Chatbot Project Report

By Qinyue Wang

Content

1 Introduction

Realizing chatbots involves knowledge about machine learning and natural language processing (NLP). With an aspect of informed decision making, there are generally two methods: the search and expert question and answer. In the search, which is a traditional method, a decision maker who has a question distills 2-3 keywords of the question and gives them to the search engine that would find documents containing keywords. When the research engine delivers documents based on popularity, the decision maker will find answers by reading the documents and analyze evidence. However, in the expert question and answer, the decision maker asks a natural language question. The expert understands the question, produces possible answers and evidence, analyzes the evidence, computes confidence and delivers responses, evidence and confidence. The later method is more time-saving and understands questions more accurately. But the expert method is a challenge all the time. In the natural language processing, the automatic open-minded question answering is a long-standing challenge in artificial intelligence to emulate the human expertise. users give rich natural language questions over a broad domain of knowledge and then robots deliver precise answers, accurate confidences and consumable justifications in the fast response time. As the real languages are really hard to understand precisely, automatic learning for reading is crucial to semantics understanding. Moreover, for the broad domain, we focus on the reusable NLP technology to analyze texts. And the structured sources provide background knowledge for interpreting texts. Watson team made great contribution to NLP. The technology behind Watson is DeepQA, which generates and scores hypotheses using an extensible collection of NLP, machine learning and reasoning algorithms.

Currently, intelligent chatbots involve in a great many of scenarios, such as intelligent customer service, virtual robots, smart home, smart watches, diagnostic assistance, contact centers and so on. Chatbots help users to obtain timely and efficiently assistance and information. And they can be applied on business applications. On many apps, chatbots are utilized to answer questions, cutting down man power cost. So, we try to create a chatbot searching for real-time stock information and apply the programs on a chat app.

2 Related knowledge

To achieve specific functions of intelligent chatbots, we need to learn plenty of techniques about regular expression, intent recognition, entity extraction, Rasa NLU, database query and state machine. The framework of the related knowledge is depicted in Fig. 1. And the knowledge will be detailed in this section.

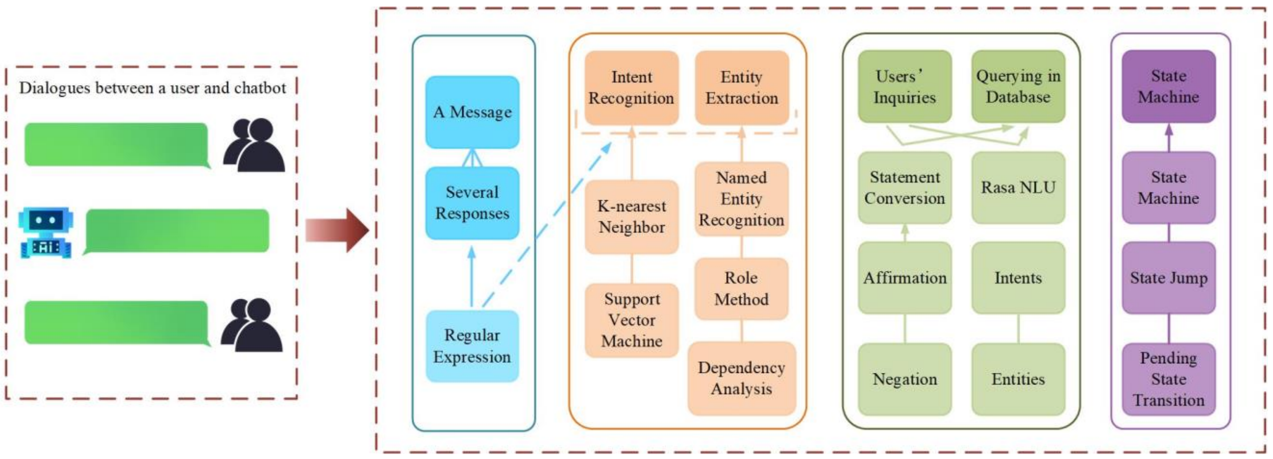
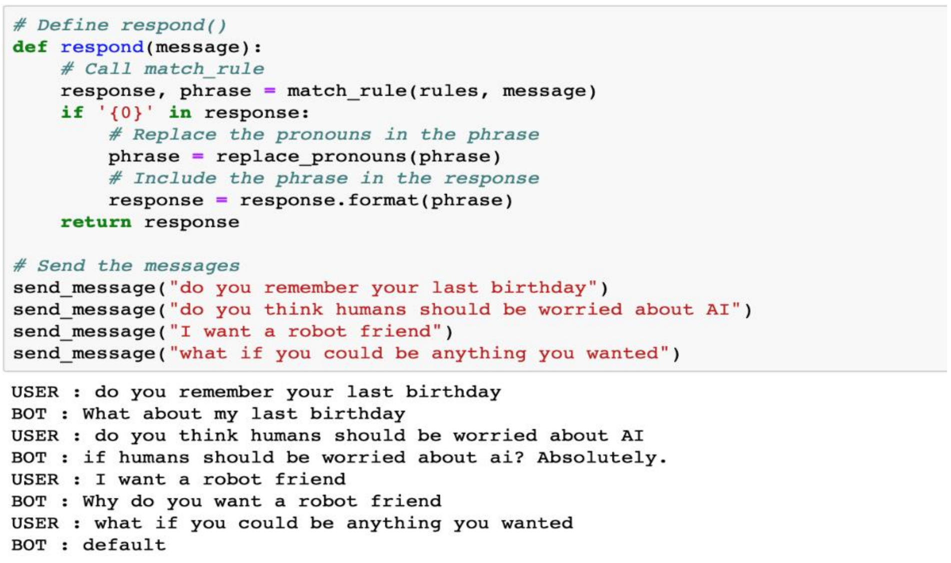


Fig. 1: The framework of related knowledge.



```
# Define respond()
def respond(message):
    # Call match_rule
    response, phrase = match_rule(rules, message)
    if '{0}' in response:
        # Replace the pronouns in the phrase
        phrase = replace_pronouns(phrase)
        # Include the phrase in the response
        response = response.format(phrase)
    return response

# Send the messages
send_message("do you remember your last birthday")
send_message("do you think humans should be worried about AI")
send_message("I want a robot friend")
send_message("what if you could be anything you wanted")
```

```
USER : do you remember your last birthday
BOT : What about my last birthday
USER : do you think humans should be worried about AI
BOT : if humans should be worried about ai? Absolutely.
USER : I want a robot friend
BOT : Why do you want a robot friend
USER : what if you could be anything you wanted
BOT : default
```

Fig. 2: The respond function of an echo robot.

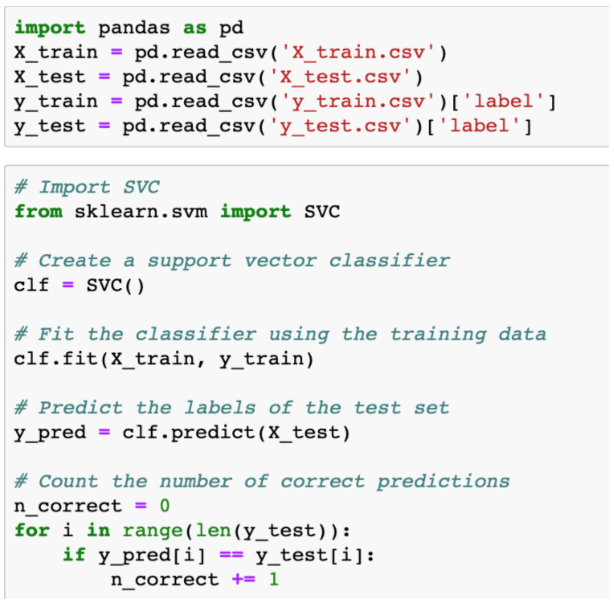
2.1 Building an echo chatbot in python

The earliest chatbot is named as ELIZA that dates back to the 1960s. Because ELIZA is based on the regular expression, in order to implement a simple chatbot with ELIZA style, the regular expression would be used to extract the meaning of texts. Initially, we set templates to display dialogues and define a function send_message( ) that calls a function response( ) and prints messages. Aiming to randomly-chosen answers on each message, we declare rules and utilize them to determine response sentences. In terms of the matching, regular expression can match messages with our given patterns in the rules dictionary through the function re.match( ). According to the results of matching, we move on to the next step. When matching exist, a random response and a phrase, the message that doesn't include the matching part would be obtained. Although the response is incomplete at this time, format( ) fills the phrase into the response in the function respond ( ). And then in line with daily conversations, we replace the original grammatical person from the phrase with the correct grammatical person through a function defined as replace_pronouns( ). Ultimately, the complete and correct response would reply to a user. The code and output are shown in Fig. 2.

2.2 Intent recognition and entity extraction

For intents, an intent can be expressed in diver kinds of ways. For entities, named entity recognition include three major categories: entities, time and numbers and seven small categories: personal names, organization names, place names, time, dates, currency and percentage. Both intents and entities are crucial in messages. we can use regular expression that is based on string matching to understand intents and entities, but the debugging of this method is comparatively difficult. Therefore, we choose to recognize intents and extract entities by machine learning. At first, word vectors are presented to determine semantics of words. When computer languages express sentences, characters, words and sentences could be regarded as a unit vector. The word vectors select a word as a unit vector, which means that a numeric vector with fixed length represents a word. Through the spaCy model, we can take advantage of the pre-training data set. In the spaCy model, each word becomes a 300-dimensional vector. And the distance between two words is equal to the angle between two vectors. The cosine of two vectors indicates the semantic similarity of two words.

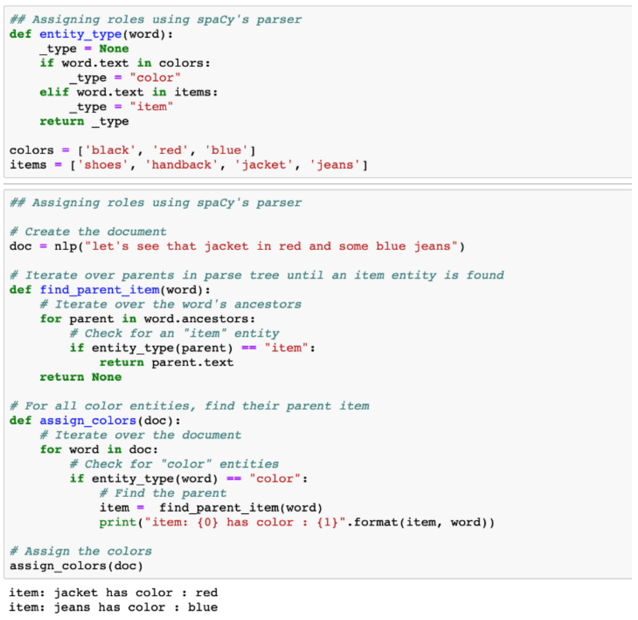
The intent recognition and classification relate to two methods: the k-nearest neighbor (KNN) classification method and support vector machine (SVM) classification method. KNN needs training data to assign every sentence with their intents. And then an intent of one of examples that is most similar to that of given sentence would be found. The intent is viewed as the best guess and interpretation. The KNN classification method is simple, and the process could be improved by SVM classification method. In the SVM, central points of vectors are focused on and new sentences are compared with these central points. The code of intent classification with sklearn is shown below:



```
import pandas as pd
X_train = pd.read_csv('X_train.csv')
X_test = pd.read_csv('X_test.csv')
y_train = pd.read_csv('y_train.csv')['label']
y_test = pd.read_csv('y_test.csv')['label']
```

```
# Import SVC
from sklearn.svm import SVC

# Create a support vector classifier
clf = SVC()

# Fit the classifier using the training data
clf.fit(X_train, y_train)

# Predict the labels of the test set
y_pred = clf.predict(X_test)

# Count the number of correct predictions
n_correct = 0
for i in range(len(y_test)):
    if y_pred[i] == y_test[i]:
        n_correct += 1
```

(a)



```
## Assigning roles using spaCy's parser
def entity_type(word):
    _type = None
    if word.text in colors:
        _type = "color"
    elif word.text in items:
        _type = "item"
    return _type

colors = ['black', 'red', 'blue']
items = ['shoes', 'handback', 'jacket', 'jeans']
```

```
## Assigning roles using spaCy's parser

# Create the document
doc = nlp("let's see that jacket in red and some blue jeans")

# Iterate over parents in parse tree until an item entity is found
def find_parent_item(word):
    # Iterate over the word's ancestors
    for parent in word.ancestors:
        # Check for an "item" entity
        if entity_type(parent) == "item":
            return parent.text
    return None

# For all color entities, find their parent item
def assign_colors(doc):
    # Iterate over the document
    for word in doc:
        # Check for "color" entities
        if entity_type(word) == "color":
            # Find the parent
            item =  find_parent_item(word)
            print("item: {0} has color : {1}".format(item, word))

# Assign the colors
assign_colors(doc)
```

```
item: jacket has color : red
item: jeans has color : blue
```

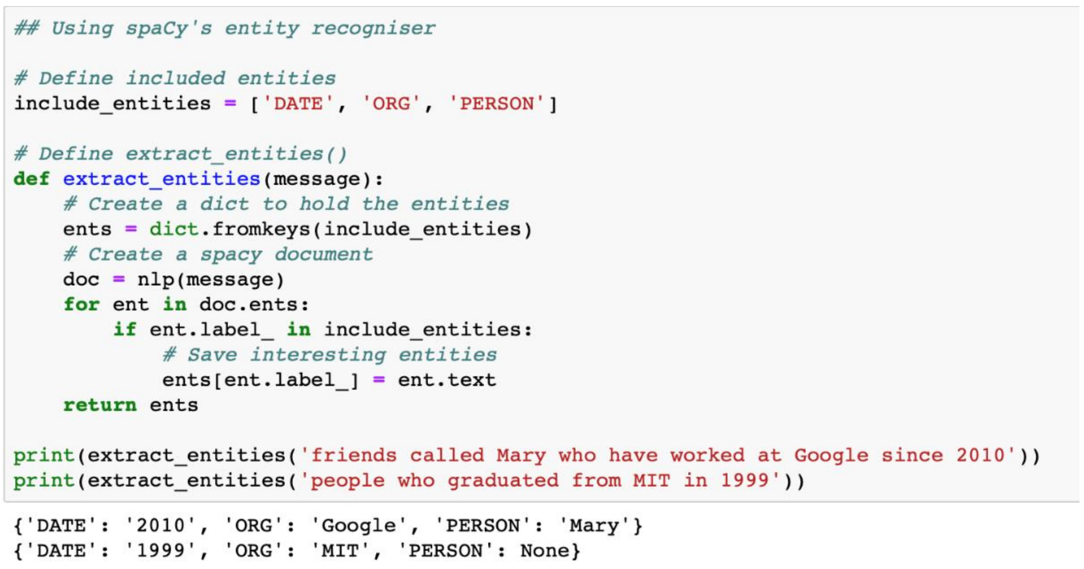
(b)

Fig. 3: The SVM classification method and dependency analysis method.

In Fig. 3a, the labels y_pred are compared with the labels y_test. And the number of correct predictions is printed.

There are three methods from the aspect of entity extraction: pre-built named entity recognition method, roles method and dependency analysis method. The role method judges entities through regular expressions. Take “from…to…” for example. We can obtain the places by re.compile(‘.*from(.*)to(.*)’). In addition, the dependency analysis method needs to find ancestors of words and dependencies between words. According to Fig. 3b, the type of a word is first determined and then find the ancestor of word. In this example, the type color of a word is found initially and then according to this word, the text of an ancestor that belongs to the type item would be returned. At last, a certain item corresponds to its own color.

As to the pre-built entity recognition, it is relatively simple and convenient applies spaCy’s built-in entities for automatic mining. The code is illustrated below in Fig. 4. At the beginning of extracting entities, we create a dict only with keys and a spaCy document according to a message. For the entities in the document, if the label of an entity is in the initial defined array of entities, the text of this entity would be held in the dict.



```
## Using spaCy's entity recogniser

# Define included entities
include_entities = ['DATE', 'ORG', 'PERSON']

# Define extract_entities()
def extract_entities(message):
    # Create a dict to hold the entities
    ents = dict.fromkeys(include_entities)
    # Create a spacy document
    doc = nlp(message)
    for ent in doc.ents:
        if ent.label_ in include_entities:
            # Save interesting entities
            ents[ent.label_] = ent.text
    return ents

print(extract_entities('friends called Mary who have worked at Google since 2010'))
print(extract_entities('people who graduated from MIT in 1999'))
```

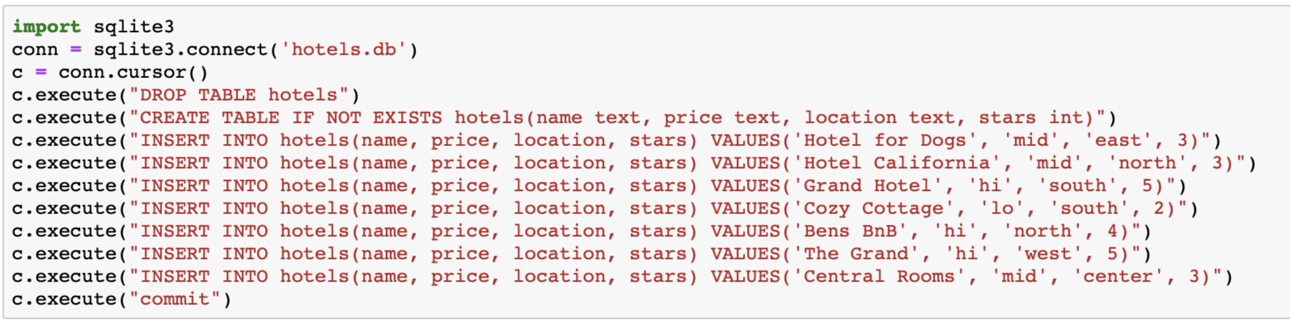
```
{'DATE': '2010', 'ORG': 'Google', 'PERSON': 'Mary'}
{'DATE': '1999', 'ORG': 'MIT', 'PERSON': None}
```

Fig. 4: The pre-built names entity recognition method.

2.3 Natural language and database

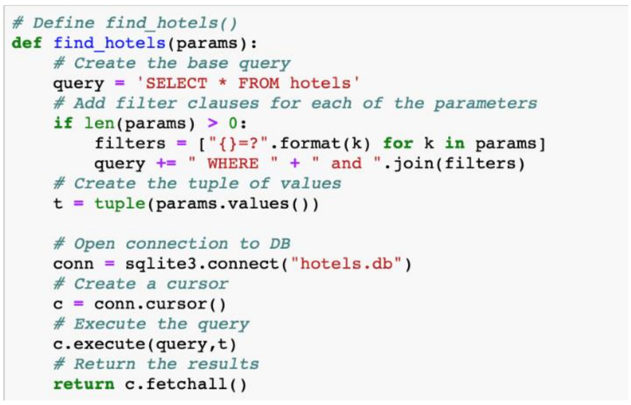
In virtual assistants, chatbots have to interact with database or API to get information about the outside world. So, before achieving functions about querying, we have to build a database of proprietary data and need configuration files and training data. Exactly, we utilize Rasa NLU to recognize intents and extract entities. In the configuration file, we use sklearn, a python library for machine learning. And algorithms could be called by spaCy and sklearn pipelines. For the training data, they help us know the intents of dialogues. Under the effect of configuration file and training data, an interpreter model is created.

Firstly, the database should be built and connected (Fig. 5). Simultaneously, to query information successfully, we use SQL statements to search for data in the database. Through the interpreter model, entities could be extracted from text and the value of an entity is also attained. And then we use these attained parameters to create correct SQL query statements (Fig. 6a). After that, the integrated SQL statements can be applicable to database queries. Lastly, we get the results of the query (Fig. 6b). The code is shown below:



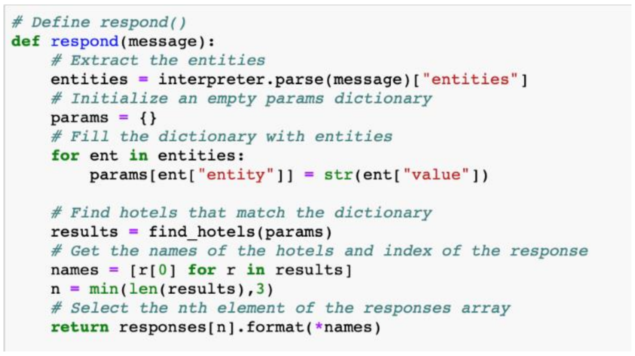
```
import sqlite3
conn = sqlite3.connect('hotels.db')
c = conn.cursor()
c.execute("DROP TABLE hotels")
c.execute("CREATE TABLE IF NOT EXISTS hotels(name text, price text, location text, stars int)")
c.execute("INSERT INTO hotels(name, price, location, stars) VALUES('Hotel for Dogs', 'mid', 'east', 3)")
c.execute("INSERT INTO hotels(name, price, location, stars) VALUES('Hotel California', 'mid', 'north', 3)")
c.execute("INSERT INTO hotels(name, price, location, stars) VALUES('Grand Hotel', 'hi', 'south', 5)")
c.execute("INSERT INTO hotels(name, price, location, stars) VALUES('Cozy Cottage', 'lo', 'south', 2)")
c.execute("INSERT INTO hotels(name, price, location, stars) VALUES('Bens BnB', 'hi', 'north', 4)")
c.execute("INSERT INTO hotels(name, price, location, stars) VALUES('The Grand', 'hi', 'west', 5)")
c.execute("INSERT INTO hotels(name, price, location, stars) VALUES('Central Rooms', 'mid', 'center', 3)")
c.execute("commit")
```

Fig. 5: Creating a database.



```
# Define find_hotels()
def find_hotels(params):
    # Create the base query
    query = 'SELECT * FROM hotels'
    # Add filter clauses for each of the parameters
    if len(params) > 0:
        filters = ["{}=?".format(k) for k in params]
        query += " WHERE " + " and ".join(filters)
    # Create the tuple of values
    t = tuple(params.values())

    # Open connection to DB
    conn = sqlite3.connect("hotels.db")
    # Create a cursor
    c = conn.cursor()
    # Execute the query
    c.execute(query,t)
    # Return the results
    return c.fetchall()
```

(a)



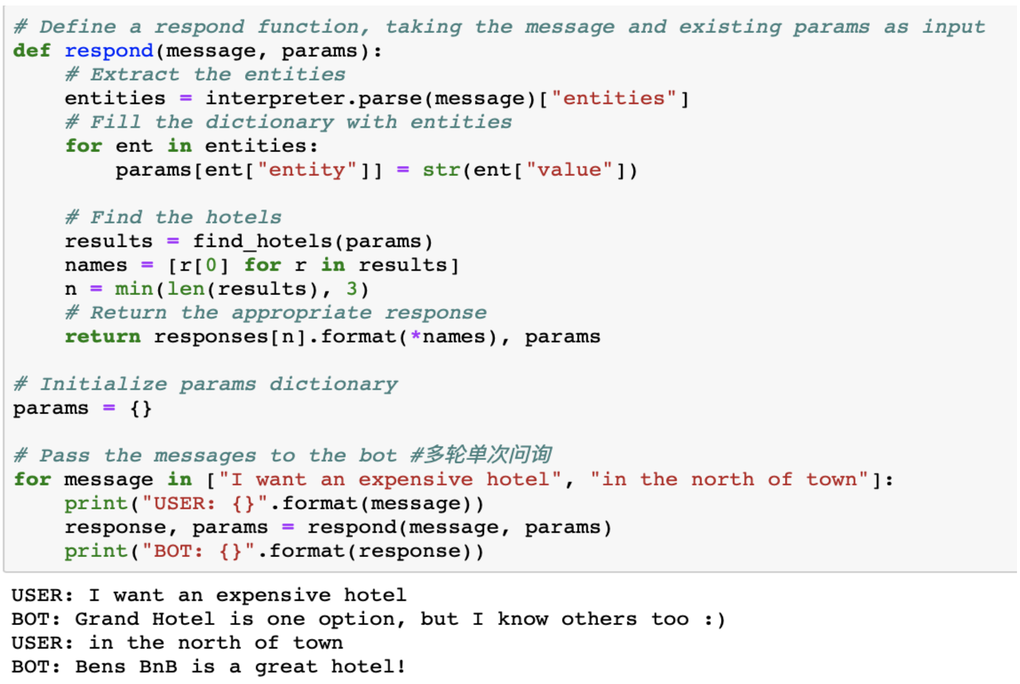
```
# Define respond()
def respond(message):
    # Extract the entities
    entities = interpreter.parse(message)["entities"]
    # Initialize an empty params dictionary
    params = {}
    # Fill the dictionary with entities
    for ent in entities:
        params[ent["entity"]] = str(ent["value"])

    # Find hotels that match the dictionary
    results = find_hotels(params)
    # Get the names of the hotels and index of the response
    names = [r[0] for r in results]
    n = min(len(results),3)
    # Select the nth element of the responses array
    return responses[n].format(*names)
```

(b)

Fig. 6: Using the natural language to explore the database.

Besides, the process of searching involves incremental slot filling and negation. The incremental slot filling refines our search. There are not only message but also params as parameters in the function respond( ). Therefore, after the first querying, the second querying could search for data based on the params from the first querying, implementing multiple rounds of a single inquiry. The code is shown below in Fig. 7. From the example in Fig. 7, the second message is based on the results of the first round of an inquiry. So, the final query result is an expensive hotel that is in the north of town.

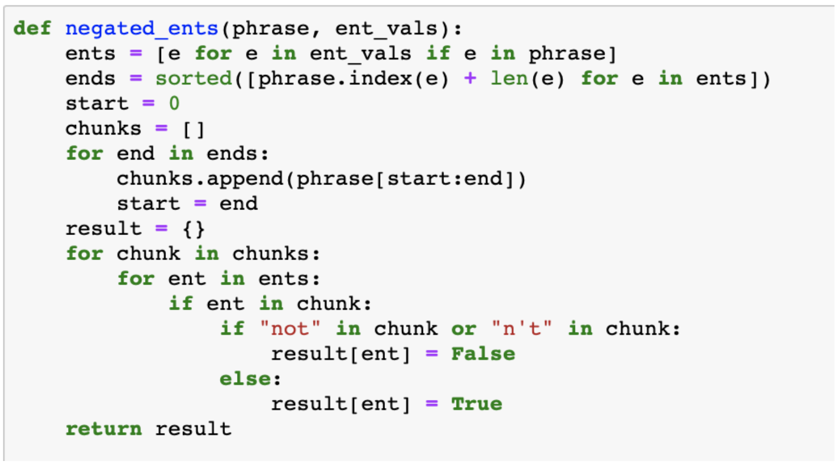
In regards to negated entities, we separate a sentence into several chunks and judge whether there is “not” or “n’t” in the chunk of an entity, determining the property of an entity. And then we respectively put affirmed parameters and negated parameters into params and neg_params to query corresponding data. The code is shown below:



```
# Define a respond function, taking the message and existing params as input
def respond(message, params):
    # Extract the entities
    entities = interpreter.parse(message)["entities"]
    # Fill the dictionary with entities
    for ent in entities:
        params[ent["entity"]] = str(ent["value"])

    # Find the hotels
    results = find_hotels(params)
    names = [r[0] for r in results]
    n = min(len(results), 3)
    # Return the appropriate response
    return responses[n].format(*names), params

# Initialize params dictionary
params = {}

# Pass the messages to the bot #多轮单次问询
for message in ["I want an expensive hotel", "in the north of town"]:
    print("USER: {}".format(message))
    response, params = respond(message, params)
    print("BOT: {}".format(response))
```

```
USER: I want an expensive hotel
BOT: Grand Hotel is one option, but I know others too :)
USER: in the north of town
BOT: Bens BnB is a great hotel!
```

Fig. 7: Incremental slot filling.



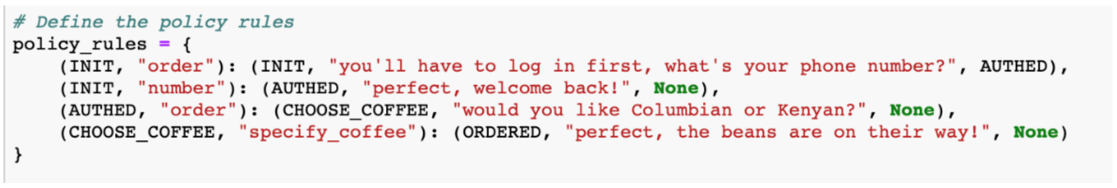
```
def negated_ents(phrase, ent_vals):
    ents = [e for e in ent_vals if e in phrase]
    ends = sorted([phrase.index(e) + len(e) for e in ents])
    start = 0
    chunks = []
    for end in ends:
        chunks.append(phrase[start:end])
        start = end
    result = {}
    for chunk in chunks:
        for ent in ents:
            if ent in chunk:
                if "not" in chunk or "n't" in chunk:
                    result[ent] = False
                else:
                    result[ent] = True
    return result
```

Fig. 8: Identifying negated entities.

2.4 State machine

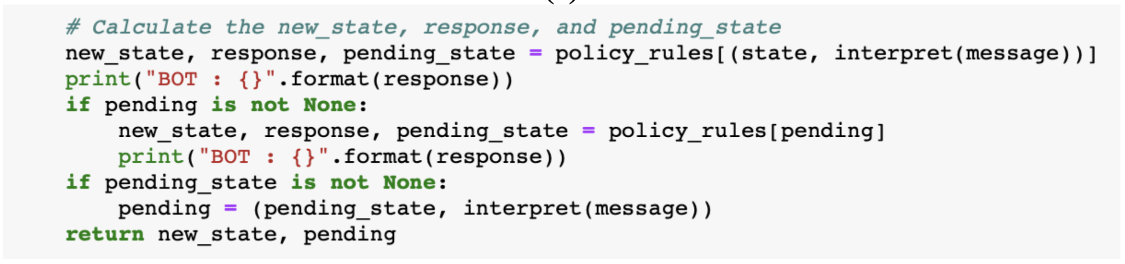
The sate machine enables chatbots to have state-jump and realize multiple rounds of multiple inquiries. To begin with, we offer policy_rules (Fig. 9a) that shows the examples of state-jump. If a state is INIT and order operation is performed, the program enters the INIT state and returns the sentence “you'll have to log in first, what's your phone number?”. And all the subsequent operations follow this policy_rules. Besides the fundamental states, the state machine includes pending states.

According to the state and the interpreting of a message in the policy_rules, the new_state, pending_state and response would be returned and pending state transition would exist (Fig. 9b).



```
# Define the policy rules
policy_rules = {
    (INIT, "order"): (INIT, "you'll have to log in first, what's your phone number?", AUTHED),
    (INIT, "number"): (AUTHED, "perfect, welcome back!", None),
    (AUTHED, "order"): (CHOOSE_COFFEE, "would you like Columbian or Kenyan?", None),
    (CHOOSE_COFFEE, "specify_coffee"): (ORDERED, "perfect, the beans are on their way!", None)
}
```

(a)

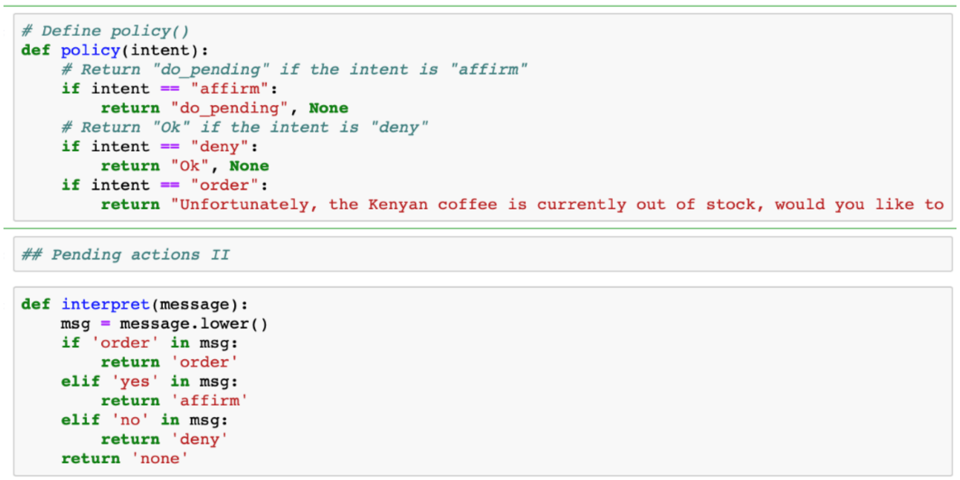


```
    # Calculate the new_state, response, and pending_state
    new_state, response, pending_state = policy_rules[(state, interpret(message))]
    print("BOT : {}".format(response))
    if pending is not None:
        new_state, response, pending_state = policy_rules[pending]
        print("BOT : {}".format(response))
    if pending_state is not None:
        pending = (pending_state, interpret(message))
    return new_state, pending
```

(b)

Fig. 9: Using the state machine.

In the multiple rounds of multiple inquiries, the negation can be integrated in the state machine. For one thing, we can use interpreter model to determine the negated message and get the name of the negative intent. And then the program puts the changed query results into the lists. For another, a function interpret( ) is defined to judge if "no" in a message and return the state "deny". And then with this negation state and the defined function policy( ), the program conducts the corresponding operation. The codes are illustrated in Fig. 10.



```
# Define policy()
def policy(intent):
    # Return "do_pending" if the intent is "affirm"
    if intent == "affirm":
        return "do_pending", None
    # Return "Ok" if the intent is "deny"
    if intent == "deny":
        return "Ok", None
    if intent == "order":
        return "Unfortunately, the Kenyan coffee is currently out of stock, would you like to
```

```
## Pending actions II
```

```
def interpret(message):
    msg = message.lower()
    if 'order' in msg:
        return 'order'
    elif 'yes' in msg:
        return 'affirm'
    elif 'no' in msg:
        return 'deny'
    return 'none'
```

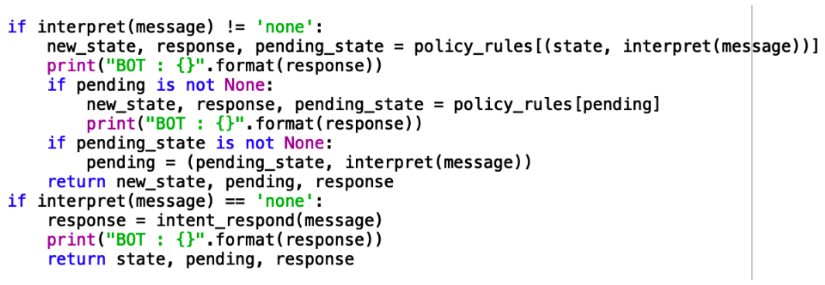
Fig. 10: The negation in the state machine.

3 Chatbot project

In practical operations, we design a chatbot to help users search for stock information. And this chatbot is deployed to the mobile chat app WeChat.

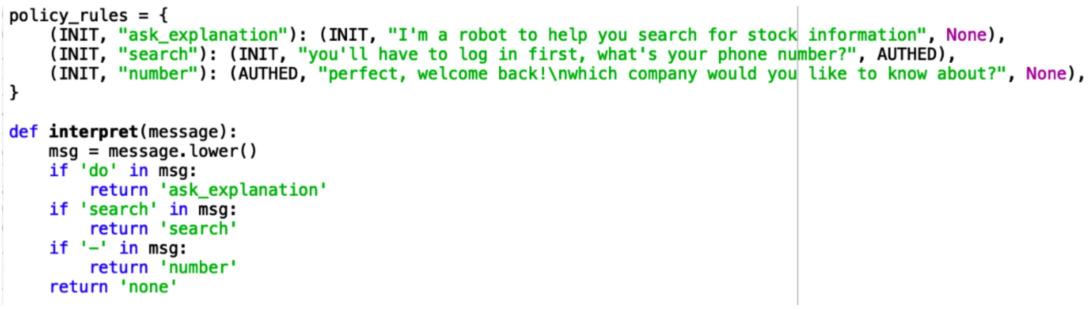
3.1 Code analysis

We classify responses of messages into two conditions through the function interpret(message) (Fig. 11). If the returned result is not none, the state machine is applied to process the start of dialogues first of all. As depicted in Fig. 12, when there is a word “do” in a message, the “ask_explanation” operation would be conducted. So, the chatbot reply a sentence “I'm a robot to help you search for stock information” to users. In accordance with policy_rules, the initial state and pending will be INIT and None. In the state machine, the subsequent operations follow the policy_rules.



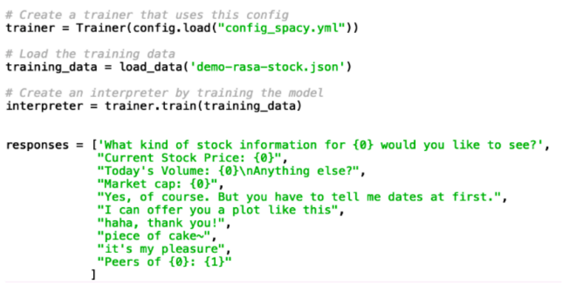
```
if interpret(message) != 'none':
    new_state, response, pending_state = policy_rules[(state, interpret(message))]
    print("BOT : {}".format(response))
    if pending is not None:
        new_state, response, pending_state = policy_rules[pending]
        print("BOT : {}".format(response))
    if pending_state is not None:
        pending = (pending_state, interpret(message))
    return new_state, pending, response
if interpret(message) == 'none':
    response = intent_respond(message)
    print("BOT : {}".format(response))
    return state, pending, response
```

Fig. 11: Handling messages according to conditions.

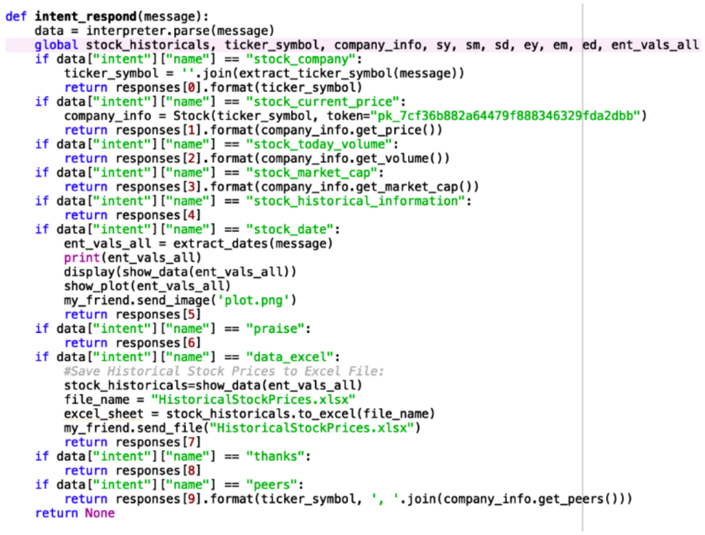


```
policy_rules = {
    (INIT, "ask_explanation"): (INIT, "I'm a robot to help you search for stock information", None),
    (INIT, "search"): (INIT, "you'll have to log in first, what's your phone number?", AUTHED),
    (INIT, "number"): (AUTHED, "perfect, welcome back!\nwhich company would you like to know about?", None),
}

def interpret(message):
    msg = message.lower()
    if 'do' in msg:
        return 'ask_explanation'
    if 'search' in msg:
        return 'search'
    if '-' in msg:
        return 'number'
    return 'none'
```

Fig. 12: The state machine in our project.



```
# Create a trainer that uses this config
trainer = Trainer(config.load("config_spacy.yml"))

# Load the training data
training_data = load_data('demo-rasa-stock.json')

# Create an interpreter by training the model
interpreter = trainer.train(training_data)


responses = ['What kind of stock information for {0} would you like to see?',
             "Current Stock Price: {0}",
             "Today's Volume: {0}\nAnything else?",
             "Market cap: {0}",
             "Yes, of course. But you have to tell me dates at first.",
             "I can offer you a plot like this",
             "haha, thank you!",
             "piece of cake~",
             "it's my pleasure",
             "Peers of {0}: {1}"
            ]
```

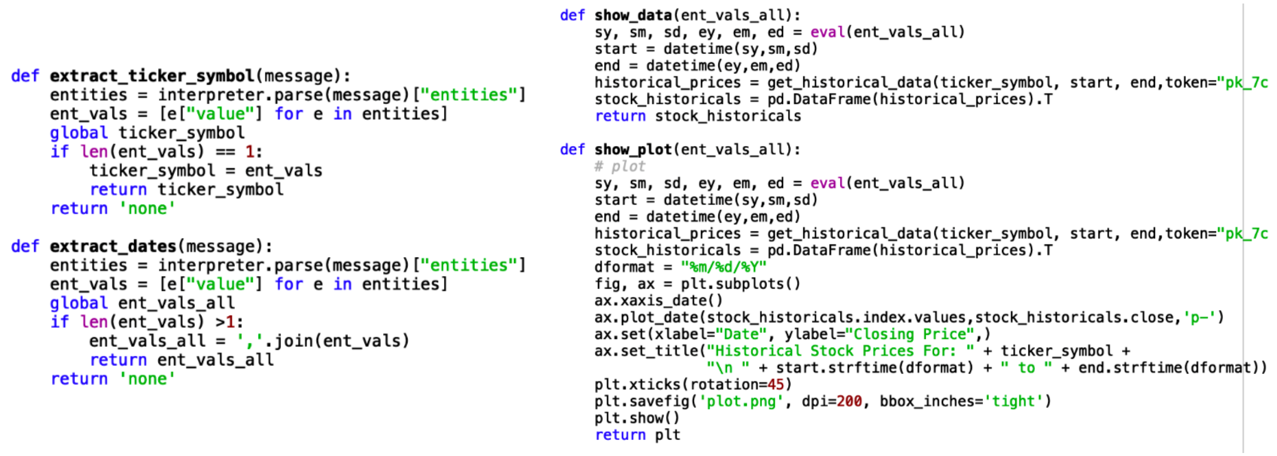


```
def intent_respond(message):
    data = interpreter.parse(message)
    global stock_historicals, ticker_symbol, company_info, sy, sm, sd, ey, em, ed, ent_vals_all
    if data["intent"]["name"] == "stock_company":
        ticker_symbol = ''.join(extract_ticker_symbol(message))
        return responses[0].format(ticker_symbol)
    if data["intent"]["name"] == "stock_current_price":
        company_info = Stock(ticker_symbol, token="pk_7cf36b882a64479f888346329fda2dbb")
        return responses[1].format(company_info.get_price())
    if data["intent"]["name"] == "stock_today_volume":
        return responses[2].format(company_info.get_volume())
    if data["intent"]["name"] == "stock_market_cap":
        return responses[3].format(company_info.get_market_cap())
    if data["intent"]["name"] == "stock_historical_information":
        return responses[4]
    if data["intent"]["name"] == "stock_date":
        ent_vals_all = extract_dates(message)
        print(ent_vals_all)
        display(show_data(ent_vals_all))
        show_plot(ent_vals_all)
        my_friend.send_image('plot.png')
        return responses[5]
    if data["intent"]["name"] == "praise":
        return responses[6]
    if data["intent"]["name"] == "data_excel":
        #Save Historical Stock Prices to Excel File:
        stock_historicals=show_data(ent_vals_all)
        file_name = "HistoricalStockPrices.xlsx"
        excel_sheet = stock_historicals.to_excel(file_name)
        my_friend.send_file("HistoricalStockPrices.xlsx")
        return responses[7]
    if data["intent"]["name"] == "thanks":
        return responses[8]
    if data["intent"]["name"] == "peers":
        return responses[9].format(ticker_symbol, ', '.join(company_info.get_peers()))
    return None
```

(a) (b)

Fig. 13: Responding to messages according to intents.

After the process of state-jump, responses of dialogues depend on the function intent_respond(message). In this part, we take advantage of Rasa NLU, training an interpreter model with a configuration file and training data (Fig. 13a). By means of the interpreter model, the name of intents is gained and then we aim to the different intent to return its response that is included in the responses array (Fig. 13b). During the querying, users are able to attain the current price, today's volume, today's volume and historical data of one kind of stock. Each information derives from the function get_price( ), get_volume( ), get_market_cap( ) and get_historical_data( ) by iexfinance API. Moreover, we define the functions extract_ticker_symbol(message) and extract_dates(message) applied to extract a ticker symbol and dates from messages (Fig. 14a). The ticker symbol is necessary for all the stock information searching. And the dates that combine starting dates and closing dates are utilized for historical data searching. Furthermore, to exhibit the historical data in various ways, functions show_data( ) and show_plot( ) are proposed (Fig. 14b). The function show_data( ) exploits DataFrame( ) of the pandas library to place historical data into a two-dimensional table. At the same time, we save these data to the local file system in the form of an excel sheet. When users need the excel sheet, the file would be sent to users. For the function show_plot( ), it draws the line chart of closing prices of a certain sort of stock from the starting date to the closing date and is displayed to users as well. Eventually, making use of wxpy library, the function reply_my_friend(msg) enables that a chatbot could be deployed on WeChat and reply messages to a friend.



```python
def extract_ticker_symbol(message):
    entities = interpreter.parse(message)["entities"]
    ent_vals = [e["value"] for e in entities]
    global ticker_symbol
    if len(ent_vals) == 1:
        ticker_symbol = ent_vals
        return ticker_symbol
    return 'none'

def extract_dates(message):
    entities = interpreter.parse(message)["entities"]
    ent_vals = [e["value"] for e in entities]
    global ent_vals_all
    if len(ent_vals) >1:
        ent_vals_all = ','.join(ent_vals)
        return ent_vals_all
    return 'none'
```

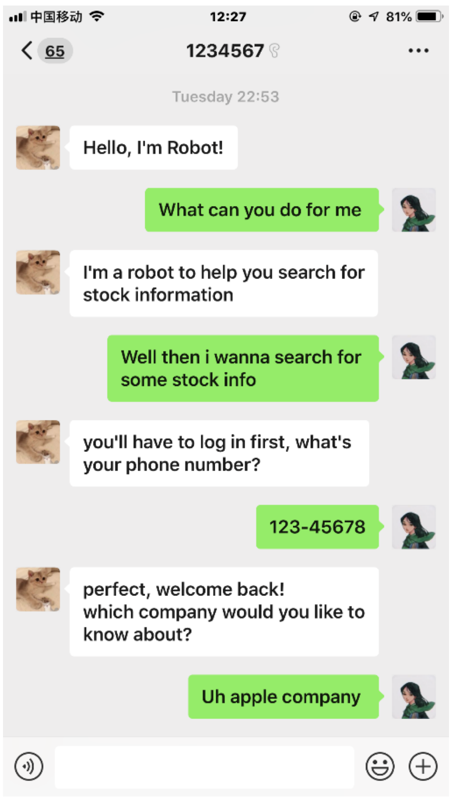
```python
def show_data(ent_vals_all):
    sy, sm, sd, ey, em, ed = eval(ent_vals_all)
    start = datetime(sy,sm,sd)
    end = datetime(ey,em,ed)
    historical_prices = get_historical_data(ticker_symbol, start, end,token="pk_7c
    stock_historicals = pd.DataFrame(historical_prices).T
    return stock_historicals

def show_plot(ent_vals_all):
    # plot
    sy, sm, sd, ey, em, ed = eval(ent_vals_all)
    start = datetime(sy,sm,sd)
    end = datetime(ey,em,ed)
    historical_prices = get_historical_data(ticker_symbol, start, end,token="pk_7c
    stock_historicals = pd.DataFrame(historical_prices).T
    dformat = "%m/%d/%Y"
    fig, ax = plt.subplots()
    ax.xaxis_date()
    ax.plot_date(stock_historicals.index.values,stock_historicals.close,'p-')
    ax.set(xlabel="Date", ylabel="Closing Price",)
    ax.set_title("Historical Stock Prices For: " + ticker_symbol +
                 "\n " + start.strftime(dformat) + " to " + end.strftime(dformat))
    plt.xticks(rotation=45)
    plt.savefig('plot.png', dpi=200, bbox_inches='tight')
    plt.show()
    return plt
```

(a) (b)

Fig. 14: Extracting entities and drawing a plot of searched historical data.

3.2 Project result

The project is about a chatbot that enables users to search for stocks. In the first place, the chatbot greets a WeChat friend, and then the user asks about the function of the chatbot. Before searching for some information about stocks, the user has to send the phone number to log in. After that, the chatbot confirms a certain stock, AAPL, which the user wants to know about (Fig. 15a). As the chatbot is able to obtain real-time stock information through the iexfiance API, the user gains the current price, volume and market cap of AAPL (Fig. 5b). In addition, the historical data could be queried when the user tells the chatbot starting dates and closing dates. The chatbot sends the historical data in two forms: a ploy and an excel file (Fig. 5c). The user can query the peers of Apple, and then starts to search for the information about another stock (Fig. 5d).



中国移动 12:27 81%

65 1234567

Tuesday 22:53

Hello, I'm Robot!

What can you do for me

I'm a robot to help you search for stock information

Well then i wanna search for some stock info

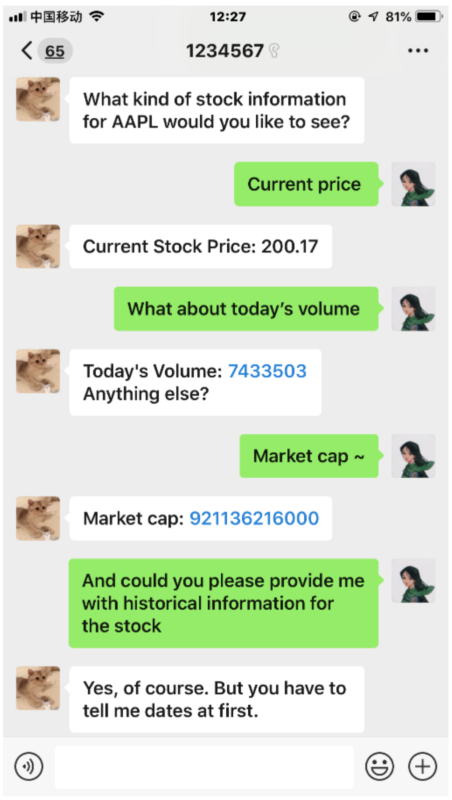
you'll have to log in first, what's your phone number?

123-45678

perfect, welcome back!
which company would you like to know about?

Uh apple company

(a)



中国移动 12:27 81%

65 1234567

What kind of stock information for AAPL would you like to see?

Current price

Current Stock Price: 200.17

What about today's volume

Today's Volume: 7433503
Anything else?

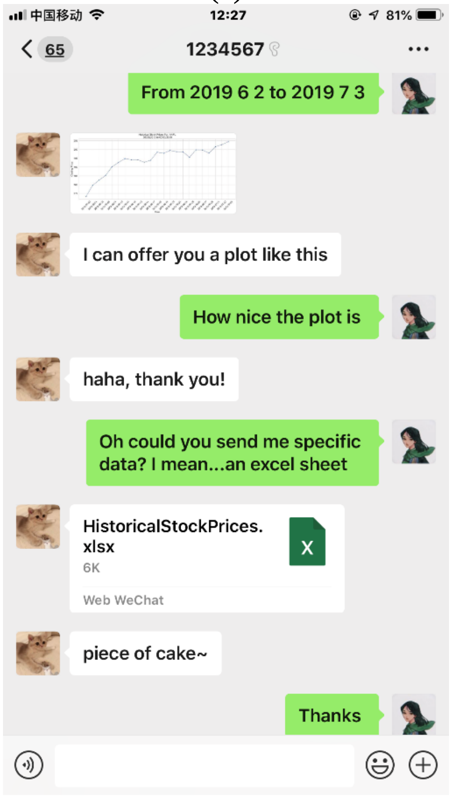
Market cap ~

Market cap: 921136216000

And could you please provide me with historical information for the stock

Yes, of course. But you have to tell me dates at first.

(b)



中国移动 12:27 81%

65 1234567

From 2019 6 2 to 2019 7 3

I can offer you a plot like this

How nice the plot is

haha, thank you!

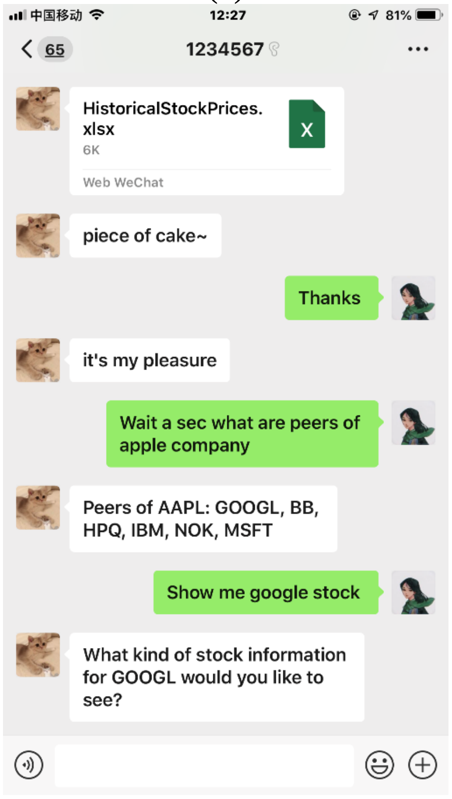
Oh could you send me specific data? I mean...an excel sheet

HistoricalStockPrices.xlsx
6K
Web WeChat

piece of cake~

Thanks

(c)



中国移动 12:27 81%

65 1234567

HistoricalStockPrices.xlsx
6K
Web WeChat

piece of cake~

Thanks

it's my pleasure

Wait a sec what are peers of apple company

Peers of AAPL: GOOGL, BB, HPQ, IBM, NOK, MSFT

Show me google stock

What kind of stock information for GOOGL would you like to see?

(d)

Fig. 15: Dialogues between a user and a chatbot.

4 Conclusion

During the learning process of intelligent chatbots, I not only acquire a great deal of basic knowledge about chatbots, but also try to create an intelligent chatbot on the basis of four-course study. As to techniques, we acquire providing diverse responses to the same question, matching and extracting intents and entities on regular expression, recognizing intents and extracting entities on machine learning method, utilizing Rasa NLU, building multiple rounds of an inquiry, creating multiple rounds of multiple inquiries with the state machine, determining negated entities and so on. In the final project, the state machine, Rasa NLU, intents recognition and entities extraction are mainly applied. What’s more, it is the first time for me to attempt to apply machine learning and natural language processing in daily life. Meanwhile, I experience convenience and commercial value that a chatbot brings about. Personally, a well-designed chatbot is very conducive to people. As chatbots can be deployed on the chat apps that most of people use almost every day, users directly touch the chatbots via chat apps. Also, chatbots include a wide range of domains, therefore, combined with API, they are very likely to meet various needs.

In the future, the training data are supposed to be improved and suitable for more questions. And more techniques that humanize chatbots will be successfully incorporated into the programs. In addition to stock searching, a great many of fields require chatbots to efficiently solve humans’ inquiries and problems in daily life.