Building a chatbot

September 20, 2020

1 Implementing Chatbots Fundamentals

1.0.1 Mohamad Yassin

09/20/2020

1.1 Introduction

This notebook introduces the fundamental techniques for building coversational chatbots. It can serve as a reference for building chatbot applications or models. The content will be split into three steps:

1. An Introduction into functions and techniques, including: respond() function, responses dictionary, random responses, response rules, users templates, regular expressions patterns, replacing pronous, and defining intents.
2. A quick overview of SpaCy's English model to create word/sentence vectors, calculate semantic cosine similarity, extract entities, and dependacny parsing. Introduction into predective machine learning with scikit-learn.
3. A demonstration of: training an interpreter to extract intents and entities by using rasaNLU's predefined spacy-sklean pipeline with a set of training data, connecting the chatbot to a SQL database to take actions, running SQL queries with Python, and capturing negations in a sentence.
4. An Implementation of a state-machine chatbot that can order coffee and chit-chat with the users.

The sample datasets and database can be found on my github: https://github.com/mohamadyassin

First things first...

[1]:
```
# Import what's neccessary
import re
import spacy
import numpy as np
import pandas as pd
import random
import json
import sqlite3
import string
import warnings
```

```
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

from rasa_nlu.config import RasaNLUModelConfig, DEFAULT_CONFIG
from rasa_nlu.model import Trainer, TrainingData
from rasa_nlu.model import Interpreter
from rasa_nlu import load_data
```

1.2 Step 1: Introductory Functions

Let's begin by defining user templates, and the functions respond() and send_message(). This bot can repeat the user's message

[96]:
```
# Create templates
bot_template = "BOT : {0}"
user_template = "USER : {0}"


# Define a function that responds to a user's message: respond
def respond(message):
    # Concatenate the user's message to the end of a standard bot respone
    bot_message = "I can hear you! You said: " + message
    # Return the result
    return bot_message


# Define a function that sends a message to the bot: send_message
def send_message(message):
    # Print user_template including the user_message
    print(user_template.format(message))
    # Get the bot's response to the message
    response = respond(message)
    # Print the bot template including the bot's response.
    print(bot_template.format(response))

# Send a message to the bot
send_message("hello")
```

```
USER : hello
BOT : I can hear you! You said: hello
```

Here we create a dictionary of responses with two variables in order to define a respond function

[101]:
```
# Define variables
name = "Greg"
```

```
weather = "cloudy"

# Define a dictionary with the predefined responses
responses = {
  "what's your name?": "my name is {0}".format(name),
  "what's today's weather?": "the weather is {0}".format(weather),
  "default": "default message"
}

# Return the matching response if there is one, default otherwise
def respond(message):
    # Check if the message is in the responses
    if message in responses:
        # Return the matching message
        bot_message = responses[message]
    else:
        # Return the "default" message
        bot_message = responses["default"]
    return bot_message

# Test our function
print(respond("what's today's weather?"))
print(respond("what's your name?"))
```

```
the weather is cloudy
my name is Greg
```

Usually users interact with the bot on a recurrent basis. To create a better personality to our bot, we can define mutiple answers to each response

[110]:
```
# Define variables
name = "Greg"
weather = "sunny"

# Define a dictionary containing a list of responses for each message
responses = {
  "what's your name?": [
      "my name is {0}".format(name),
      "they call me {0}".format(name),
      "I go by {0}".format(name)
   ],
  "what's today's weather?": [
      "the weather is {0}".format(weather),
      "it's {0} today".format(weather)
    ],
  "default": ["default message"]
}
```

```
# Use random.choice() to choose a matching response
def respond(message):
    # Check if the message is in the responses
    if message in responses:
        # Return a random matching response
        bot_message = random.choice(responses[message])
    else:
        # Return a random "default" response
        bot_message = random.choice(responses["default"])
    return bot_message

# Test our function
print(respond("what's today's weather?"))
print(respond("what's today's weather?"))
print(respond("what's today's weather?"))
print(respond("what's your name?"))
print(respond("what's your name?"))
print(respond("what's your name?"))
```

```
the weather is sunny
it's sunny today
it's sunny today
they call me Greg
my name is Greg
I go by Greg
```

Define a responses dictionary then choose a random answer

[115]:
```
# Define responses
responses = {'question': ["I don't know :(", 'you tell me!'],
 'statement': ['tell me more!',
  'why do you think that?',
  'how long have you felt this way?',
  'I find that extremely interesting',
  'can you back that up?',
  'oh wow!',
  ':)']}

# Classify as question if it ends wih '?'
def respond(message):
    # Check for a question mark
    if message.endswith('?'):
        # Return a random question
        return random.choice(responses["question"])
    # Return a random statement
    return random.choice(responses["statement"])

# Test our function
```

```
print(respond("how can I find the bathroom?"))
print(respond("this is a really cool bot"))
```

```
I don't know :(
oh wow!
```

Let's take our chit-cat bot one step further with Regular expressions (regex). Regex are a powerful tool that allows us to search pattens in a text. It also permits high degree of flexibility when defining repsponses

[116]:
```
# Define rules of patterns and responses lists
rules = {
    #1
    'I want (.*)':     #the regex '.*' returns the rest of sentence
         ['What would it mean if you got {0}',
  'Why do you want {0}',
  "What's stopping you from getting {0}"],
    #2
 'do you remember (.*)':
         ['Did you think I would forget {0}',
  "Why haven't you been able to forget {0}",
  'What about {0}',
  'Yes .. and?'],
    #3
 'do you think (.*)':
         ['if {0}? Absolutely.', 'No chance'],
    #4
 'if (.*)':
         ["Do you really think it's likely that {0}",
  'Do you wish that {0}',
  'What do you think about {0}',
  'Really--if {0}']
}

# Define match_rule()
def match_rule(rules, message):
    response, phrase = "default", None

    # Iterate over the rules dictionary
    for pattern, responses in rules.items():
        # Create a match object
        match = re.search(pattern, message)
        if match is not None:
            # Choose a random response
            response = random.choice(responses)
            if '{0}' in response:
                phrase = match.group(1)
    # Return the response and phrase
```

```
    return response.format(replace_pronouns(phrase))

# Test match_rule
print(match_rule(rules, "do you remember your last birthday"))
print(match_rule(rules, "do you know if I can eat apples"))
print(match_rule(rules, "do you wish that the weather is sunny year-round?"))
```

```
Did you think I would forget your last birthday
Really--if I can eat apples
default
```

There's something funny about our bot's responses. It returned the wrong prounouns. Let's see if we replace pronouns

[146]:
```
# Define replace_pronouns()
def replace_pronouns(message):

    message = message.lower()
    if 'me' in message:
        # Replace 'me' with 'you'
        return re.sub('me','you', message)
    if 'my' in message:
        # Replace 'my' with 'your'
        return re.sub('my','your', message)
    if 'your' in message:
        # Replace 'your' with 'my'
        return re.sub('your','my', message)
    if 'you' in message:
        # Replace 'you' with 'me'
        return re.sub('you','me',message)

    return message


# Define match_rule()
def match_rule(rules, message):
    response, phrase = "default", None


    # Iterate over the rules dictionary
    for pattern, responses in rules.items():
        # Create a match object
        match = re.search(pattern, message)
        if match is not None:
            # Choose a random response
            response = random.choice(responses)
            if '{0}' in response:
                phrase = match.group(1)
```

```
    # Return the response and phrase
    return replace_pronouns(response.format(phrase))

# Test match_rule
print(match_rule(rules, "do you remember your last birthday"))
print(match_rule(rules, "do you know if she likes me?"))
print(match_rule(rules, "do you wish that the weather is sunny year-round?"))
```

```
what about my last birthday
do you really think it's likely that she likes you?
default
```

Let's use regex to define a function that finds names in a sentence. We will use keywords and patterns

[149]:
```
# Define find_name()
def find_name(message):
    name = None
    # Create a pattern for checking if the keywords occur
    name_keyword = re.compile(r"name|call")
    # Create a pattern for finding capitalized words
    name_pattern = re.compile(r"[A-Z]{1}[a-z]*")
    if name_keyword.search(message):
        # Get the matching words in the string
        name_words = name_pattern.findall(message)
        if len(name_words) > 0:
            # Return the name if the keywords are present
            name = ' '.join(name_words)
    return name

# Define respond()
def respond(message):
    # Find the name
    name = find_name(message)
    if name is None:
        return "Hi there!"
    else:
        return "Hello, {0}!".format(name)


# Define a function that sends a user's message
def send_message(message):
    print(user_template.format(message))
    response = respond(message)
    print(bot_template.format(response))


# Send messages
```

```
send_message("my name is David Copperfield")
send_message("call me Mohamad")
send_message("What's going on?")
```

```
USER : my name is David Copperfield
BOT : Hello, David Copperfield!
USER : call me Mohamad
BOT : Hello, Mohamad!
USER : What's going on?
BOT : Hi there!
```

In order for our bot to take real-world actions, we have to define intents. Let's take a quick look at a function that will allow us to find search intent in a message with regex. We will first define patterns and responses. Then we will define functions that can match the intent then respond to the user

[150]:
```
# Define variations of the same intent
intents = {'greet': ['hello', 'hi', 'hey'],
           'goodbye': ['bye', 'farewell'],
           'thankyou': ['thank', 'thx']}

#Define responses to intents
responses = {'default': 'default message',
 'goodbye': 'goodbye for now',
 'greet': 'Hello you! :)',
 'thankyou': 'you are very welcome'}

# Instantiate an empty dictionary
patterns = {}


# Iterate over the keywords dictionary
for intent, keys in intents.items():
    # Create regular expressions and compile them into pattern objects
    patterns[intent] = re.compile('|'.join(keys))


# Now we have our patterns with variations ready for regex
print(patterns)
```

```
{'greet': re.compile('hello|hi|hey'), 'goodbye': re.compile('bye|farewell'),
'thankyou': re.compile('thank|thx')}
```

[151]:
```
# Define a function to find the intent of a message
def match_intent(message):
    matched_intent = None
    # Iterate over items in patterns dicitionary
    for intent, pattern in patterns.items():
```

```
        # Check if the pattern occurs in the message
        if pattern.search(message):
            matched_intent = intent
    return matched_intent

# Define a respond function
def respond(message):
    # Call the match_intent function
    intent = match_intent(message)
    # Fall back to the default response
    key = "default"
    if intent in responses:
        key = intent
    return responses[key]


# Send messages
send_message("hello!")
send_message("bye byeee")
send_message("thanks very much!")
```

```
USER : hello!
BOT : Hello you! :)
USER : bye byeee
BOT : goodbye for now
USER : thanks very much!
BOT : you are very welcome
```

1.3 Step 2: SpaCy and Scikit-learn

Let's begin by importing SpaCy's English model. We use the meduim-size model that has to be manually downloaded into our environment. Similarity is calculated using the cosine function. So we are measuring the degree of similiarty between tokens or ducements. A similarity of 1 means that both are in the same direction. 0 is perpendicular. And -1 is opposite directions.

[52]:
```
# This will load the default en model
nlp = spacy.load('en_core_web_md')
```

[22]:
```
# Test semantic similarity
doc = nlp('cat')
print("similarity betwen cat and can: ", doc.similarity(nlp('can')))
print("similarity betwen cat and dog: ", doc.similarity(nlp('dog')))
print("similarity betwen cat and monkey: ", doc.similarity(nlp('monkey')))
```

```
similarity betwen cat and can:  0.30165289379772614
similarity betwen cat and dog:  0.8016854705531046
similarity betwen cat and monkey:  0.5351812775125145
```

In this demonstration, we use the medium-size model which contains 300 unique tokens for each vector

[46]:
```
#Check vectors length
nlp.vocab.vectors_length
```

[46]:
```
300
```

Import the dataset which consists of sentences and their respective intents in order train a model on predicting intetns

[68]:
```
# Read atis intent as a dataframe
df = pd.read_csv('atis/atis_intents.csv')

# Inspect the dataframe
print("number of samples: {}".format(len(df)))
print(df.head())
```

```
number of samples: 4977
        atis_flight  \
0       atis_flight
1  atis_flight_time
2      atis_airfare
3      atis_airfare
4       atis_flight

   i want to fly from boston at 838 am and arrive in denver at 1110 in the
morning
0   what flights are available from pittsburgh to…
1   what is the arrival time in san francisco for…
2            cheapest airfare from tacoma to orlando
3   round trip fares from pittsburgh to philadelp…
4   i need a flight tomorrow from columbus to min…
```

[82]:
```
# Identify the number of unique labels
print("number of unique labels: ", len(df.iloc[:,0].unique()))
```

```
number of unique labels:  22
```

[81]:
```
# Take a closer look at the intent labels
df.iloc[:,0].value_counts()/len(df)
```

[81]:
```
atis_flight                 0.736387
atis_airfare                0.084991
atis_ground_service         0.051236
atis_airline                0.031545
atis_abbreviation           0.029536
atis_aircraft               0.016275
```

```
atis_flight_time                              0.010850
atis_quantity                                 0.010247
atis_flight#atis_airfare                      0.004219
atis_distance                                 0.004018
atis_airport                                  0.004018
atis_city                                     0.003818
atis_ground_fare                              0.003617
atis_capacity                                 0.003215
atis_flight_no                                0.002411
atis_restriction                              0.001206
atis_meal                                     0.001206
atis_airline#atis_flight_no                   0.000402
atis_airfare#atis_flight_time                 0.000201
atis_ground_service#atis_ground_fare          0.000201
atis_cheapest                                 0.000201
atis_aircraft#atis_flight#atis_flight_no      0.000201
Name: atis_flight, dtype: float64
```

Since the first 5 labels represent more than 93% of the samples, we will drop the bottom 18 labels because they will not contribute to our classification model. This way our classifier has a better chance to predict the correct labels. This is an important note to take when choosing training data for real-life models.

We can better transform our data by dealing with integer labels so we will use sklearn's label encoder to get the job done

[97]:
```
# Instantiate a LabelEncoder then transform our labels column
le = LabelEncoder()
df.iloc[:,0] = le.fit_transform(df.iloc[:,0])
```

[97]:
```
0       12
1       15
2        3
3        3
4       12
        ..
4972     3
4973    12
4974     5
4975    12
4976    12
Name: atis_flight, Length: 4977, dtype: int64
```

[98]:
```
# Find out the top 5 intents we want to keep
df.iloc[:,0].value_counts()/len(df)
```

[98]:
```
12    0.736387
3     0.084991
```

```
17    0.051236
5     0.031545
0     0.029536
1     0.016275
15    0.010850
20    0.010247
13    0.004219
7     0.004018
11    0.004018
10    0.003818
16    0.003617
8     0.003215
14    0.002411
19    0.001206
21    0.001206
6     0.000402
4     0.000201
18    0.000201
2     0.000201
9     0.000201
Name: atis_flight, dtype: float64
```

In order to quickly select the top five labels, we will set the integer labels as an index then select the desired values based on the value caounts above

[144]:
```
# Set the labels as index and select samples
df = df.set_index(df.iloc[:,0]).loc[(12,3,17,5,0),:]
# Drop the index
df = df.reset_index(drop=True)
```

[145]:
```
#Inspect the dataframe
df
```

[145]:
```
      atis_flight  \
0              12
1              12
2              12
3              12
4              12
…             …
4642            0
4643            0
4644            0
4645            0
4646            0

      i want to fly from boston at 838 am and arrive in denver at 1110 in the
```

```
morning
0      what flights are available from pittsburgh to…
1      i need a flight tomorrow from columbus to min…
2      show me the flights from pittsburgh to los an…
3                  all flights from boston to washington
4      show me the flights from dallas to san francisco
…                                                    …
4642                        what does fare code qo mean
4643                                what does ewr mean
4644                               what is fare code h
4645                     what does the fare code qw mean
4646                         what does fare code qo mean

[4647 rows x 2 columns]
```

[127]:
```
# Double check for any null values
df.isnull().sum()
```

[127]:
```
atis_flight
0
 i want to fly from boston at 838 am and arrive in denver at 1110 in the morning
0
dtype: int64
```

Now we have the dataframe ready to use. We will begin with the cosine similarity function from sklearn. In order to do so, we first create word vectors for each of our sentences with SpaCy's English medium-size model. This is a simple and very easy to use method that can be useful sometimes for tasks such a recommender system

[150]:
```
# Identify features and labels
X = np.array(df.iloc[:,1])
y = np.array(df.iloc[:,0])

# Create a shape
X_train_shape = (
    len(X),
    nlp.vocab.vectors_length)

# Create zeros array
X_train_vector = np.zeros(X_train_shape)

# Loop over array with vector for each sentence
for i, sentence in enumerate(X):
    X_train_vector[i,:] = nlp(sentence).vector
```

[191]:
```
# Test message
```

```
test_mesage = "I would like to find a nonstop flight from georgia to Washingotn␣
 ↪DC"

# Create a vector from nlp
test_x = nlp(test_mesage).vector

# Calculate a list of scores with list comrehension
scores = [
    cosine_similarity(X_train_vector[i,:].reshape(-1,1), test_x.reshape(-1,1)).
 ↪mean()
    for i in range(len(X))
]
# Search the highest score and match it agaisnt its labels index
test_score = y[np.argmax(scores)]

# Use a dictionary to interpret the score
score_dict = {12: "atis_flight",
              3: "atis_airfare",
              17: "atis_ground_service",
              5: "atis_airline ",
              0: "atis_abbreviation"}

print("predicted intent:   {}".format(score_dict[test_score]))
```

```
predicted intent:   atis_flight
```

Here we use the same data to predict labeled intents by using a support vector classifier from sklearn. We begin by splitting the data then fitting the model, then we will count the number of correct prediction

[210]:
```
# Split the data into train and test samples
X_train, X_test, y_train, y_test =␣
 ↪train_test_split(X_train_vector,y,test_size=0.1,random_state=42)

# Create a support vector classifier
model = SVC(C=1, gamma='scale')

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

# Predict the labels of the test set
y_pred = model.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
```

```
print("Predicted {0} correctly out of {1} test examples".format(n_correct,␣
 ↪len(y_test)))
```

```
Predicted 441 correctly out of 465 test examples
```

1.4 Step 3: Rasa NLU

As a first step, we'll define the configuration to our model then create a trainer out of it in order to create an interpreter and train it on our data. The interpreter will parse intents and entities from our sentences. We are using restaurants data for this purpose. Then we will connect to a local sample sql database that will allow us to test our bot's ability to take actions

[4]:
```
# Create args dictionary
args = {"pipeline": "spacy_sklearn"}

# Create a configuration and trainer
config = RasaNLUModelConfig(configuration_values=args)
trainer = Trainer(config)
```

[5]:
```
warnings.simplefilter('ignore')

# Load the training data
training_data = load_data("training_data.json")

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

# Test the interpreter
print(interpreter.parse("I'm looking for a Mexican restaurant in the North of␣
 ↪town"))
```

```
Fitting 2 folds for each of 6 candidates, totalling 12 fits
{'intent': {'name': 'restaurant_search', 'confidence': 0.8342435271519987},
'entities': [{'start': 18, 'end': 25, 'value': 'mexican', 'entity': 'cuisine',
'confidence': 0.6231186599376848, 'extractor': 'CRFEntityExtractor'}, {'start':
44, 'end': 49, 'value': 'north', 'entity': 'location', 'confidence':
0.8386232722553214, 'extractor': 'CRFEntityExtractor'}], 'intent_ranking':
[{'name': 'restaurant_search', 'confidence': 0.8342435271519987}, {'name':
'greet', 'confidence': 0.04127640804383251}, {'name': 'affirm', 'confidence':
0.03619636710182238}, {'name': 'chitchat/ask_weather', 'confidence':
0.03102607470749123}, {'name': 'chitchat/ask_name', 'confidence':
0.028698000180374494}, {'name': 'goodbye', 'confidence': 0.02855962281448071}],
'text': "I'm looking for a Mexican restaurant in the North of town"}

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  12 out of  12 | elapsed:    0.1s finished
```

[6]:
```
# Extract entities using the interpreter
message = "I'm looking for a Mexican restaurant in the North of town"
data = interpreter.parse(message)    #trained model

#Save entities as key value pairs
params = {}
for ent in data["entities"]:
    params[ent['entity']] = ent['value'] #this should return price & location

print("extracted entities: {}".format(params))
```

```
extracted entities: {'cuisine': 'mexican', 'location': 'north'}
```

Now we connect our sample database in order to have our bot start taking some real world actions

[7]:
```
# Create connection
conn = sqlite3.connect("hotels.db")

# Create a cursor to access db connection
c = conn.cursor()

# Execute a query
c.execute("SELECT * FROM hotels")

# Get the results from the sursor
c.fetchall()        # Returns a list of tuples
```

[7]:
```
[('Hotel for Dogs', 'mid', 'east', 3),
 ('Hotel California', 'mid', 'north', 3),
 ('Grand Hotel', 'hi', 'south', 5),
 ('Cozy Cottage', 'lo', 'south', 2),
 ("Ben's BnB", 'hi', 'north', 4),
 ('The Grand', 'hi', 'west', 5),
 ('Central Rooms', 'mid', 'center', 3)]
```

[22]:
```
# Extract entities using the interpreter
message = "cheap hotel in the south"
data = interpreter.parse(message)    # Our trained model

# Save entities as key value pairs
params = {}
for ent in data["entities"]:
    params[ent['entity']] = ent['value'] # This should return price & location

# Here's a hard coded dict for the sake of trial
params = {'location': 'south', 'price': 'lo'}
```

Let's take a look at how we construct our queries befor we actually use them in a function

[24]:
```
# Extract filters for later
query = "SELECT name FROM hotels"
filters = ["{}=?".format(k) for k in params.keys()]  # This returns extracted␣
 ↪entities

# Define condition
conditions = " and ".join(filters)

# Define final query
final_q = " WHERE ".join([query, conditions])

# Define query params
t = tuple(params.values())

# Let's try it out!
c.execute(final_q,t)
c.fetchall()
```

[24]:
```
[('Cozy Cottage',)]
```

We can use the two steps above to build our function

[97]:
```
# 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()


# Create the dictionary of column names and values
params = {"location":"south",
    "price": "lo"}
```

```
# Find the hotels that match the parameters
print(find_hotels(params))
```

```
[('Cozy Cottage', 'lo', 'south', 2)]
```

We want our chatbot to respond to the user with the results of the query. What happens if we run a query and get more than one result, multiple results, or no results at all? We can define the responses and function in the following manner

[69]:
```
# How to return responses when we have 0 or >1 results
responses = [
    "I'm sorry :( I couldn't fins anything like that",
    "what about {}?",
    "{} is one option, but I know others too :)"
]

# Run a query
results = c.execute("SELECT name FROM hotels WHERE location='north' and␣
 ↪price='hi'").fetchall()

# Get the length
len(results)

# Define index
index = min(len(results), len(responses)-1)

# Index response
responses[index]

# Print response
print(responses[index].format(results))
```

```
what about [("Ben's BnB",)]?
```

[50]:
```
# 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)

print(respond("I want a hotel in the north of town"))
```

```
Hotel California is one option, but I know others too :)
```

Having our chatbot detect negation could be a tricky task. One way to accomplish this by capturing the enteties that are preceeded by 'not' or 'n't'. However, this techniques will fail us with other forms of negation

[64]:
```
# Capturing negations
doc = nlp('not sushi, maybe pizza?')
indicies = [1, 4]   #starting from the entity, ending at len+1

ents, negated_ents = [], []  #initialize empty lists

start = 0   #start from zero

for i in indicies:
    phrase = "{}".format(doc[start:i])
    if "not" in phrase or "n't" in phrase:
        negated_ents.append(doc[i])
    else:
        ents.append(doc[i])
    start = i

print("negated entities: {}".format(negated_ents))
print("other entities: {}".format(ents))
```

```
negated entities: [sushi]
other entities: [pizza]
```

Let's use the multiple answers techniques in a function then test it

[98]:
```
# Define responses
responses = ["I'm sorry :( I couldn't find anything like that",
 '{} is a great hotel!',
 '{} or {} would work!',
 '{} is one option, but I know others too :)']

# 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
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: Hotel for Dogs is one option, but I know others too :)
USER: in the north of town
BOT: Hotel California or Ben's BnB would work!
```

1.5 Step4: State Machine

Two main components of a state machine are: states and policy rules. We define the states according to the workflow of the app. Policy rules are defined as a dictionary of tuples of keys and values. The keys include the state and intent, and the values include the state and response. The state is selected after the bot detects an intent (with the trained interpreter) and the response is sent accordingly. Usually the state defaults to a certain value

[102]:
```
# Define the three states
INIT = 0
CHOOSE_COFFEE = 1
ORDERED = 2

# Define the policy rules
policy = {
    (INIT, "order"): (CHOOSE_COFFEE, "ok, Colombian or Kenyan?"),
    (INIT, "none"): (INIT, "I'm sorry - I'm not sure how to help you"),
    (CHOOSE_COFFEE, "specify_coffee"): (ORDERED, "perfect, the beans are on␣
 ↪their way!"),
    (CHOOSE_COFFEE, "none"): (CHOOSE_COFFEE, "I'm sorry - would you like␣
 ↪Colombian or Kenyan?"),
}

# Define a function that interprets the intent
```

```
def interpret(message):
    msg = message.lower()
    if 'order' in msg:
        return 'order'
    if 'kenyan' in msg or 'colombian' in msg:
        return 'specify_coffee'
    return 'none'

# Initial state is usually predefined
state = INIT
# Define a respond function
def respond(policy, state, message):
    (new_state, response) = policy[(state, interpret(message))]
    return new_state, response

# Define a function that sends a message
def send_message(policy, state, message):
    print("USER : {}".format(message))
    new_state, response = respond(policy, state, message)
    print("BOT : {}".format(response))
    return new_state


# Create the list of messages
messages = [ "I'd like to become a professional dancer",
    "well then I'd like to order some coffee",
    "my favourite animal is a zebra",
    "kenyan"]

# Call send_message() for each message
state = INIT
for message in messages:
    state = send_message(policy, state, message)
```

```
USER : I'd like to become a professional dancer
BOT : I'm sorry - I'm not sure how to help you
USER : well then I'd like to order some coffee
BOT : ok, Colombian or Kenyan?
USER : my favourite animal is a zebra
BOT : I'm sorry - would you like Colombian or Kenyan?
USER : kenyan
BOT : perfect, the beans are on their way!
```

Usually users ask the explainatory questions, so it can be helpful to add such an intent to our policy rules

[110]:
```
# Define the policy rules dictionary
policy_rules = {
```

```
    (INIT, "ask_explanation"): (INIT, "I'm a bot to help you order coffee␣
 ↪beans"),
    (INIT, "order"): (CHOOSE_COFFEE, "ok, Colombian or Kenyan?"),
    (CHOOSE_COFFEE, "specify_coffee"): (ORDERED, "perfect, the beans are on␣
 ↪their way!"),
    (CHOOSE_COFFEE, "ask_explanation"): (CHOOSE_COFFEE,
                                         "We have two kinds of coffee beans -␣
 ↪the Kenyan ones make a slightly sweeter coffee, and cost $6. The Brazilian␣
 ↪beans make a nutty coffee and cost $5.")
}

# Define a function that interprets the intent
def interpret(message):
    msg = message.lower()
    if 'order' in msg:
        return 'order'
    if 'kenyan' in msg or 'colombian' in msg:
        return 'specify_coffee'
    if 'what' in msg:
        return 'ask_explanation'
    return 'none'

# Initial state is usually predefined
state = INIT

# Define a respond function
def respond(state, message):
    (new_state, response) = policy_rules[(state, interpret(message))]
    return new_state, response

# Define a function that sends a message
def send_message(state, message):
    print("USER : {}".format(message))
    new_state, response = respond(state, message)
    print("BOT : {}".format(response))
    return new_state

# Define send_messages()
def send_messages(messages):
    state = INIT
    for msg in messages:
        state = send_message(state, msg)

# Send the messages
send_messages([
    "what can you do for me?",
    "well then I'd like to order some coffee",
```

```
    "what do you mean by that?",
    "kenyan"
])
```

```
USER : what can you do for me?
BOT : I'm a bot to help you order coffee beans
USER : well then I'd like to order some coffee
BOT : ok, Colombian or Kenyan?
USER : what do you mean by that?
BOT : We have two kinds of coffee beans - the Kenyan ones make a slightly
sweeter coffee, and cost $6. The Brazilian beans make a nutty coffee and cost
$5.
USER : kenyan
BOT : perfect, the beans are on their way!
```

Sometimes the workflow or process require an authentication or other state that can remain pending until the user identifies the conditions. Using a pending state can be useful for this task

[156]:
```
# Define the states
INIT=0
AUTHED=1
CHOOSE_COFFEE=2
ORDERED=3

# 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 Colombian or Kenyan?",␣
 ↪None),
    (CHOOSE_COFFEE, "specify_coffee"): (ORDERED, "perfect, the beans are on␣
 ↪their way!", None)
}

# Defind the interpret() function
def interpret(message):
    msg = message.lower()
    if 'order' in msg:
        return 'order'
    if 'kenyan' in msg or 'colombian' in msg:
        return 'specify_coffee'
    if any([d in msg for d in string.digits]):
        return 'number'
    return 'none'

# Defind the send_message() function
def send_message(state, pending, message):
```

```
    print("USER : {}".format(message))
    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

# Define send_messages()
def send_messages(messages):
    state = INIT
    pending = None
    for msg in messages:
        state, pending = send_message(state, pending, msg)

# Send the messages
send_messages(["I'd like to order some coffee",
    "555-1234",
    "kenyan"])
```

```
USER : I'd like to order some coffee
BOT : you'll have to log in first, what's your phone number?
USER : 555-1234
BOT : perfect, welcome back!
BOT : would you like Colombian or Kenyan?
USER : kenyan
BOT : perfect, the beans are on their way!
BOT : would you like Colombian or Kenyan?
```

Now we will define a bot that can chit-chat and order coffee at the same time

[173]:
```
eliza_rules = {'I want (.*)': ['What would it mean if you got {0}',
  'Why do you want {0}',
  "What's stopping you from getting {0}"],
 'do you remember (.*)': ['Did you think I would forget {0}',
  "Why haven't you been able to forget {0}",
  'What about {0}',
  'Yes .. and?'],
 'do you think (.*)': ['if {0}? Absolutely.', 'No chance'],
 'if (.*)': ["Do you really think it's likely that {0}",
  'Do you wish that {0}',
  'What do you think about {0}',
  'Really--if {0}']}

# Define a function that matches chit-chat rules
```

```
def match_rule(rules, message):
    for pattern, responses in rules.items():
        match = re.search(pattern, message)
        if match is not None:
            response = random.choice(responses)
            var = match.group(1) if '{0}' in response else None
            return response, var
    return "default", None

# Define a function to replace pronouns
def replace_pronouns(message):
    message = message.lower()
    if 'me' in message:
        return re.sub('me', 'you', message)
    if 'i' in message:
        return re.sub('i', 'you', message)
    elif 'my' in message:
        return re.sub('my', 'your', message)
    elif 'your' in message:
        return re.sub('your', 'my', message)
    elif 'you' in message:
        return re.sub('you', 'me', message)
    return message

# Defind the interpret() function
def interpret(message):
    msg = message.lower()
    if 'order' in msg:
        return 'order'
    if 'kenyan' in msg or 'colombian' in msg:
        return 'specify_coffee'
    if any([d in msg for d in string.digits]):
            return 'number'
    return 'none'

# Define chitchat_response()
def chitchat_response(message):
    # Call match_rule()
    response, var = match_rule(eliza_rules, message)
    # Return none if response is "default"
    if response == "default":
        return None
    if '{0}' in response:
        # Replace the pronouns of phrase
        var = replace_pronouns(var)
        # Calculate the response
        response = response.format(var)
```

```
    return response

# Define send_message()
def send_message(state, pending, message):
    print("USER : {}".format(message))
    response = chitchat_response(message)
    if response is not None:
        print("BOT : {}".format(response))
        return state, None

    # 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

# Define send_messages()
def send_messages(messages):
    state = INIT
    pending = None
    for msg in messages:
        state, pending = send_message(state, pending, msg)

# Send the messages
send_messages([
    "I'd like to order some coffee",
    "555-12345",
    "do you remember when I ordered 1000 kilos by accident?",
    "kenyan"
])
```

```
USER : I'd like to order some coffee
BOT : you'll have to log in first, what's your phone number?
USER : 555-12345
BOT : perfect, welcome back!
BOT : would you like Colombian or Kenyan?
USER : do you remember when I ordered 1000 kilos by accident?
BOT : Yes .. and?
USER : kenyan
BOT : perfect, the beans are on their way!
```