Rapl Chatbot

Project Details:

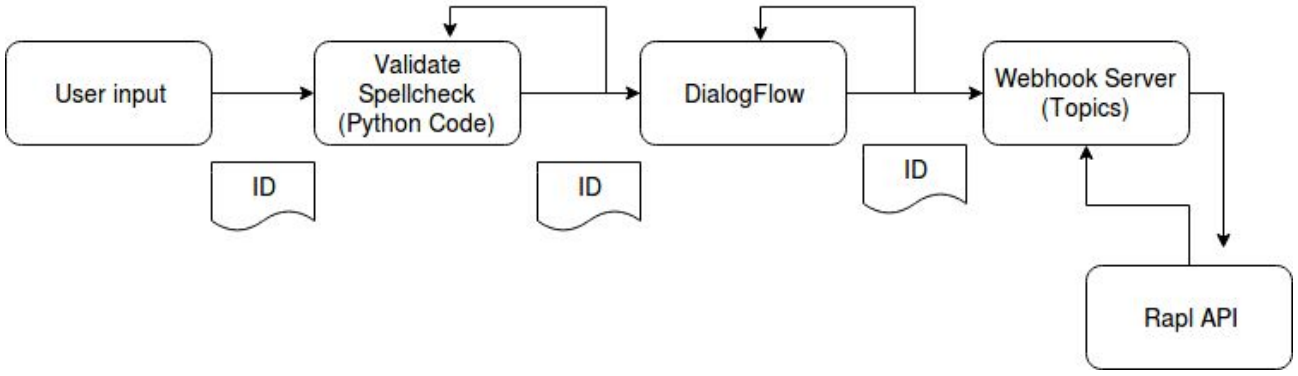
Client side: Angularjs

- Github Repository for AngularJs code: https://github.com/rpranav22/AngularChatbot/tree/rapl (Rapl branch)
- URL for accessing the chatbot on a local system: localhost:4200/

Server side: Python+Flask and Dialogflow:

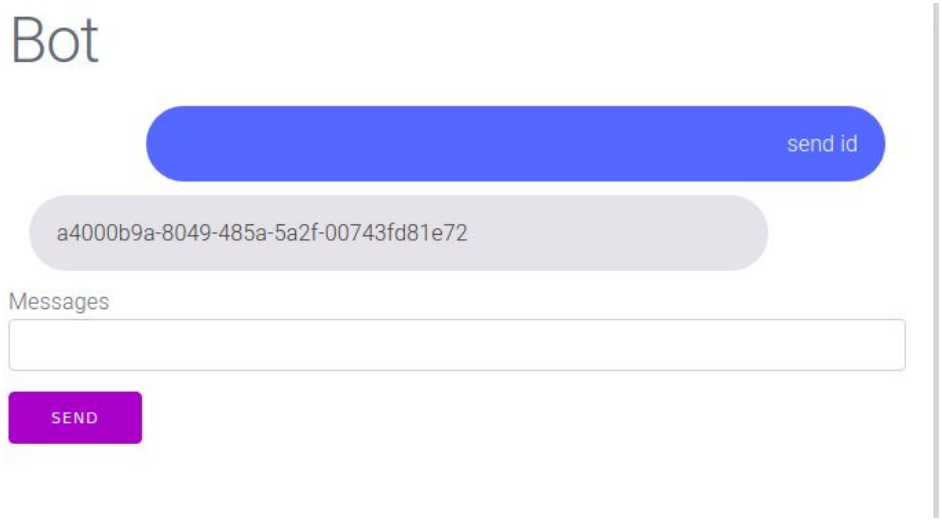
- Python+Flask server is deployed on Heroku and the server is running at url: https://ls-chatbot.herokuapp.com
    - Github Repository: https://github.com/rpranav22/Chatbot_Flask
- Dialoglow server acts as intermediary server between Angularjs and Python.

Flowchart:



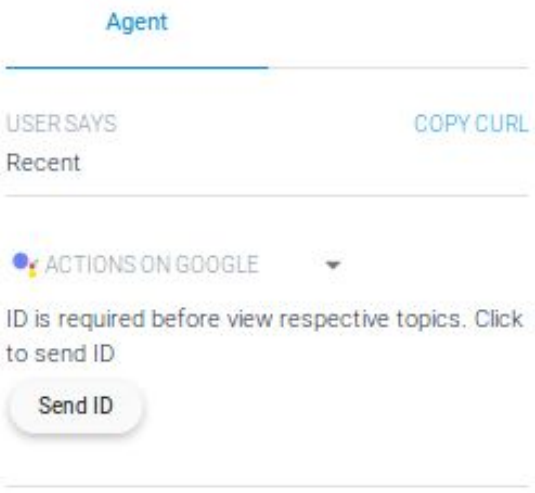
Scenario 1:

Storing and sending the unique User ID upon refreshing the chat window from the client side.



Steps: Refresh the page to generate a unique ID for the specific session that will be sent automatically to the server and stored there.

- All queries from the client side are sent through two modules in the pipeline before the client receives a response.
- Firstly, all messages sent by the user initially undergoes a spellcheck which is triggered from the Angularjs script by sending a post request to the spellcheck server which is deployed on Heroku.
- Once the user input is validated by *spellcheck*; which in this case is always valid – will be sent through to the dialogflow agent which handles a bulk of the initial user query processing.
- The agent then matches the input to a specific intent and fetches a response. In this case, since webhook is enabled, the query is further forwarded to an external server via a POST request.
- Our external server is written in Python and the REST services are enabled by the Flask framework to handle requests.



- Clicking on the above displayed “Send ID” button, our webhook enabled external server is triggered and receives a JSON object with all details that looks like this:

RAW API RESPONSE | FULFILLMENT REQUEST | FULFILLMENT RESPONSE | FULFILLMENT STATUS

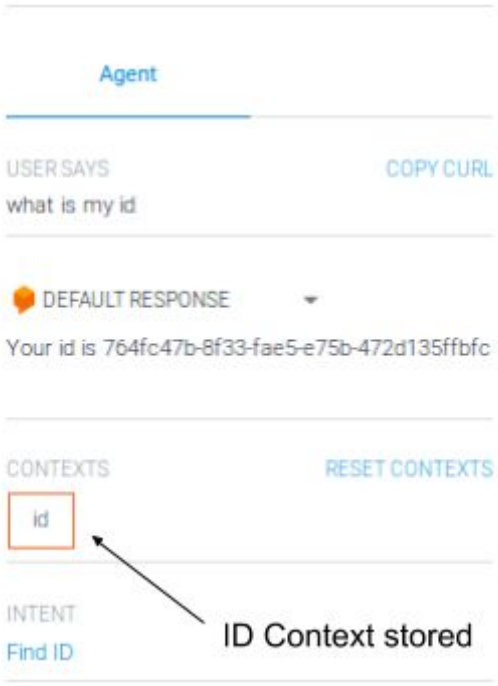

```
{
  "responseId": "a912052d-3212-41f7-9090-84e3bc9fa7ab",
  "queryResult": {
    "queryText": "send ID",
    "parameters": {},
    "allRequiredParamsPresent": true,
    "fulfillmentMessages": [
      {
        "text": {
          "text": [
            ""
          ]
        }
      }
    ],
    "intent": {
      "name": "projects/chatbot-6fb36/agent/intents/e4faa770-a6c8-4d74-bfd2-04e806aeb4d7",
      "displayName": "send_id"
    },
    "intentDetectionConfidence": 1,
    "languageCode": "en"
  },
  "originalDetectIntentRequest": {
    "payload": {}
  },
  "session": "projects/chatbot-6fb36/agent/sessions/764fc47b-8f33-fae5-e75b-472d135ffbfc"
}
```

- From the session field, a session ID is extracted for the specific user and stored in the server through a session variable.
    - The field is of the format:
      `projects/$(PROJECT_ID)/agent/sessions/$(SESSION_ID)`

```
if intent == "send_id":
    session['id'] = "".join(json_data["session"].split('/')[-1:])
    response['fulfillmentText'].append(session['id'])
    id_context["name"] = preID + session['id'] + postID
    id_context["parameters"]["id.original"] = session["id"]
    id_context["parameters"]["id"] = session["id"]
    response["outputContexts"].append(id_context)
    quickReply["quickReplies"]["title"] = "Click if you want to view all topics."
    quickReply["quickReplies"]["quickReplies"].append("show topics")
    response["fulfillmentMessages"] = [quickReply]
    return jsonify(response)
```

- Once the ID is extracted and stored in the server, it must also respond to the Dialogflow agent with a response as well as the output contexts (optional) to be stored by the agent.
- The response from the server will look like this:

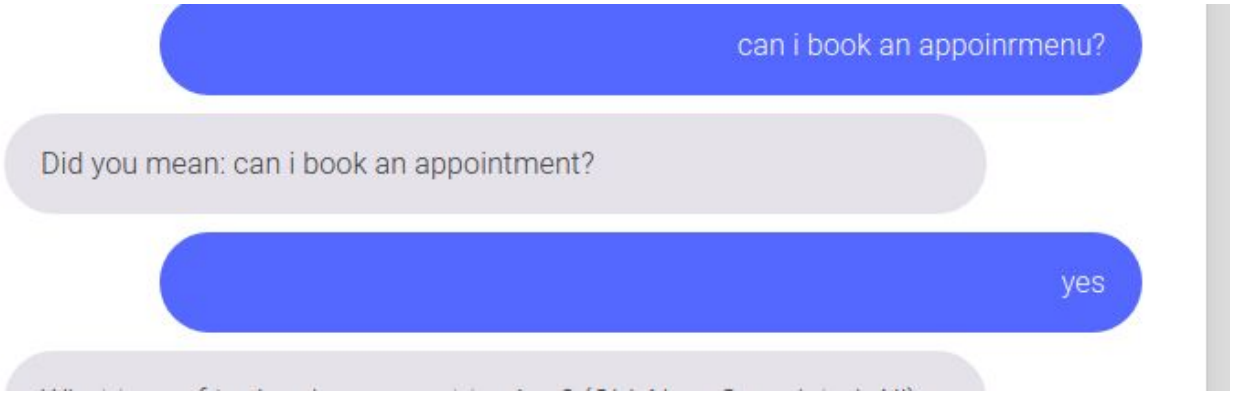
- This response will update the context in Dialogflow and once we query the agent, we can use this ID as an input context.



- Depending on the platform that the user is querying the system from, the message is displayed accordingly to the user.

Scenario 2

Spellcheck and validate each input before it is forwarded to the Dialogflow agent or the server.



- Spell check is implemented on all user queries regardless of the input.
- Unlike the previous scenario, this step involves only one step where the query is sent to server which is deployed on Heroku solely for the purpose of spell check.
- The code on the client side for this looks like:

```
const userMessage = new Message(msg, 'user','text');

this.update(userMessage);

const data = this.httpClient.post ('https://ls-chatbot.herokuapp.com/spellcheck', body, {withCredentials: true})
.subscribe(
  res => {
    console.log(body.get('msg'))
    const rec = res as JSON
    console.log(rec['query'], "spellcheck: ", rec['spellcheck'])
    console.log(rec, rec['topic']);

    if(rec['spellcheck'] == false){
      var botMessage = new Message(rec['text'], 'bot', 'text')
      console.log(botMessage);
      this.update(botMessage);
    }
    else{
      return this._client.textRequest(rec['query'])
    .then(res => {
      const speech = res.result.fulfillment.speech;
      const action = res.result.action;
```

- In this code, the client handles the response from the spellcheck server and checks if it is valid or not. If it is not valid, a suggested correction returned to the user.
- Further, depending on the user input, the corrected query will be forwarded to the agent.
- On the server side of spell check, the system is able to find possible word corrections which have a maximum of two edits. It is however possible to increase the number of edits, but this has been reduced for time efficiency.
- For each word over 4 letters in length, the code produces a list words that are close to the given word and their corresponding probabilities of likeliness to be replaced.

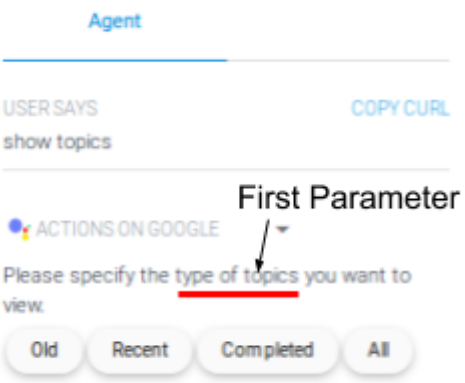
- *The following excerpt is from app.py in function* `spellcheck()`:

```
@app.route('/spellcheck', methods=['POST'])
def spellcheck():
    sc = SC()
    userInput = request.form['msg']
    int(userInput, session)
unction should be lowercase more... (Ctrl+F1)
    response['text'] = []
    if 'spellcheck' in session:
        print("entering session spellcheck")
        if userInput == "yes":
            response['spellcheck'] = True
            response['query'] = session['ques_corrected']
            response['text'].append("fine for now")
            del session['spellcheck']
            del session['ques_corrected']
            del session['ques']
        elif userInput == "no":
            response['spellcheck'] = True
            response['query'] = session['ques']
            response['text'].append("going ahead with {}".format(session['ques']))
            del session['spellcheck']
            del session['ques_corrected']
            del session['ques']
        else:
            del session['spellcheck']
            del session['ques_corrected']
            del session['ques']
    else:

        word_list = userInput.split(' ')
        sq = list(filter(lambda x: x, map(lambda x: re.sub(r'[^A-Za-z]', '', x), word_list)))
        print("\nsq: ", sq)
        corrected = []
        # self.stopWords = stopwords.words("english")
        for word in sq:
            # if word in stopwords.words("english"):
            #     continue
            poss = sc.correction(word)
            if word != poss and not word in stopwords.words("english") and len(poss) > 4:
                corrected.append(poss)
            else:
                corrected.append(word)

        print("corrected: ", corrected)

        if sq != corrected:
            corrected = ' '.join(corrected)
            print("\n\nDid you mean: {}?".format(corrected))
            response['text'].append("Did you mean: {}?".format(corrected))
            response['query'] = corrected
            response['spellcheck'] = False
            session['spellcheck'] = False
            session['ques_corrected'] = corrected
            session['ques'] = userInput
        else:
            response['spellcheck']=True
            response['query'] = userInput
            response['text'].append("fine for now")

    return jsonify(response)
```

- From this, the server responds with the most likely suggestion to the user.

Scenario 3

Slot Filling for required parameters and returning dynamic text for suggested input. For example, the user requires two parameters before viewing topic; they would be required to input their ID and topic type (old, recent, completed) specifically.



- When the user queries the system to show topics, the agent responds with the first required parameter that needs to be fulfilled.
    - NOTE: If the user had queried “show **old** topics” or “show **recent** topics”, this would not have been necessary as the parameter would have been satisfied.
- Now the agent prompts the user to select one of the options for topic type using the suggestion chips on the screen.
- Once the agent stores the topic type, it checks if the user ID (Second Parameter) has already been stored in the server for this session. This is done by enabling slot filling to the webhook.

- If the ID has already been stored in the server, all conditions to show topics will have been satisfied. Now, the server starts generating all the topics specific to the user and returns it as options to select and ask further questions.
- By enabling webhook call for slot filling we can manipulate the response depending on the parameters as follows:

Excerpt from app.py under function `response()`

```
    elif intent == "get_topics":
        allFiles = getTopics(id="2345")


        for top in allFiles:
            quickReply["quickReplies"]['quickReplies'].append("select {}".format(top))

        params = json_data["queryResult"]["parameters"]
        if params["topictype"] == "":
            quickReply["quickReplies"]["title"] = "Please specify the type of topics you want to view. "
            topicTypes = ["Old", "Recent", "Completed", "All"]
            quickReply["quickReplies"]["quickReplies"] = topicTypes
            response["fulfillmentMessages"]= [quickReply] Suggestion Chips
            response['fulfillmentText'] = "What type of topics do you want to view? (Old, New, Completed, All)"
            # text["text"]["text"] = response["fulfillmentText"]
            # response["fulfillmentMessages"].append(text)
        elif params["id"] == "":
            quickReply["quickReplies"]["title"] = "ID is required before view respective topics. Click to send ID"
            quickReply["quickReplies"]["quickReplies"] = ["send ID"]
            response['fulfillmentMessages'] = [quickReply]
            response['fulfillmentText'].append("You have not entered your ID yet, please do so.")
            text["text"]["text"] = response["fulfillmentText"]
            response["fulfillmentMessages"].append(text)
        else:  Case when both parameters are satisfied
            response["fulfillmentMessages"] = [quickReply]
            response['fulfillmentText'].append("Here are all your topics: pick one. \n{}".format(" ".join(allFiles)))
            text["text"]["text"] = response["fulfillmentText"]
            response["fulfillmentMessages"].append(text)

        # del response['fulfillmentText']
        print("sess: ", session)
        return jsonify(response)
```

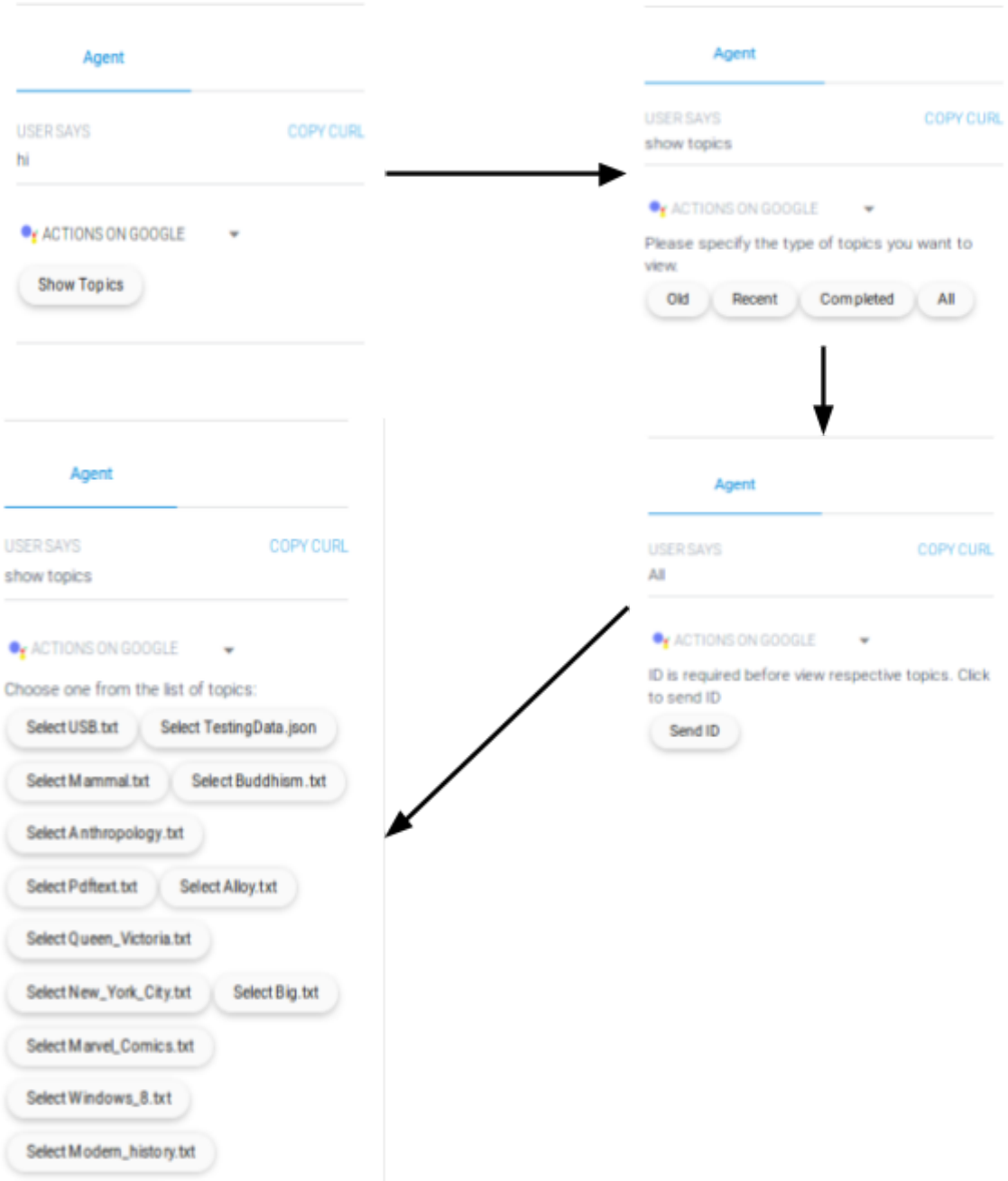
- Suggestion chips are provided from the server side depending on the platform the chatbot is being accessed form. I the response JSON, the *fulfillmentText* field corresponds to the default text response and the *fulfillmentMessages* field corresponds to the rich text to be returned based on the platform. The template for this response is as follows:

Excerpt from app.py under function `response()`

```
237     quickReply = {
238
239         "platform": "ACTIONS_ON_GOOGLE",
240         "quickReplies": {
241             "title": "Choose one from the list of topics: ",
242             "quickReplies": [
243             ]
244         }
245     }
```

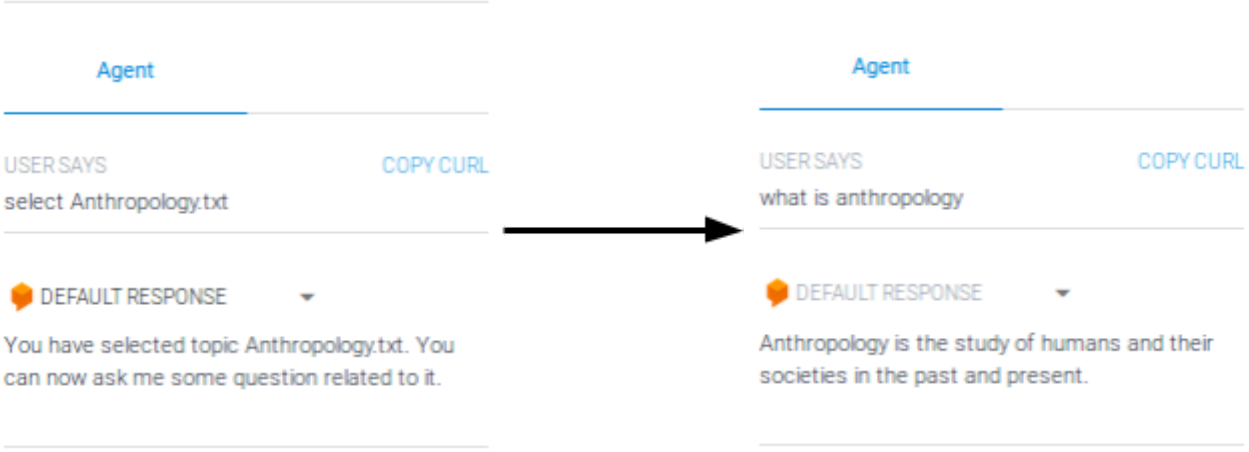
- In the case that both parameters are already satisfied all topics are returned to the user as suggestion chips and the user can pick any topic to ask further questions related to it.

The entire flow of this scenario would be as follows:



Scenario 4:

If the user wants to query the system on a specific topic, user can do so by selecting a topic first and then asking a related question.



- Depending on the selected topic, the python server loads the specified files if it exists.
- Once the user queries the agent, the system processes the query to be able to answer the specific question.
- The system uses a TFIDF model to rank the various paragraphs in the query and returns the most relevant paragraph to the query.

The following excerpt is from DocumentRetrievalModel.py in function `query()`:

```
107     # To find answer to the question by first finding relevant paragraph, then
108     # by finding relevant sentence and then by procssing sentence to get answer
109     # based on expected answer type
110     # Input:
111     #           pQ(ProcessedQuestion) : Instance of ProcessedQuestion
112     # Output:
113     #           answer(str) : Response of QA System
114     def query(self,pQ):
115
116         # Get relevant Paragraph
117         relevantParagraph = self.getSimilarParagraph(pQ.qVector)
118
119         # Get All sentences
120         sentences = []
121         for tup in relevantParagraph:
122             if tup != None:
123                 p2 = self.paragraphs[tup[0]]
124                 sentences.extend(sent_tokenize(p2))
125
126         # Get Relevant Sentences
127         if len(sentences) == 0:
128             return "Oops! Unable to find answer"
129
130         # Get most relevant sentence using unigram similarity
131         relevantSentences = self.getMostRelevantSentences(sentences,pQ,1)
132
133         # AnswerType
134         aType = pQ.aType
135
136         # Default Answer
137         answer = relevantSentences[0][0]
138
139         ps = PorterStemmer()
140         # For question type looking for Person
141         if aType == "PERSON":
142             ne = self.getNamedEntity([s[0] for s in relevantSentences])
```

- Once we have the paragraph, we search for an answer line by line depending on the expected answer type which is generated by processing the query.
- With the most relevant line we start to compare the most relevant ngram to find a more accurate answer.
- Once we get this final answer, it is returned to the user in the form of simple text.