Chatbot for Intent Recognition

Report by

Adina Dingankar

The George Washington University

Author Note

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

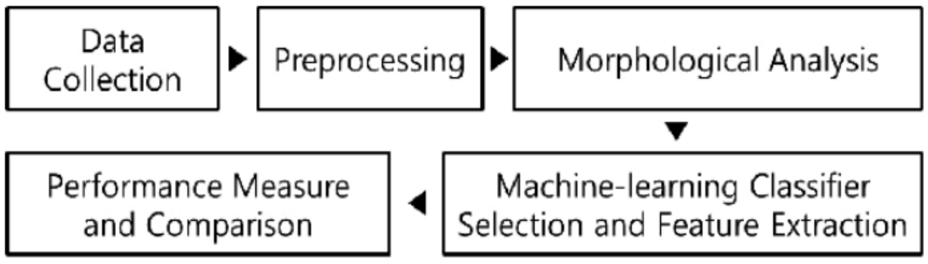
TABLE OF CONTENTS

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 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

My contribution in this project was to make a custom-built .json file with different intents and responses. Taking the inspiration of 'CLINIC50' dataset, several intents were custom added to the dataset. BERT model was developed and tested on the dataset.

Information on Algorithm development:

BERT model of the project was inspired based on the following background study.

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.

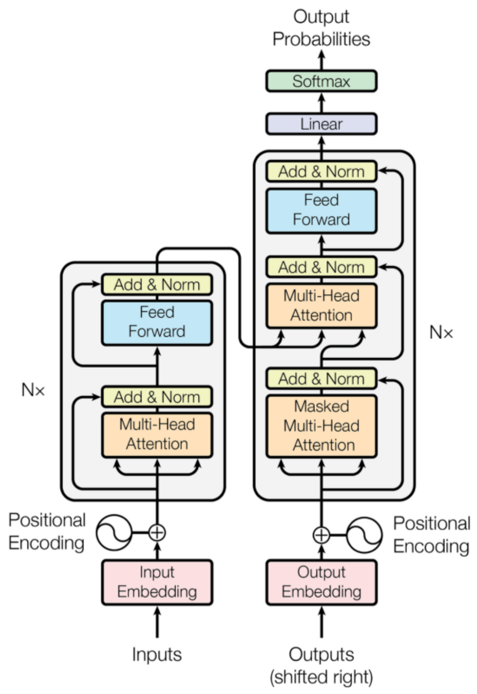


Figure 1: The Transformer - model architecture.

What is the need for BERT?

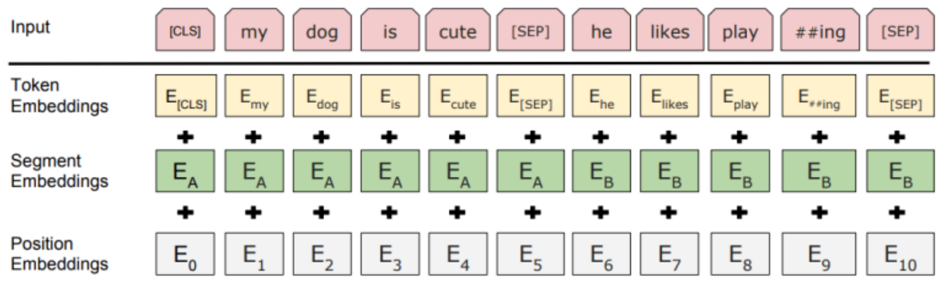
Generally, language models read the input sequence in one direction: either left to right or right to left. This kind of one-directional training works well when the aim is to predict/generate the next word. But in order to have a deeper sense of language context, BERT uses bidirectional training. Sometimes, 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.

EXPERIMENTAL SETUP

AWS_GCP_SETUP:

AWS instance setup:

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

GCP instance setup:

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

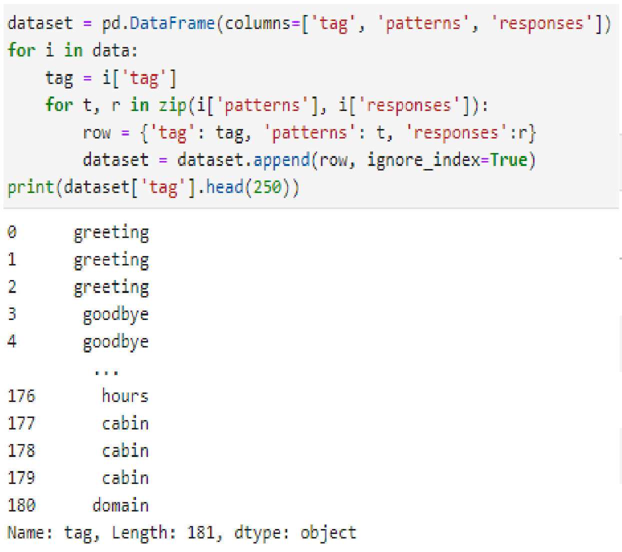
For windows,

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

DATASET

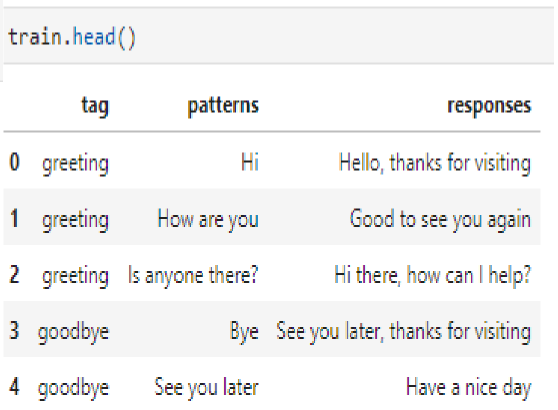
Since the dataset is custom built for our use, we don’t have test data we have implemented .json file and converted into the csv file. The actual size of the train data is 868 rows and 3 attributes we are using the same data and mark it as test data to test the performance of the model. Hence before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

For 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
```

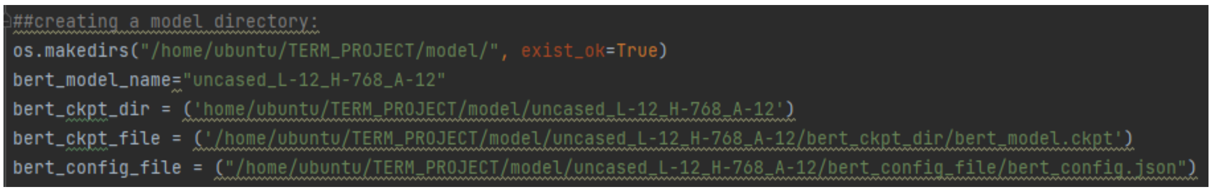


```
train.head()
```

	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

MODEL IMPLEMENTATION:

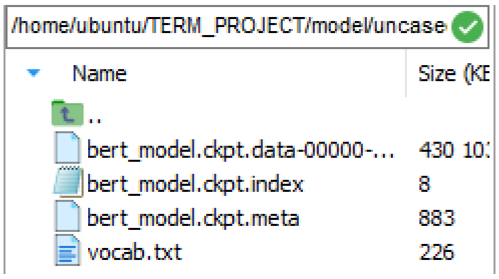
For BERT implementation the packages which are required are Bert-for-tf2, BertModelLayer and tqdm of the specific versions mentioned in the requirement.txt. After installing all the necessary libraries we have mounted our project path to the model which contains the uncased-bert-pretrained-model which 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

_________________________________________________________________
```

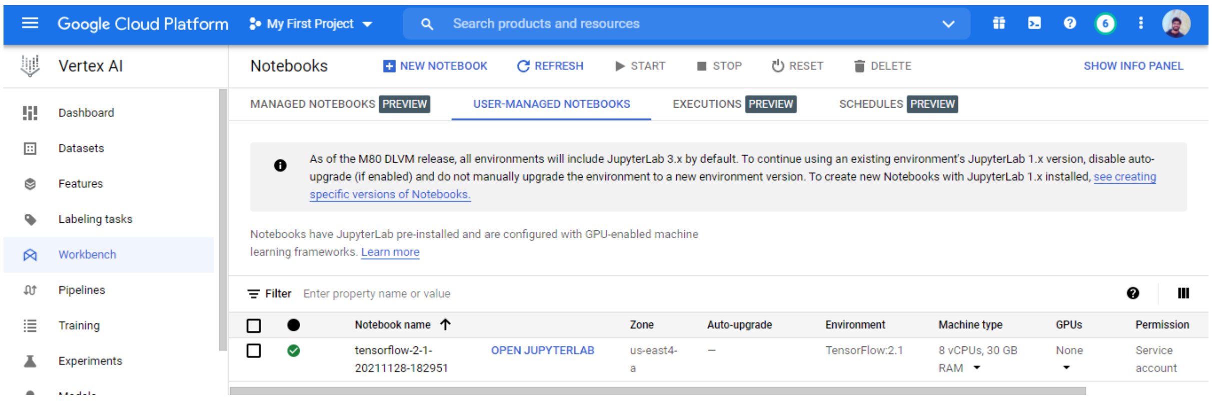
```
train_acc = model.evaluate(data.train_x, data.train_y)
test_acc = model.evaluate(data.test_x, data.test_y)
```

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:

1. Go to AI platforms on console.cloud.google.com.
2. Create a notebook, set the OS -Ubuntu_20.0.4LTS
3. Select Pytorch1.9 setup from the drop down menu (NLTK, GPU,Tensorflow)
4. Core for the CPU: 8 core cpu.
5. GPU: NVIDIA tesla P4/T4
6. 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.

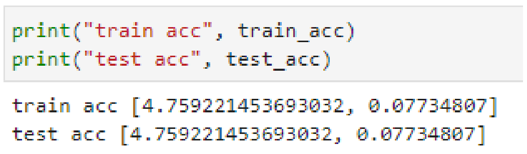


As of the M80 DLVM release, all environments will include JupyterLab 3.x by default. To continue using an existing environment's JupyterLab 1.x version, disable auto-upgrade (if enabled) and do not manually upgrade the environment to a new environment version. To create new Notebooks with JupyterLab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

RESULTS



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

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

MODEL SUMMARY

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.

CONCLUSION

The best model is LSTM based on the accuracy scores generated from the models.

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

CODE PERCENTAGE

My code part contains total 608 lines of code. Out of which, I have referred to official documentations of hugging face transformers to understand the flow and layout of the model. Considering that I have copied 200 lines of code and out of which 50 lines out of code are modified as per the project’s needs. The code percentage of my code is (200-50) / 200+50*100 = 6.65%

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

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