# Chatbot for Intent Recognition

Report by

**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**

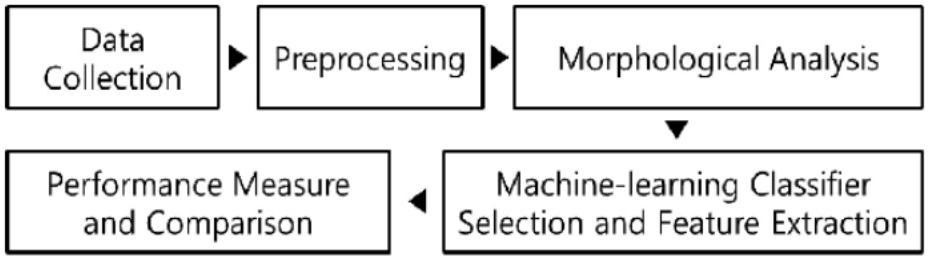

| **TOPIC** | **PAGE NO.** |
|---|---|
| INTRODUCTION | 3 |
| DESCRIPTION OF DATASET | 4 |
| DESCRIPTION OF WORK | 5 |
| MODEL SUMMARY AND RESULTS | 14 |
| CONCLUSION | 16 |
| CODE PERCENTAGE | 16 |
| REFERENCES | 17 |

## 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.




I contributed on working with LSTM model and integrating it to Chatbot UI.

## DESCRIPTION OF DATASET

The motivation for the Intent Recognition dataset has 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:

Tags: Intents

Patterns: User Input

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 WORK

### 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?

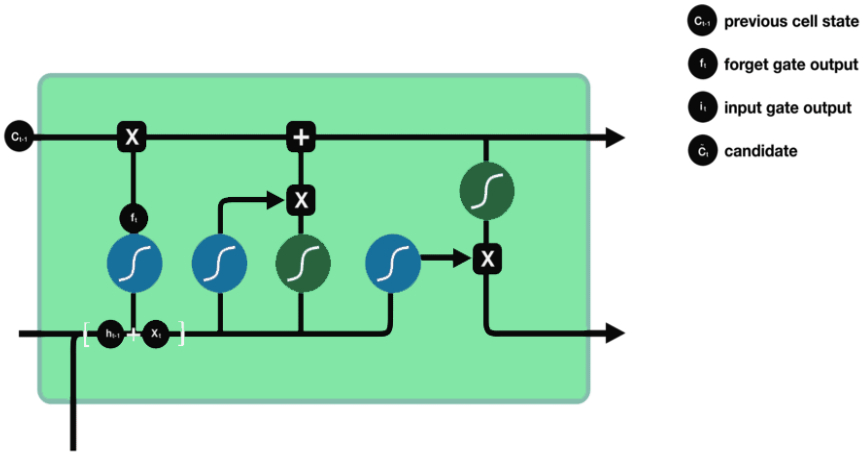
LSTM has 3 main gates.

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

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

### FORGET Gate

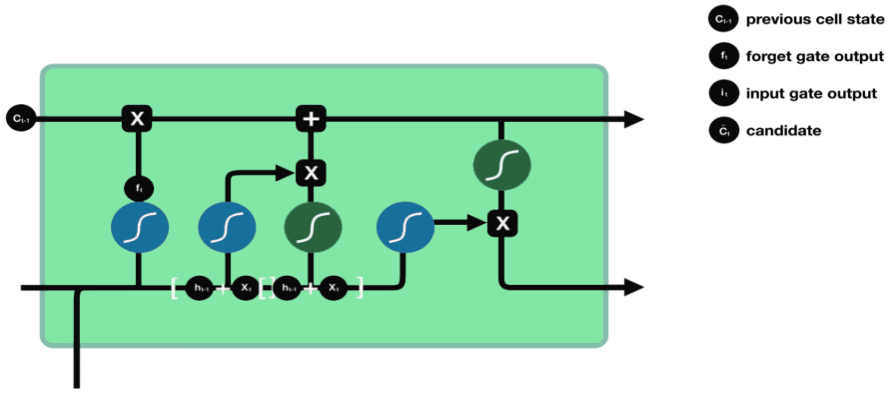
This gate is responsible for deciding which information is kept for calculating the cell state and which is not relevant and can be discarded. The ht-1 is the information from the previous hidden state (previous cell) and xt is the information from the current cell. These are the 2 inputs given to the 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

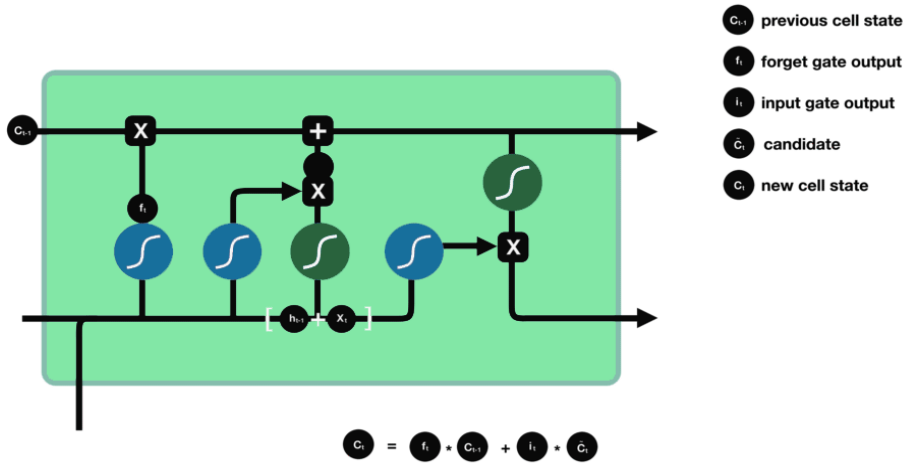
Input Gate updates the cell state and decides which information is important and which is not. As forget gate helps to discard the information, the input gate helps to find out important information and store certain data

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;

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

**EXPERIMENTAL SETUP**

**AWS_GCP_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:

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

For windows,

17. Create a SSH session in mobaxterm
18. Host: External IP from any one of the instances
19. Name: ubuntu
20. Add the private key downloaded from the instances
21. Click ok.
22. Create a folder in the cloud

PyCharm integration:

23. Create a project
24. Tools -> Deployment -> Configuration -> Add SFTP
25. SSH configuration: Host- IP address from instances, name: ubuntu, authorize using the private key pair.
26. Test the connection
27. In mappings, map the remote local path and the cloud folder's path
28. Deployment -> Configuration -> Automatic upload
29. Setup the Interpreter
30. 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: I have used 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)
```

We can observe that I have separated tags,patterns and response to lists with index counter to keep track of tags index. X_train is the patterns and Y_train is the tag index.

```
d = {} #dictionary to store tags with corresponding responses
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)
```

I created dictionary to store tags as keys and responses as values which is helpful in getting predictions of intents with associated responses.Also, I have stored all the tags in tags.txt file.

**MODEL IMPLEMENTATION:**

**LSTM**

For LSTM once the dataset is been built then I further created a data dictionary which is used to store the tags, I have used pre trained LSTM model and Glove model of 6B.50d and 100d .txt as well which is used to tokenize the data and performed one hot encoding on the y_train which contains the tag_index and reponses.

```
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 I have performed LSTM model building by embedding 1 dense layer with softmax activation function.

This Creates a Keras Embedding() layer and loads in pre-trained GloVe 50-dimensional vectors.

Arguments:

word_to_vec_map -- dictionary mapping words to their GloVe vector representation.

word_to_index -- dictionary mapping from words to their indices in the vocabulary (400,001 words)

Returns:

embedding_layer -- pretrained layer Keras instance

```
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
```

Function creating the embed_func model's graph.

Arguments:

input_shape -- shape of the input, usually (max_len,)

word_to_vec_map -- dictionary mapping every word in a vocabulary into its 50-dimensional vector representation

word_to_index -- dictionary mapping from words to their indices in the vocabulary (400,001 words)

Returns:

model -- a model instance in Keras

```
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)
```

I have calculated the best model loss and accuracy.Then, I have tested the model on a single sentence, the model performed well - recognized the right intent and gave the response. Initially, the test_sent is passed through the convert_sent_indices to convert into indices array, then imputed to the model to predict the indexes of the tag and used the argmax for right index , which is then mapped to the corresponding tags and finally generated random response by mapping intents with responses using dictionary which I created earlier

```
#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]))
"""
```

I have taken the user input as input parameter to botResponse function, then gave it to the model to predict the right intent and passed intent index to map the responses.I have used random function to choose random responses for that corresponding intent. I have also gave a condition if the tag is not present in the dictionary.

```
#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])
```

I have used the following functions from utilities.py in main.py wherever required,

```python
def read_glove_vecs(glove_file):
    with open(glove_file, 'r') as f:
        words = set()
        word_to_vec_map = {}
        for line in f:
            line = line.strip().split()
            curr_word = line[0]
            words.add(curr_word)
            word_to_vec_map[curr_word] = np.array(line[1:], dtype=np.float64)

        i = 1
        words_to_index = {}
        index_to_words = {}
        for w in sorted(words):
            words_to_index[w] = i
            index_to_words[i] = w
            i = i + 1
    return words_to_index, index_to_words, word_to_vec_map
```

```python
def softmax(x):
    """Compute softmax values for each sets of scores in x."""
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()
```

```python
def convert_to_one_hot(Y, C):
    Y = np.eye(C)[Y.reshape(-1)]
    return Y
```

## MODEL SUMMARY AND RESULTS

### 1.Model Summary of LSTM Model

```
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: 20,000,050
_________________________________________________________________
Model: "model"
```

## 2.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
```

1. Accuracy of the model, Accuracy = 90.14%.
2. Loss = 0.2037

## CONCLUSION

After making a comparison of all the model implementation I got to observe that LSTM is having the highest accuracy of 90% and hence we have further used the model for chatbot implementation.The further enhancements which can be done to this data is to increase the size of the train set and populate a test data.The model can also be improved by tuning the parameters in a better way to make the accurate predictions.

## CODE PERCENTAGE

My code part contains total 300 lines of code. Out of which, I have referred to some documentations to demonstrate the basic syntaxes of the code. Considering that as copied content my code will be having 210 lines of code and out of which 190 lines would have been modified according to the project needs.Added 90 lines of own code

Code percentage = 210-190 / (210+90) *100 = 6.67%

# REFERENCES


https://arxiv.org/abs/1805.10190

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

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/

https://www.analyticsvidhya.com/blog/2021/06/natural-language-processing-sentiment-analysis-using-lstm/