model.ckpt.index	8
bert_model.ckpt.meta	883
vocab.txt	226

In the model building phase, we have first created a class named as Data Preparation wherein we are assigning the text and label columns which in our case is pattern and tag , then after we are calling the train and test data as it is which is then tokenized using Full tokenizer which points to the vocab.txt file which came inbuilt as a part of the uncased-bert-pretrained-model the vocab.txt contains 30k words which is used for text embedding before sending the data in the tokenized form to the model to understand.

```
##input data prepp
class DataPreparation:
  text_column = "patterns"
  label_column = "tag"

  def __init__(self, train, test, tokenizer: FullTokenizer, classes,
max_seq_len=192):
    self.tokenizer = tokenizer
    self.max_seq_len = 0
    self.classes = classes

    ((self.train_x, self.train_y), (self.test_x, self.test_y)) =
map(self.prepare_data, [train, test])

    print("max seq_len", self.max_seq_len)
    self.max_seq_len = min(self.max_seq_len, max_seq_len)
    self.train_x, self.test_x = map(self.data_padding, [self.train_x,
self.test_x])

  def prepare_data(self, df):
    x, y = [], []

    for _, row in tqdm(df.iterrows()):
      text, label = row[DataPreparation.text_column],
row[DataPreparation.label_column]
```

```
      tokens = self.tokenizer.tokenize(text)
      tokens = ["[CLS]"] + tokens + ["[SEP]"]
      token_ids = self.tokenizer.convert_tokens_to_ids(tokens)
      self.max_seq_len = max(self.max_seq_len, len(token_ids))
      x.append(token_ids)
      y.append(self.classes.index(label))

    return np.array(x), np.array(y)

  def data_padding(self, ids):
    x = []
    for input_ids in ids:
      input_ids = input_ids[:min(len(input_ids), self.max_seq_len - 2)]
      input_ids = input_ids + [0] * (self.max_seq_len - len(input_ids))
      x.append(np.array(input_ids))
    return np.array(x)
tokenizer =
FullTokenizer(vocab_file=('/home/ubuntu/TERM_PROJECT/model/uncased_L-12_H-
768_A-12/bert_ckpt_dir/vocab.txt'))
```

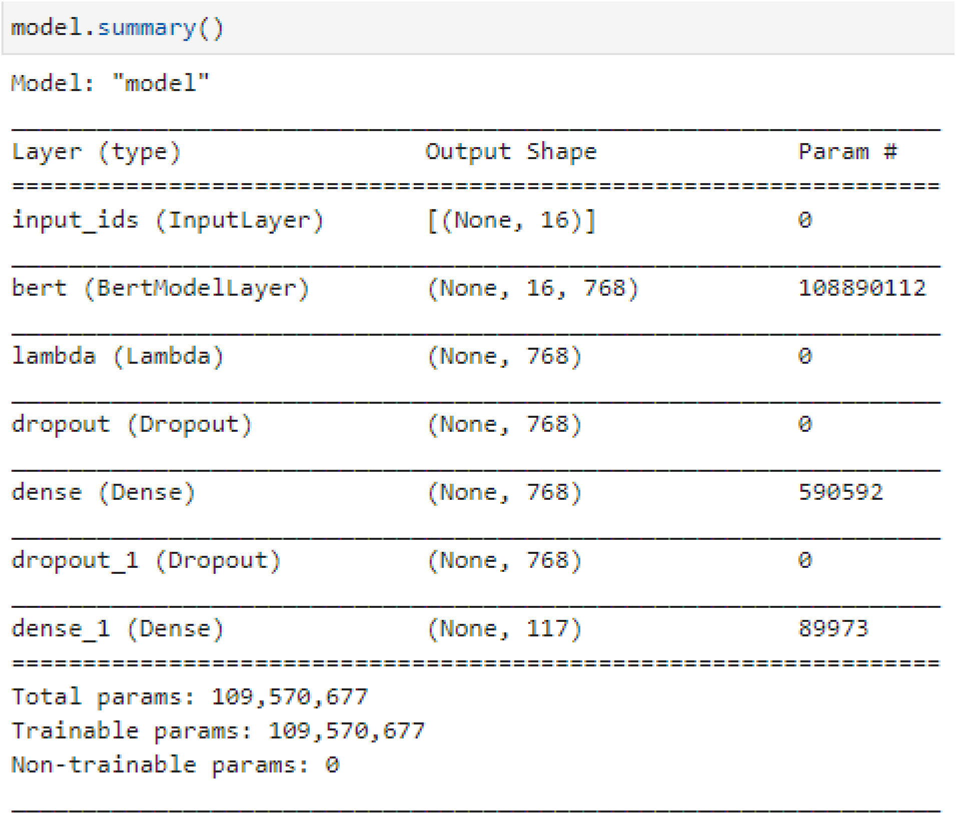
The second stage is to create a model where in we are calling the pretrained bert model and we will be embedding the keras model with two dense layers and we have use two dropout layers with value of 0.5 using the activation function of tanh and softmax as in comparison to other activations functions which we hyper parameterized it on our model this two gave a better calculation of the output.

```
def model_defination(max_seq_len, bert_ckpt_file):
  with tf.io.gfile.GFile(bert_config_file, "r") as reader:
    bc = StockBertConfig.from_json_string(reader.read())
    bert_params = map_stock_config_to_params(bc)
    bert_params.adapter_size = None
    bert = BertModelLayer.from_params(bert_params, name="bert")

  input_ids = keras.layers.Input(shape=(max_seq_len,), dtype='int32',
name="input_ids")
  bert_output = bert(input_ids)

  print("bert shape", bert_output.shape)

  cls_out = keras.layers.Lambda(lambda seq: seq[:, 0, :])(bert_output)
  cls_out = keras.layers.Dropout(0.5)(cls_out)
  logits = keras.layers.Dense(units=768, activation="tanh")(cls_out)
  logits = keras.layers.Dropout(0.5)(logits)
  logits = keras.layers.Dense(units=len(classes),
activation="softmax")(logits)

  model = keras.Model(inputs=input_ids, outputs=logits)
  model.build(input_shape=(None, max_seq_len))

  load_stock_weights(bert, bert_ckpt_file)

  return model
```

The max length of sequence which is been used is 192 -128 before fitting the model.

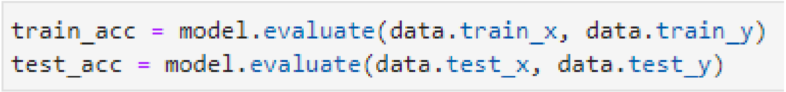
```
classes = train.tag.unique().tolist()
print(classes)
data = DataPreparation(train, test, tokenizer, classes, max_seq_len=128)
model = model_defination(data.max_seq_len, bert_ckpt_file)
```

The different performance metric which we have used for estimation are model.summary and model.evalutate

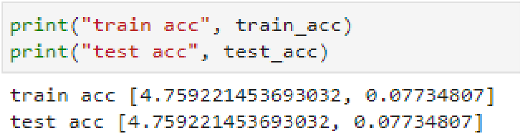


```
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_ids (InputLayer)       [(None, 16)]              0         
_________________________________________________________________
bert (BertModelLayer)        (None, 16, 768)           108890112 
_________________________________________________________________
lambda (Lambda)              (None, 768)               0         
_________________________________________________________________
dropout (Dropout)            (None, 768)               0         
_________________________________________________________________
dense (Dense)                (None, 768)               590592    
_________________________________________________________________
dropout_1 (Dropout)          (None, 768)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 117)               89973     
=================================================================
Total params: 109,570,677
Trainable params: 109,570,677
Non-trainable params: 0
_________________________________________________________________
```



```
train_acc = model.evaluate(data.train_x, data.train_y)
test_acc = model.evaluate(data.test_x, data.test_y)
```



```
print("train acc", train_acc)
print("test acc", test_acc)
```

```
train acc [4.759221453693032, 0.07734807]
test acc [4.759221453693032, 0.07734807]
```

Since there are dependency challenges with respect to the tensorflow version if one requires to run the code on AWS EC2 instance along with the installation of requirement.txt the tensorflow version should be set to 2.0.0 for running the above-mentioned code.

For running the code on GCP the following mentioned steps must be followed:

Go to AI platforms on console.cloud.google.com.

Create a notebook, set the OS -Ubuntu_20.0.4LTS

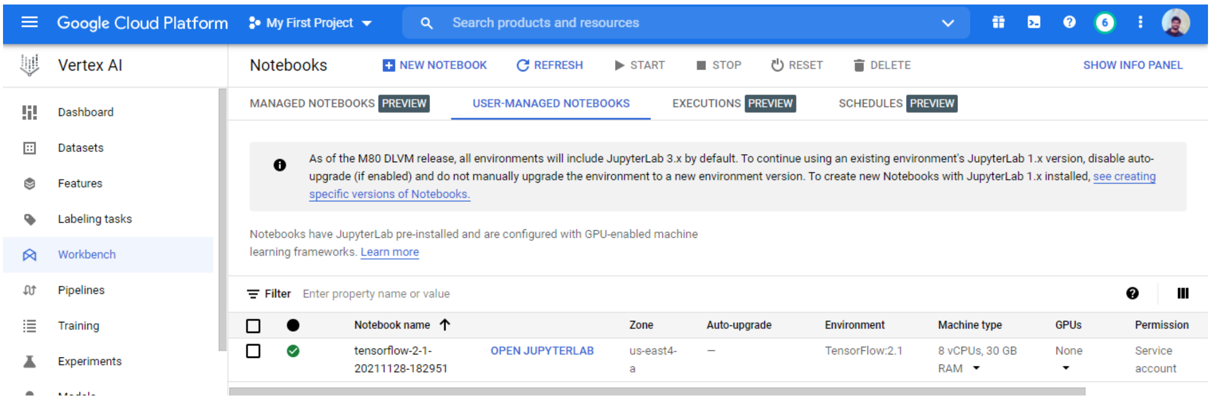
Select Pytorch1.9 setup from the drop down menu (NLTK, GPU,Tensorflow)

Core for the CPU: 8 core cpu.

GPU: NVIDIA tesla P4/T4

Exit

Since AI platforms are notebooks integrated with Jupyter notebooks you can easily set the configurations as per the dependency to run the above code.



As of the M80 DLVM release, all environments will include JupyterLab 3.x by default. To continue using an existing environment's JupyterLab 1.x version, disable auto-upgrade (if enabled) and do not manually upgrade the environment to a new environment version. To create new Notebooks with JupyterLab 1.x installed, see creating specific versions of Notebooks.

Notebooks have JupyterLab pre-installed and are configured with GPU-enabled machine learning frameworks. Learn more

□	●	Notebook name ↑		Zone	Auto-upgrade	Environment	Machine type	GPUs	Permission
□	✓	tensorflow-2-1-20211128-182951	OPEN JUPYTERLAB	us-east4-a	–	TensorFlow:2.1	8 vCPUs, 30 GB RAM	None	Service account

Distil BERT

For Distil BERT implementation we are making use of distilled-bert-uncased model

```
#Defining the pretrained distil bert model
model_class, tokenizer_class, pretrained_weights = (ppb.DistilBertModel,
ppb.DistilBertTokenizer, 'distilbert-base-uncased')
# Load pretrained model/tokenizer
tokenizer = tokenizer_class.from_pretrained(pretrained_weights)
model = model_class.from_pretrained(pretrained_weights)
```

Once the data is been tokenized by using the pattern column of the data, we are using attention mask and building a tensor out of the padded input and sending it to the distil bert model

```
tokenized = dataset['patterns'].apply((lambda x: tokenizer.encode(x,
add_special_tokens=True)))
max_len = 0
for i in tokenized.values:
    if len(i) > max_len:
        max_len = len(i)

padded = np.array([i + [0]*(max_len-len(i)) for i in tokenized.values])
np.array(padded).shape
attention_mask = np.where(padded != 0, 1, 0)
attention_mask.shape
#create an input tensor out of the padded token matrix, and send that to
DistilBERT
input_ids = torch.tensor(np.array(padded))
with torch.no_grad():
  # last_hidden_states holds the outputs of DistilBERT.
  # It is a tuple with the shape (number of examples, max number of tokens
in the sequence, number of hidden units in the DistilBERT model).
    last_hidden_states = model(input_ids)
 # Slice the output for the first position for all the sequences, take all
hidden unit outputs
features = last_hidden_states[0][:,0,:].numpy()
```

We have used RandomizedSearchCV where the estimator is Logistic Regression to find the best params and scores for our model since we are using Distil BERT to embed the text and modelling is performed by Logistic Regression

```
# search for the best value of the C parameter, which determines
regularization strength.
parameters = {'C': np.linspace(0.0001, 100, 20)}
```

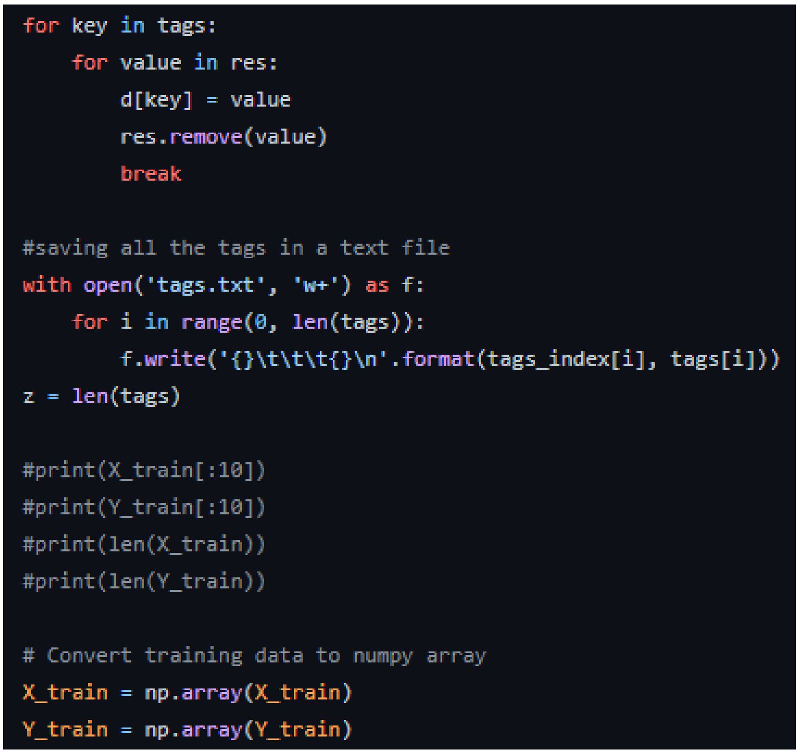
```
lr_finder = RandomizedSearchCV(estimator = LogisticRegression() , scoring = 
'accuracy',
                               param_distributions = parameters,
                               cv = KFold(n_splits=5, shuffle=True, 
random_state = seed),
                               verbose=50, random_state=seed, n_jobs = -1)
lr_finder.fit(X_train, y_train)

print('best parameters: ', lr_finder.best_params_)
print('best scores: ', lr_finder.best_score_)
```

The models performance is measured by using the classification report from sklearn.metric package.

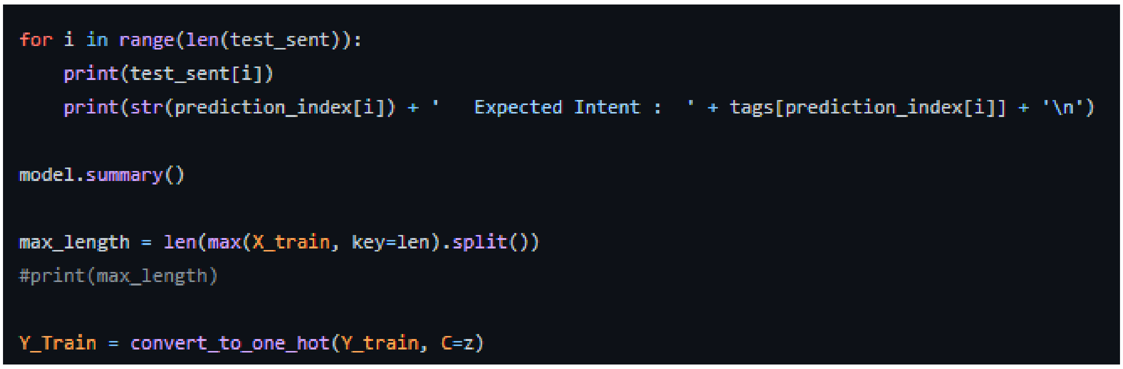
```
lr_cv = lr_finder.best_estimator_.fit(X_train, y_train)
y_lr_pred = lr_cv.predict(X_test)
print(metrics.classification_report(y_test, y_lr_pred))
```

LSTM

For LSTM once the dataset is been built then we further create a data dictionary which is used to store the tags, we are using pretrained LSTM model and Glove model of 6B.50d and 100d .txt as well which is used to tokenize the data and we are performing one hot encoding on the y_train which contains the tag_index and reponses.

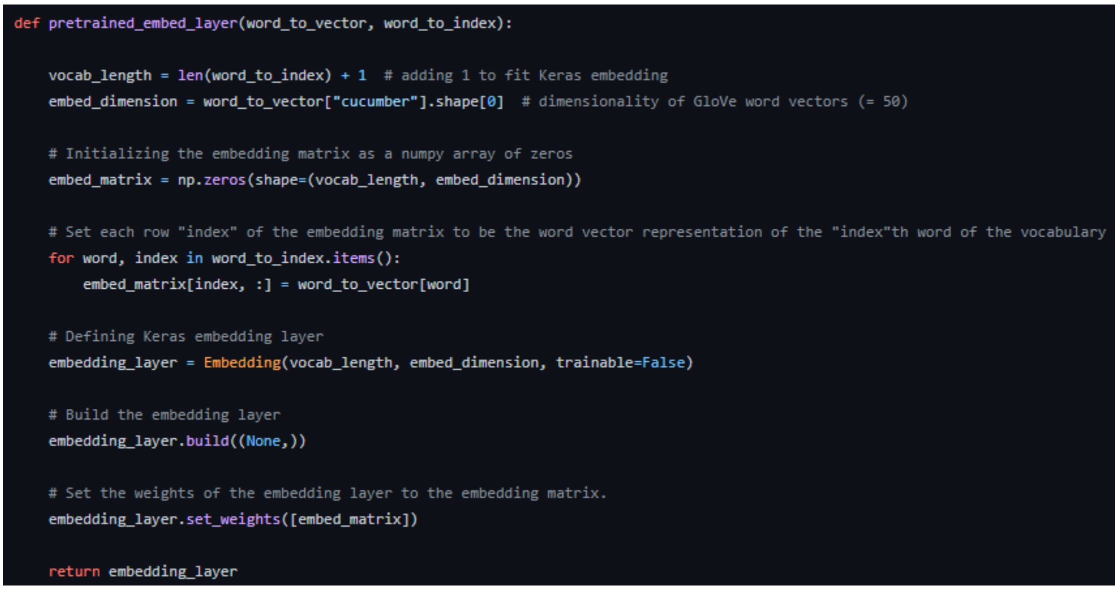


```
for key in tags:
    for value in res:
        d[key] = value
        res.remove(value)
        break

#saving all the tags in a text file
with open('tags.txt', 'w+') as f:
    for i in range(0, len(tags)):
        f.write('{}\t\t\t{}\n'.format(tags_index[i], tags[i]))
z = len(tags)

#print(X_train[:10])
#print(Y_train[:10])
#print(len(X_train))
#print(len(Y_train))

# Convert training data to numpy array
X_train = np.array(X_train)
Y_train = np.array(Y_train)
```

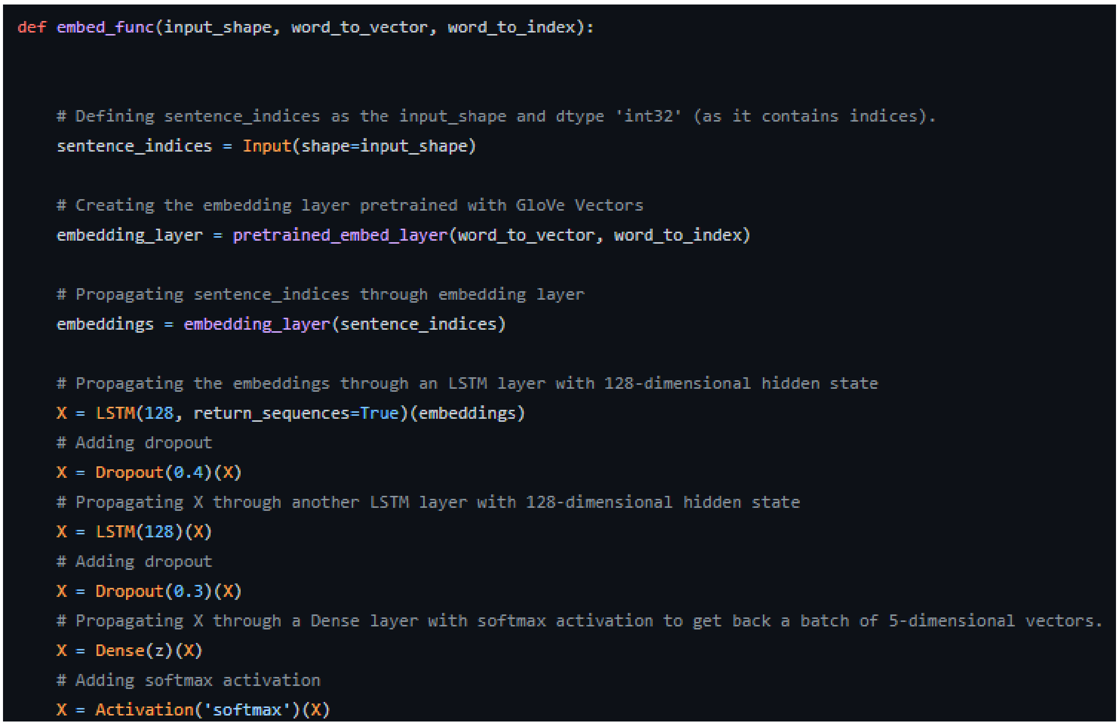


```
for i in range(len(test_sent)):
    print(test_sent[i])
    print(str(prediction_index[i]) + '   Expected Intent :  ' + tags[prediction_index[i]] + '\n')

model.summary()

max_length = len(max(X_train, key=len).split())
#print(max_length)

Y_Train = convert_to_one_hot(Y_train, C=z)
```

Now we have performed LSTM model building by embedding 1 dense layer with softmax activation function.

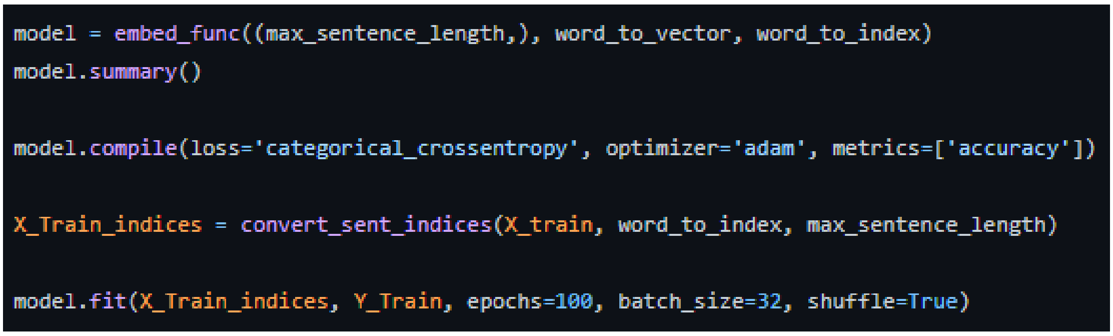


```
def pretrained_embed_layer(word_to_vector, word_to_index):

    vocab_length = len(word_to_index) + 1  # adding 1 to fit Keras embedding
    embed_dimension = word_to_vector["cucumber"].shape[0]  # dimensionality of GloVe word vectors (= 50)

    # Initializing the embedding matrix as a numpy array of zeros
    embed_matrix = np.zeros(shape=(vocab_length, embed_dimension))

    # Set each row "index" of the embedding matrix to be the word vector representation of the "index"th word of the vocabulary
    for word, index in word_to_index.items():
        embed_matrix[index, :] = word_to_vector[word]

    # Defining Keras embedding layer
    embedding_layer = Embedding(vocab_length, embed_dimension, trainable=False)

    # Build the embedding layer
    embedding_layer.build((None,))

    # Set the weights of the embedding layer to the embedding matrix.
    embedding_layer.set_weights([embed_matrix])

    return embedding_layer
```

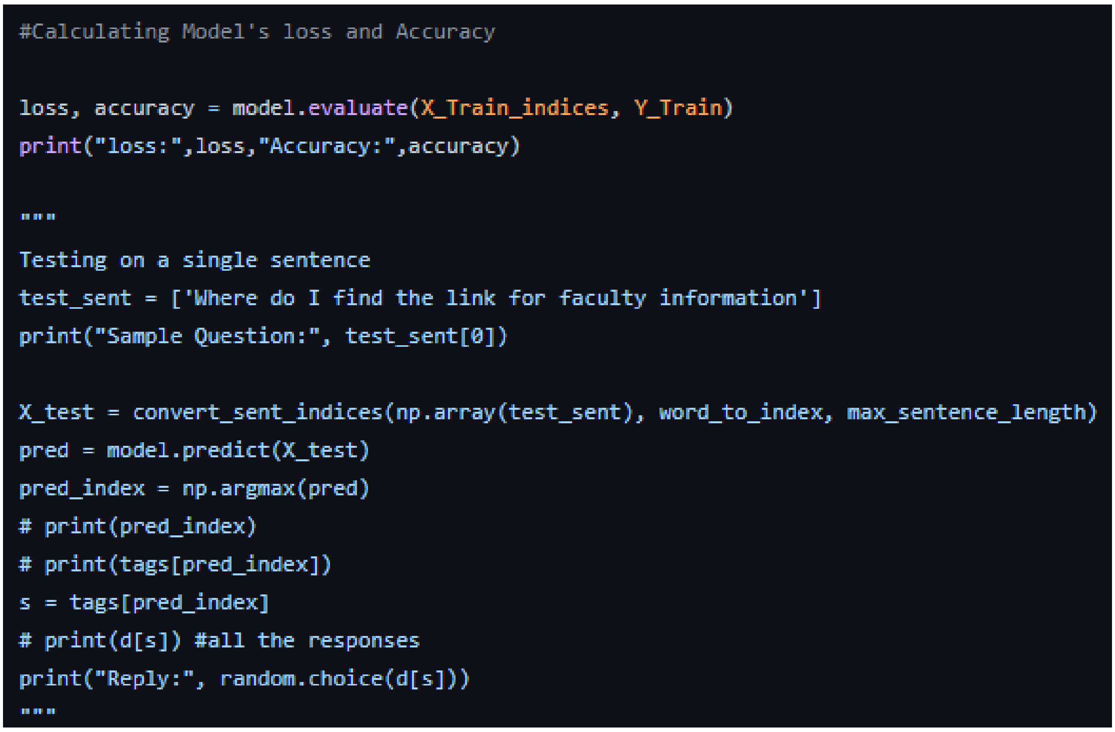


```
def embed_func(input_shape, word_to_vector, word_to_index):


    # Defining sentence_indices as the input_shape and dtype 'int32' (as it contains indices).
    sentence_indices = Input(shape=input_shape)

    # Creating the embedding layer pretrained with GloVe Vectors
    embedding_layer = pretrained_embed_layer(word_to_vector, word_to_index)

    # Propagating sentence_indices through embedding layer
    embeddings = embedding_layer(sentence_indices)

    # Propagating the embeddings through an LSTM layer with 128-dimensional hidden state
    X = LSTM(128, return_sequences=True)(embeddings)
    # Adding dropout
    X = Dropout(0.4)(X)
    # Propagating X through another LSTM layer with 128-dimensional hidden state
    X = LSTM(128)(X)
    # Adding dropout
    X = Dropout(0.3)(X)
    # Propagating X through a Dense layer with softmax activation to get back a batch of 5-dimensional vectors.
    X = Dense(z)(X)
    # Adding softmax activation
    X = Activation('softmax')(X)
```

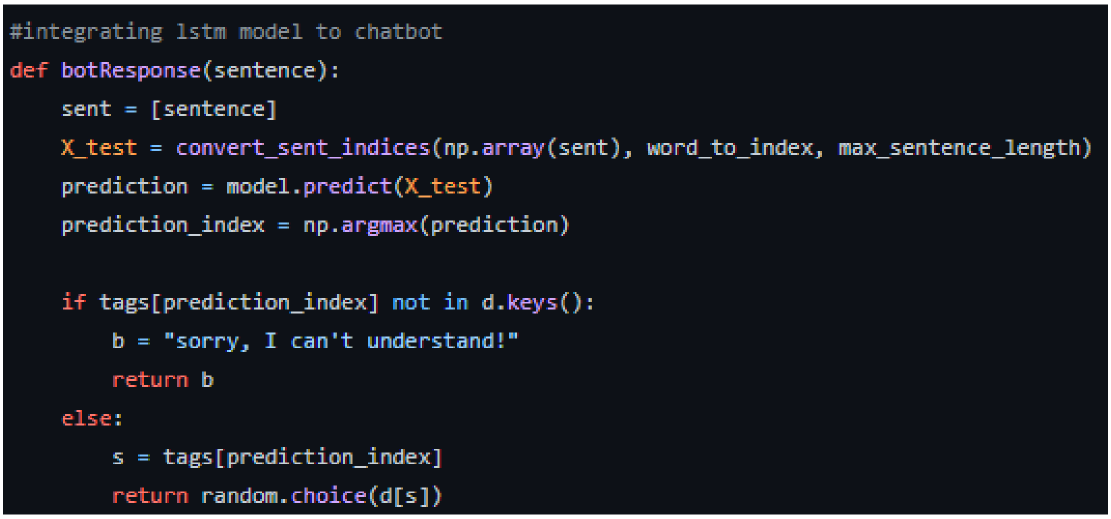
The performance metrics used for the model are:



```
model = embed_func((max_sentence_length,), word_to_vector, word_to_index)
model.summary()

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

X_Train_indices = convert_sent_indices(X_train, word_to_index, max_sentence_length)

model.fit(X_Train_indices, Y_Train, epochs=100, batch_size=32, shuffle=True)
```



```
#Calculating Model's loss and Accuracy

loss, accuracy = model.evaluate(X_Train_indices, Y_Train)
print("loss:",loss,"Accuracy:",accuracy)

"""
Testing on a single sentence
test_sent = ['Where do I find the link for faculty information']
print("Sample Question:", test_sent[0])

X_test = convert_sent_indices(np.array(test_sent), word_to_index, max_sentence_length)
pred = model.predict(X_test)
pred_index = np.argmax(pred)
# print(pred_index)
# print(tags[pred_index])
s = tags[pred_index]
# print(d[s]) #all the responses
print("Reply:", random.choice(d[s]))
"""
```



```
#integrating lstm model to chatbot
def botResponse(sentence):
    sent = [sentence]
    X_test = convert_sent_indices(np.array(sent), word_to_index, max_sentence_length)
    prediction = model.predict(X_test)
    prediction_index = np.argmax(prediction)

    if tags[prediction_index] not in d.keys():
        b = "sorry, I can't understand!"
        return b
    else:
        s = tags[prediction_index]
        return random.choice(d[s])
```

Since the accuracy score of BERT, DISTIL BERT was relative lesser than LSTM hence we have used LSTM for our chatbot implementation.

CHATBOT

The packages required for the chatbot implementation are tkinter and python.Steps to set up tkinter on AWS are as follows:

1.First we need to start the AWS instance and connect to pycharm

2.install XQuartz terminal,

3.Connect to the terminal using the followiing command - ssh -X -i file.pem ubuntu@publickey

4.run the main.py first then run the chat_bot.py file

The layout of the UI is divided into different functions we have created a class named Chat Application we firstly have init function which is use to initialize the Chat Application and run function shows the window up and running , Then after we have created a function setup_main_window which forms the major functionalities of the chat window like head label is used to give a label to the head of the chat window, tiny label is used on the user end to display the message for eg “ hello , how are you ?” tiny divider is used to have a break between the chat window and header. Scrollbar is used to scroll through the messages in the chat application, message entry box is where the user inputs the message and send button.

The two functions on_enter_pressed and insert message is integrated with the model code where in the model.py file will be executed and post running that the chatbot.py file will be run which will pop up the window and the user enters the message from the message entry box on the submission of which it then goes to the LSTM model through the insert message function , the bot will read the txt message inputted recognize the intent and on the basis of the intent recognized it will give the associated response back to the user.

FLOW-CHART

Dataset:

The dataset is named as Intents.json is a custom-built dataset by following the rubrics of Few-Shot-Detection for OOS(Out-Of-Scope) intents. The dataset has 180 intents further divided into patterns which are the inputs taken from the user and responses which will be delivered by the bot.

All the Few-Shot-Detection datasets have a practical approach of two columns which includes the text and the intents as the purpose of the dataset is to help the model predict the intents of the given sentence. Since we are developing a chatbot it was quite necessary to train the model both on the inputs and responses which will help the bot to first read the sentence, recognize/detect the intent and then further give the response and this methodology for development led us to create a custom-built not domain intents. The size of the dataset has 868 rows and 3 columns named as tag, pattern and responses.

BERT:

We are referring to Hugging face Transformers for BERT development and we are using pretrained uncased Bert model. The parameters which are mentioned in the config.json file for the model has 12 hidden layers with the vocab size of 30522.On top of that we have embedded keras model with two dense layers using tanh and SoftMax function. Since BERT does has the possibility of overfitting the data hence, we have set two dropout layers to 0.5 and the elements of the hidden layers have been reduced from 768-181.

Distil BERT:

We aren’t setting any hyperparameters for distill Bert as we are simply performing the text embedding and modelling using Logistic Regression, Since the distill Bert has the degradation in the performance by 3% in comparison to BERT hence their dint occur any chances of overfitting to take the preventive measures.

LSTM:

The Hyperparameters which are using for LSTM is we have created two dropouts of 0.3 and 0.4 to eliminate the chances of overfitting with one dense layer having max length of 128 and activation function we have set to SoftMax since tanh and relu dint gave a better accuracy in comparison to SoftMax.

Activation Functions:

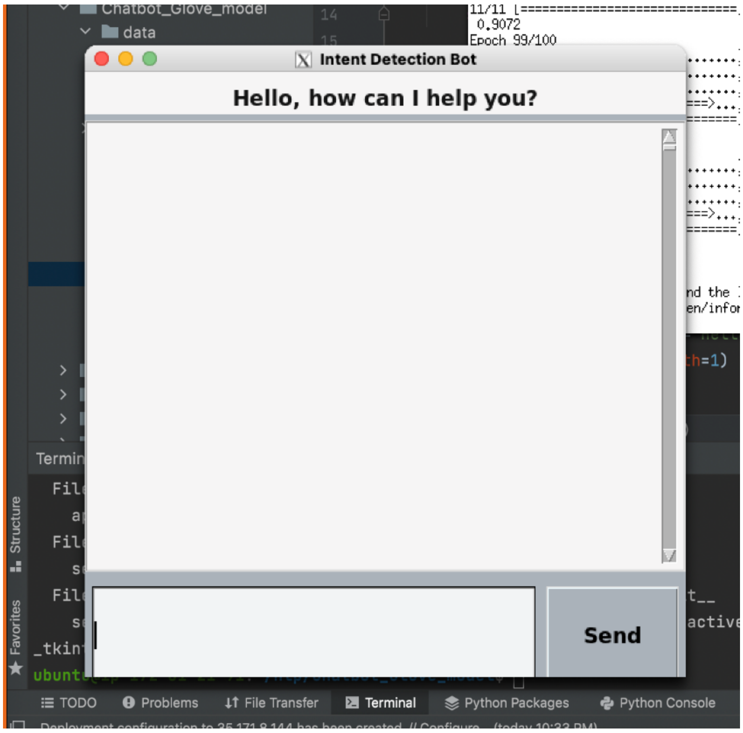
SoftMax is a very interesting activation function because it not only maps our output to a [0,1] range but also maps each output in such a way that the total sum is 1. The output of SoftMax is therefore a probability distribution.

SoftMax is used for multi-classification in logistic regression model (multivariate) whereas Sigmoid is used for binary classification in logistic regression model.

RESULTS

CHATBOT

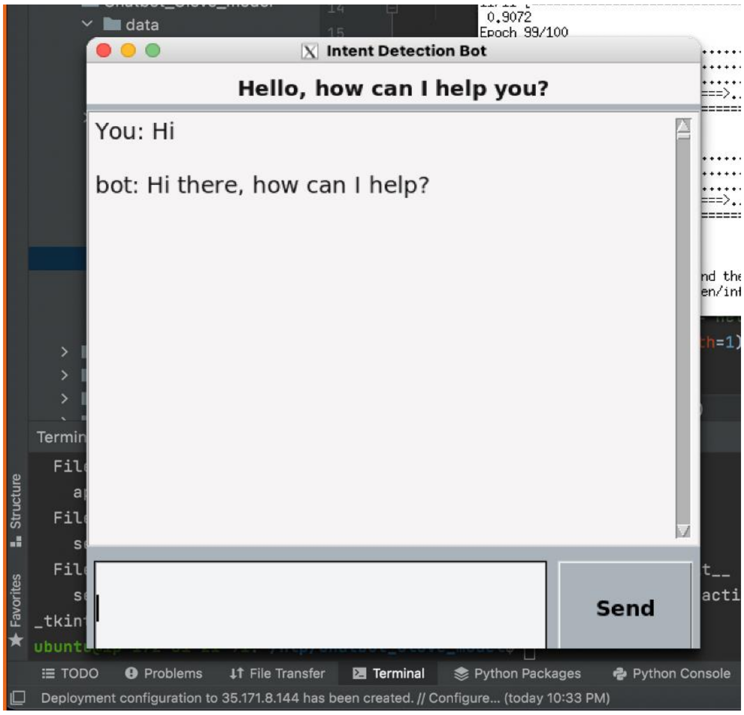
Initial UI Window: This is the initial window of the Chat Bot which gets executed after running the main.py file and chat_bot.py file.



```
11/11 [==============================
 0.9072
Epoch 99/100
```

Fig1.Initial UI Window

Step2: In this step we will be asking questions to chatbot which we have added as a pattern in the json file to see how the bot responds us on the basis of the intent detected.



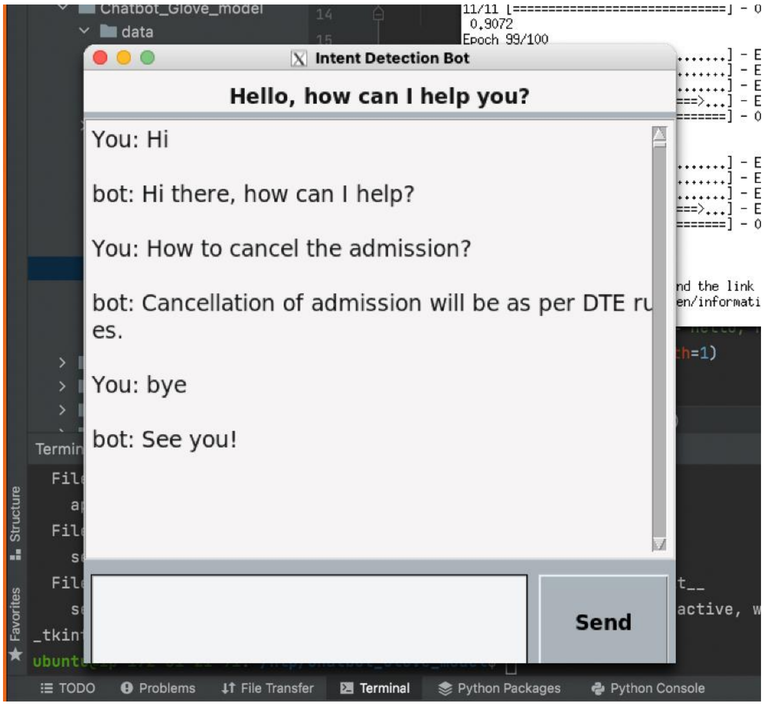
You: Hi

bot: Hi there, how can I help?

Deployment configuration to 35.171.8.144 has been created. // Configure... (today 10:33 PM)

Fig2.Step2 of the UI

Step3: In this window we are clearly able to see that the bot did responds us on the basis of the intent so our question to the bot was how to cancel the admission where in the intent for this question was admission cancellation on the basis of the responses feed to the model the bot gave us the exactly expected Ans to our question.



You: Hi

bot: Hi there, how can I help?

You: How to cancel the admission?

bot: Cancellation of admission will be as per DTE ru
es.

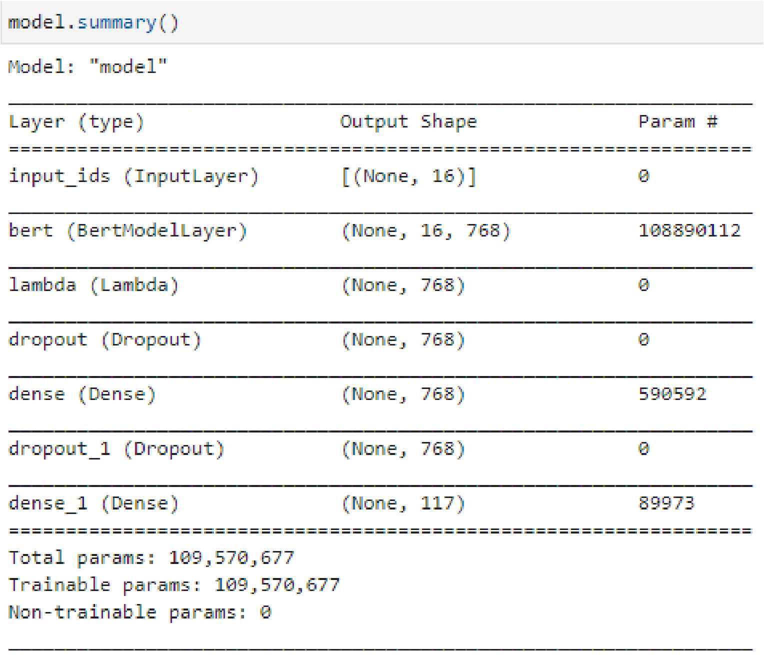
You: bye

bot: See you!

Fig3.Step3 of the UI

BERT

Below mentioned is the model summary

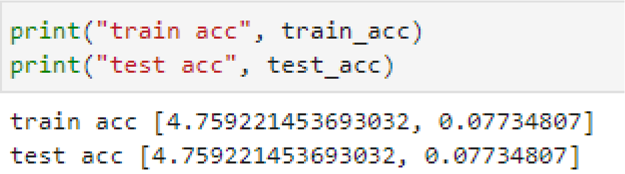


```
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_ids (InputLayer)       [(None, 16)]              0
_________________________________________________________________
bert (BertModelLayer)        (None, 16, 768)           108890112
_________________________________________________________________
lambda (Lambda)              (None, 768)               0
_________________________________________________________________
dropout (Dropout)            (None, 768)               0
_________________________________________________________________
dense (Dense)                (None, 768)               590592
_________________________________________________________________
dropout_1 (Dropout)          (None, 768)               0
_________________________________________________________________
dense_1 (Dense)              (None, 117)               89973
=================================================================
Total params: 109,570,677
Trainable params: 109,570,677
Non-trainable params: 0
_________________________________________________________________
```

Fig4. Model summary of BERT Transformer

Performance Metric for BERT Transformer



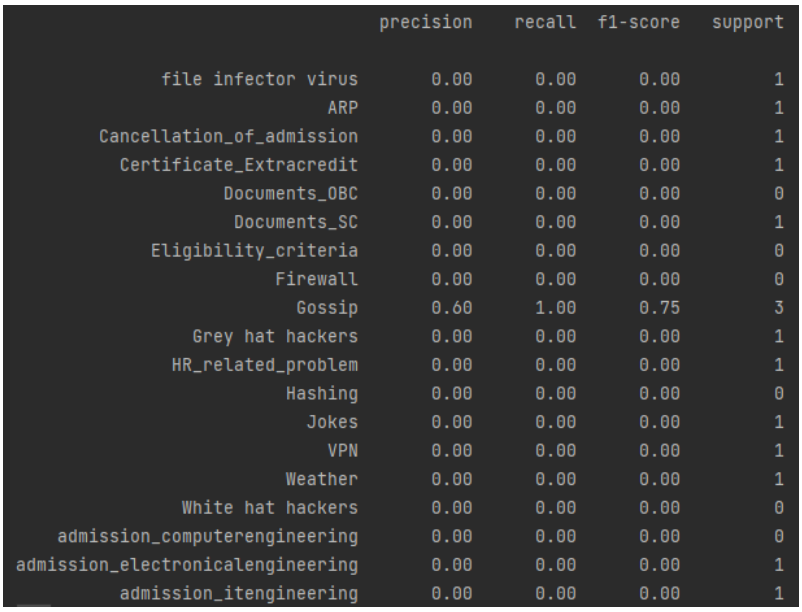
```
print("train acc", train_acc)
print("test acc", test_acc)
```

```
train acc [4.759221453693032, 0.07734807]
test acc [4.759221453693032, 0.07734807]
```

Fig5. Performance Metric for BERT Transformer

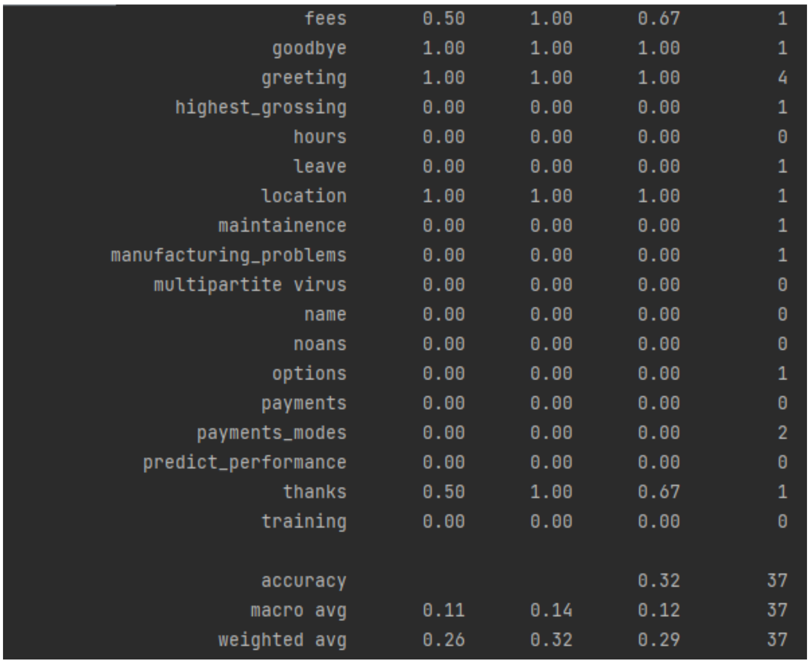
DISTIL BERT

Performance metrics for distill bert



```
                                   precision    recall  f1-score   support

                file infector virus       0.00      0.00      0.00         1
                                ARP       0.00      0.00      0.00         1
            Cancellation_of_admission       0.00      0.00      0.00         1
             Certificate_Extracredit       0.00      0.00      0.00         1
                       Documents_OBC       0.00      0.00      0.00         0
                        Documents_SC       0.00      0.00      0.00         1
                 Eligibility_criteria       0.00      0.00      0.00         0
                            Firewall       0.00      0.00      0.00         0
                              Gossip       0.60      1.00      0.75         3
                    Grey hat hackers       0.00      0.00      0.00         1
                  HR_related_problem       0.00      0.00      0.00         1
                             Hashing       0.00      0.00      0.00         0
                               Jokes       0.00      0.00      0.00         1
                                 VPN       0.00      0.00      0.00         1
                             Weather       0.00      0.00      0.00         1
                   White hat hackers       0.00      0.00      0.00         0
        admission_computerengineering       0.00      0.00      0.00         0
     admission_electronicalengineering       0.00      0.00      0.00         1
             admission_itengineering       0.00      0.00      0.00         1
```

Fig6. Performance Metric of Distil BERT

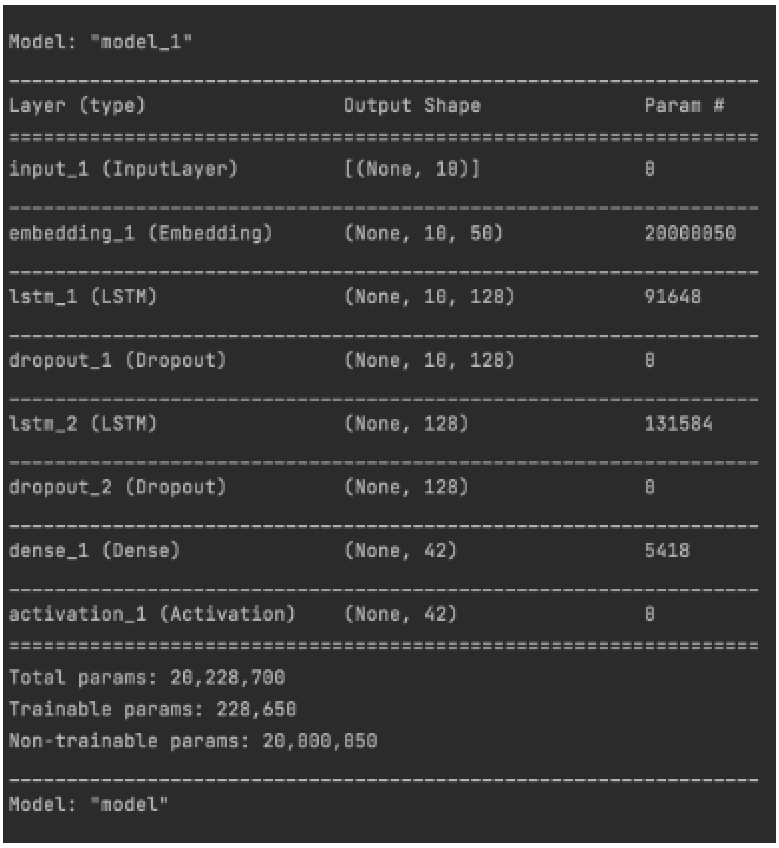


```
                             fees       0.50      1.00      0.67         1
                          goodbye       1.00      1.00      1.00         1
                         greeting       1.00      1.00      1.00         4
                 highest_grossing       0.00      0.00      0.00         1
                            hours       0.00      0.00      0.00         0
                            leave       0.00      0.00      0.00         1
                         location       1.00      1.00      1.00         1
                     maintainence       0.00      0.00      0.00         1
             manufacturing_problems       0.00      0.00      0.00         1
                multipartite virus       0.00      0.00      0.00         0
                             name       0.00      0.00      0.00         0
                            noans       0.00      0.00      0.00         0
                          options       0.00      0.00      0.00         1
                         payments       0.00      0.00      0.00         0
                    payments_modes       0.00      0.00      0.00         2
               predict_performance       0.00      0.00      0.00         0
                           thanks       0.50      1.00      0.67         1
                         training       0.00      0.00      0.00         0

                         accuracy                           0.32        37
                        macro avg       0.11      0.14      0.12        37
                     weighted avg       0.26      0.32      0.29        37
```

Fig7.Performance Metric of Distil BERT

LSTM

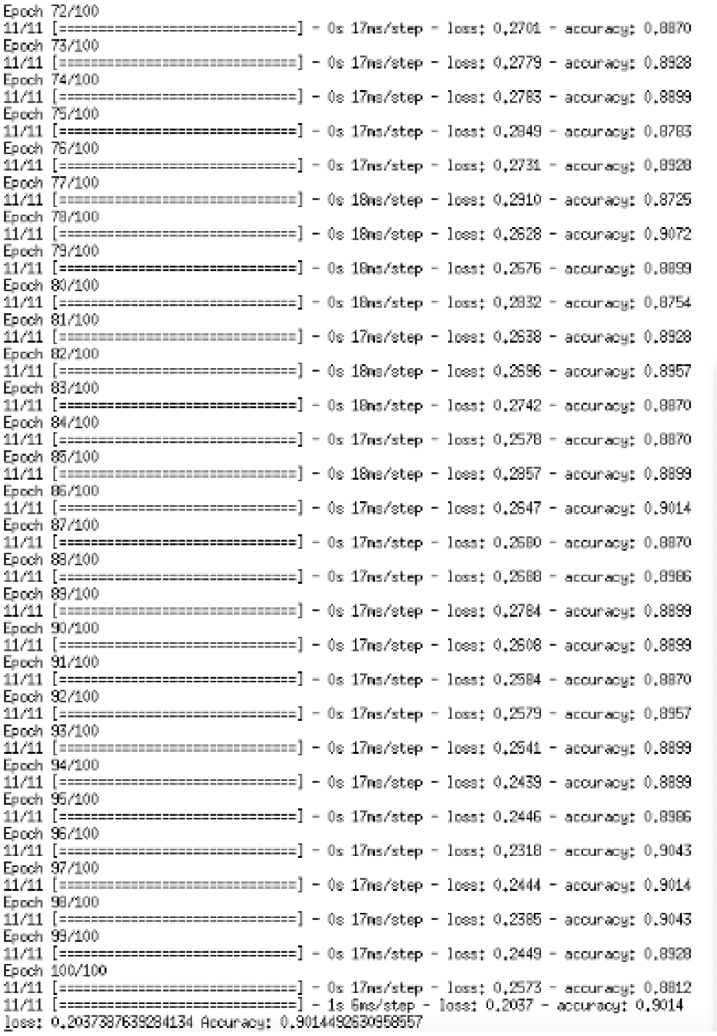
Below attached is the model summary



```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 10)]              0
_________________________________________________________________
embedding_1 (Embedding)      (None, 10, 50)            20000850
_________________________________________________________________
lstm_1 (LSTM)                (None, 10, 128)           91648
_________________________________________________________________
dropout_1 (Dropout)          (None, 10, 128)           0
_________________________________________________________________
lstm_2 (LSTM)                (None, 128)               131584
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 42)                5418
_________________________________________________________________
activation_1 (Activation)    (None, 42)                0
=================================================================
Total params: 20,228,700
Trainable params: 228,650
Non-trainable params: 20,000,850
_________________________________________________________________
Model: "model"
```

Fig8. Model Summary of LSTM Model

Below attached is the performance metrics of LSTM



```
Epoch 72/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2701 - accuracy: 0.8870
Epoch 73/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2779 - accuracy: 0.8928
Epoch 74/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2783 - accuracy: 0.8899
Epoch 75/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2849 - accuracy: 0.8783
Epoch 76/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2731 - accuracy: 0.8928
Epoch 77/100
11/11 [==============================] - 0s 18ms/step - loss: 0.2910 - accuracy: 0.8725
Epoch 78/100
11/11 [==============================] - 0s 18ms/step - loss: 0.2628 - accuracy: 0.9072
Epoch 79/100
11/11 [==============================] - 0s 18ms/step - loss: 0.2676 - accuracy: 0.8899
Epoch 80/100
11/11 [==============================] - 0s 18ms/step - loss: 0.2832 - accuracy: 0.8754
Epoch 81/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2638 - accuracy: 0.8928
Epoch 82/100
11/11 [==============================] - 0s 18ms/step - loss: 0.2696 - accuracy: 0.8957
Epoch 83/100
11/11 [==============================] - 0s 18ms/step - loss: 0.2742 - accuracy: 0.8870
Epoch 84/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2578 - accuracy: 0.8870
Epoch 85/100
11/11 [==============================] - 0s 18ms/step - loss: 0.2857 - accuracy: 0.8899
Epoch 86/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2647 - accuracy: 0.9014
Epoch 87/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2680 - accuracy: 0.8870
Epoch 88/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2688 - accuracy: 0.8986
Epoch 89/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2784 - accuracy: 0.8899
Epoch 90/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2608 - accuracy: 0.8899
Epoch 91/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2584 - accuracy: 0.8870
Epoch 92/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2579 - accuracy: 0.8957
Epoch 93/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2541 - accuracy: 0.8899
Epoch 94/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2439 - accuracy: 0.8899
Epoch 95/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2446 - accuracy: 0.8986
Epoch 96/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2318 - accuracy: 0.9043
Epoch 97/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2444 - accuracy: 0.9014
Epoch 98/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2385 - accuracy: 0.9043
Epoch 99/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2449 - accuracy: 0.8928
Epoch 100/100
11/11 [==============================] - 0s 17ms/step - loss: 0.2573 - accuracy: 0.8812
11/11 [==============================] - 1s 6ms/step - loss: 0.2037 - accuracy: 0.9014
loss: 0.2037387639284134 Accuracy: 0.9014492630958557
```

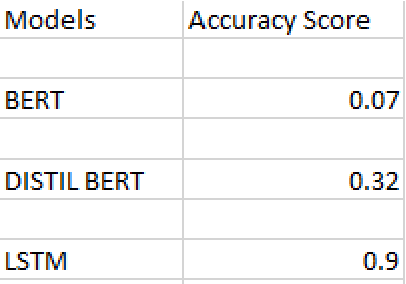
Fig9.Accuracy Score of LSTM Model

]

SUMMARY

We have implemented BERT, Distil BERT and LSTM models on intents.json data. The challenging task was to develop a custom built data for the chatbot since train data does include only 868 rows owing to which performance of BERT and DISTIL BERT was extremely poor since both the models are highly computational and are used for complex language use cases, the understanding we gained through the research is both the models do perform well on the training and test data of minimum 8000 entries respectively which was the major drawback of our use case since it didn’t had test set independently and was populated using train set itself. The study does say that DISTIL BERT’s performance is always considered to be degraded by 3% in comparison to BERT transformer but for our use case it did perform well with the accuracy score of 0.5% which for bert is 0.08%.

After making a comparison of all the model implementation we got to observe that LSTM is having the highest accuracy of 90% and hence we have further used the model for chatbot implementation.



Models	Accuracy Score
BERT	0.07
DISTIL BERT	0.32
LSTM	0.9

CONCLUSION

The further enhancements which can be done to this data is to increase the size of the train set and populate a test data. Also while doing the research for our use case we have come across ConVert which is most popularly used to deal such multi class text classification data which is related to the user queries the explanation of the model is ConveRT is a dual sentence encoder , it is effective, affordable, and quick to train also the size of the ConveRT model is less compared to the BERT model. ConveRT as per the company PolyAI who developed it was trained on Reddit conversational data (context, response). Since ConveRT has no implementation in Tensorflow hence it needs to be implemented from scratch and this will be the stage 2 of the project.

$$J = \sum_{i=1}^{K} S(x_i, y_i) - \sum_{i=1}^{K} \log \sum_{j=1}^{K} e^{S(x_i, y_j)}$$

ConveRT Loss

S(xi,yi) = Similarity between context its corresponding response

S(xi,yj) = Similarity between context and other responses

REFERENCES

https://arxiv.org/abs/1805.10190

https://github.com/sonos/nlu-benchmark/tree/master/2017-06-custom-intent-engines

https://github.com/huggingface/transformers

https://huggingface.co/docs/transformers/model_doc/bert

https://huggingface.co/docs/transformers/model_doc/distilbert

https://www.analyticsvidhya.com/blog/2021/06/lstm-for-text-classification/

https://marutitech.com/make-intelligent-chatbot/

https://paperswithcode.com/task/intent-detection

https://www.sciencedirect.com/science/article/pii/S1877050918320374

https://analyticsindiamag.com/hands-on-guide-to-word-embeddings-using-glove/

APPENDIX

Technical Considerations:

- Installation of Python 3.5 & above, Pycharm and Anaconda are necessary
- Packages like numpy, pandas, matplotlib, sklearn, Tkinter needs to be installed in the computer in order to execute the application
- Steps to setup an instance on AWS and GCP are given here: https://github.com/Rehamanikandan/Final-Project-Group6/blob/main/Code/Setup_AWS_GCP.pdf
- In order to execute the code on cloud (AWS & GCP) please install requirements.txt file

  https://github.com/Rehamanikandan/Final-Project-Group6/blob/main/Code/Requirements.txt

GitHub Repo Link:

- All the documents related to this project are included in the following repo link:

  https://github.com/Rehamanikandan/Final-Project-Group6

- The repo has following folders and file:

- README.md – Defines the structure of the repo
- Code – This folder contains the code sub folders:
  - BERT
  - DATASET
  - DISTIL BERT
  - LSTM
  - UI
  - Requirements.txt
  - SetUp_AWS_GCP.pdf
- Final _Group _Presentation – This folder has the PDF version of group project presentation
- Group_Proposal: This folder includes the proposal submitted by the group.
- Final_Group_Project_Report – This folder includes the complete report of the project in PDF format.
- Individual_Project – This folder contains the folders of report and codes of the group members.