The Stock Chatbot Project Report

Haozhan Sun

Australian National University

Feb 2019

Contents

1 Introduction

1.1 Overview

A chatbot (also known as a smartbot, talkbot, chatterbot, or artificial conversational entity) is a computer program or an artificial intelligence which conducts a conversation via auditory or textual methods.[1] Such programs are often designed to convincingly simulate how a human would behave as a conversational partner, thereby passing the Turing test. Chatbots are typically used in dialog systems for various practical purposes including customer service or information acquisition.

Today, most chatbots are accessed via virtual assistants such as Google Assistant and Amazon Alexa, via messaging apps such as Facebook Messenger or WeChat, or via individual organizations' apps and websites.[2] Chatbots can be classified into usage categories such as conversational commerce (e-commerce via chat), analytics, communication, customer support, design, developer tools, education, entertainment, finance, food, games, health, HR, marketing, news, personal, productivity, shopping, social, sports, travel and utilities.[3]

1.2 Application

"A 2017 study showed 4% of companies used chatbots. According to a 2016 study, 80% of businesses said they intended to have one by 2020."[4]

The survey above shows that the chatbot project like this project is necessary, and there tend to be a broader market in the near future.

Many companies' chatbots run on messaging apps like Facebook Messenger (since 2016), WeChat (since 2013), WhatsApp, LiveChat, Kik, Slack, Line, Telegram, or simply via SMS. They are used for B2C customer service, sales and marketing.[5] The bots usually appear as one of the user's contacts, but can sometimes act as participants in a group chat.

Many banks, insurers, media companies, e-commerce companies, airlines, hotel chains, retailers, health care providers, government entities and restaurant chains have used chatbots to answer simple questions, increase customer engagement, for promotion, and to offer additional ways to order from them.[6]

2 Intent Extraction

The intent identification part is done with the help of *RasaNLU* framework.

RasaNLU supports multiple intent classifiers like `sklearn`, mitie and `tensorflow`. But we adopt `sklearn` with classical SVM classifier, since deep learning methods need a larger training set to give satisfactory accuracy.

2.1 Preprocess

2.1.1 Tokenize text to words

Basic tokenizer in NLP tasks is used here to tokenize a sentence into tokens:

```
"Suggest me some mexican restaurants"
["suggest", "me", "some", "mexican", "restaurants"]
```

2.1.2 Convert words to features

`Spacy word2vec` is adopted to convert words to numbers. During this process the tokens are mapped in a N-Dimensional space based on its similarity with other words in spacy's pre-trained corpus.

```
["suggest", "me", "some", "mexican", "restaurants"]
[1.40101, 1.3003, 0.45647, 1.8934, 1.67677]
```

2.1.3 Labeling features

Intents are categorized by a label given in training data. These labels are usually the actions the intent meant to perform. Few examples of intent labels in this chatbot project are `greet`, `current_price` etc.

We have converted text-samples from words to numbers for our ML model. When it comes to labels we do not convert it into word vectors rather assign a unique number

to it. This process is called as encoding.

RasaNLU uses Sklearn's LabelEncoder to perform this step.

```
>>> labels
[greet, good_bye, restuarant_search, good_bye]
>>> le = LabelEncoder()
>>> y = le.fit_transform(labels)
>>> y [0, 1, 2, 1 ...
```

2.2 Classify intents

The ML algorithm that RasaNLU uses to classify intents is the `SVM` with `GridSearchCV`. The advantage of `GridSearchCV` is that you can train `SVM` with different configurations and at the end of training it returns a trained `SVM` model with the best configuration.

2.2.1 Configuration

The configuration defines list of all possible parameter `GridSearchCV` would train `SVM` on. `GridSearchCV` will create `SVM`'s with all the combinations.

```
defaults = {
"C":  [1, 2, 5, 10, 20, 100],
"kernels":  ["linear"],
"max_cross_validation_folds":  5}
C = defaults["C"]
kernels = defaults["kernels"]
tuned_parameters = ["C":  C, "kernel":  [str(k) for k in kernels]]
folds = defaults["max_cross_validation_folds"]
cv_splits = max(2, min(folds, np.min(np.bincount(y)) // 5))
```

2.2.2 Training

1. Create GridSearchCV object with defined configuration
2. Fit the training data using fit method.

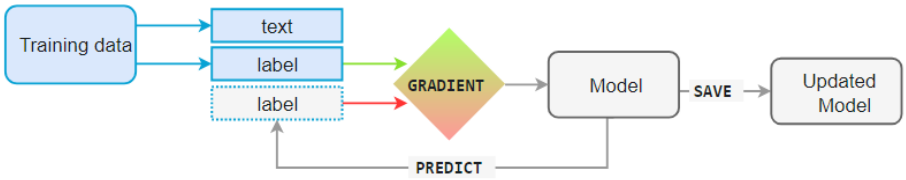

Figure 1: The process of model training in spaCy

After fitting the data, there will be an N-Dimensional statistical model which can classify similar texts to one of the trained intents.

```
clf = GridSearchCV(
SVC(C=1, probability=True, class_weight='balanced'),
param_grid=tuned_parameters, n_jobs=1, cv=cv_splits,
scoring='f1_weighted', verbose=1)
clf.fit(X, y)
```

2.3 Testing

First we are going to test the ML model with one training example from our training data. Since we already have the features extracted lets feed it into `clf.predict` method. The prediction returns an numerical value which needs to be decoded to arrive at the actual intent tag.[7]

```
# show me few mexican restaurants
>>> X_test = intent_examples[24].get("text_features").reshape(1,
-1)
>>> pred_result = clf.predict(X_test)
>>> le.inverse_transform(pred_result)
['restaurant_search']
```

3 Named Entity Recognition

The chatbot use `ner_crf` extractor capsulated in `sklearn-crfsuite`, which uses CRF as its model.

Conditional Random Fields (CRFs) are undirected statistical graphical models, a special case of which is a linear chain that corresponds to a conditionally trained finite-state machine. Such models are well suited to sequence analysis, and CRFs in particular have been shown to be useful in part-of-speech tagging, shallow parsing, and named en- tity recognition for newswire data.

Let $\boldsymbol{o} = \langle o_1, o_2, ..., o_n \rangle$ be an sequence of observed words of length $\boldsymbol{n}$. Let $\boldsymbol{S}$ be a set of states in a finite state machine, each corresponding to a label $\boldsymbol{l} \in \boldsymbol{L}$ (e.g. `greet`, `current_price`, etc.). Let $\boldsymbol{s} = \langle s_1, s_2, ..., s_n \rangle$ be the sequence of states in S that correspond to the labels assigned to words in the input sequence $\mathbf{o}$. Linear-chain CRFs define the conditional probability of a state sequence given an input sequence to be:

$$\boldsymbol{P(s|o)} = \frac{\mathbf{1}}{\boldsymbol{Z_o}} \boldsymbol{exp}\left(\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}}\sum_{\boldsymbol{j=1}}^{\boldsymbol{m}} \boldsymbol{\lambda_j f_j(s_{i-1}, s_i, o, i)}\right) \tag{1}$$

where $\boldsymbol{Z_o}$ is a normalization factor of all state sequences, $\boldsymbol{f_j}$ is one of m functions that describes a feature, and $\boldsymbol{\lambda_j}$ is a learned weight for each such feature function. This paper considers the case of CRFs that use a first-order Markov independence assumption with binary feature functions. For example, a feature may have a value of 0 in most cases, but given the text "the current price of" it has the value 1 along the transition where $\boldsymbol{s_{i-1}}$ corresponds to a state with the label `current_price`, $\boldsymbol{s_i}$ corresponds to a state with the label `greet`, and $\boldsymbol{f_j}$ is the feature function Word=current price $\in \boldsymbol{o}$ at position $\boldsymbol{i}$ in the sequence.

Intuitively, the learned feature weight $\boldsymbol{\lambda_j}$ for each feature $\boldsymbol{f_j}$ should be positive for features that are correlated with the target label, negative for features that are anti-correlated with the label, and near zero for relatively uninformative features. These weights are set to maximize the conditional log likelihood of labeled sequences in a training set $\boldsymbol{D} = \left\{\langle o, l\rangle_{(1)}, ..., \langle o, l\rangle_{(n)}\right\}$:

$$\boldsymbol{LL(D)} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{log}\left(\boldsymbol{P}(l_{(i)}|o_{(i)})\right) - \sum_{\boldsymbol{j=1}}^{\boldsymbol{m}} \frac{\boldsymbol{\lambda_j^2}}{\boldsymbol{2\sigma^2}} \tag{2}$$

When the training state sequences are fully labeled and unambiguous, the objective function is convex, thus the model is guaranteed to find the optimal weight settings in terms of $\boldsymbol{LL}(\boldsymbol{D})$. Once these settings are found, the labeling for an new, unlabeled sequence can be done using a modified Viterbi algorithm.[8]

For this chatbot project, in the object returned after parsing there are two fields that show information about how the pipeline impacted the entities returned.

The `extractor` field of an entity tells you which entity extractor found this particular entity. The `processors` field contains the name of components that altered this specific entity. The use of synonyms can also cause the `value` field not match the `text` exactly. Instead it will return the trained synonym.

```
"text": "show me chinese restaurants",
"intent": "restaurant_search",
"entities": [{
"start": 8,
"end": 15,
"value": "chinese",
"entity": "cuisine",
"extractor": "ner_crf",
"confidence": 0.854,
"processors": []
}]
```

Here are some examples of how the entities are recognized.

```
A = "Hi".
B = "Tell me the price of Tesla."
C = "I want to get the historical close price of TSLA".
```

Text	Intent	Entities	Start	End	Value
A	"greet"	-	-	-	-
B	"current_price"	company	21	26	"Tesla"
C	"vague_query"	hst_data_type	29	34	"close"

Table 1: Examples of the extraction of entities with CRF

4 Finite State Machine

4.1 Definition and rationale

A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation.

It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs.

A further distinction is between deterministic (DFA) and non-deterministic (NFA) automata. In a non-deterministic automaton, an input can lead to one, more than one, or no transition for a given state. The stock chatbot in this project complies with the NFA transition rules.

In accordance with the general classification, the following formal definitions are found:

A deterministic finite state machine or acceptor deterministic finite state machine is a quintuple $(\sigma, S, s_0, \delta, F)$, where:

- σ is the input alphabet (a finite, non-empty set of symbols).
- S is a finite, non-empty set of states.
- s_0 is an initial state, an element of S.
- δ is the state-transition function:$\delta : S \times \sigma \rightarrow P(S)$, i.e., δ would return a set of states.
- F is the set of final states, a (possibly empty) subset of S.

For both deterministic and non-deterministic FSMs, it is conventional to allow δ to be a partial function, i.e. $\delta(q, x)$ does not have to be defined for every combination of $q \in S$ and $x \in \Sigma$. If an FSM M is in a state q, the next symbol is x and $\delta(q, x)$ is not defined, then M can announce an error (i.e. reject the input).[9]

4.2 Application of FSM

In this stock chatbot, it complies with the NFA transition rules as follows:

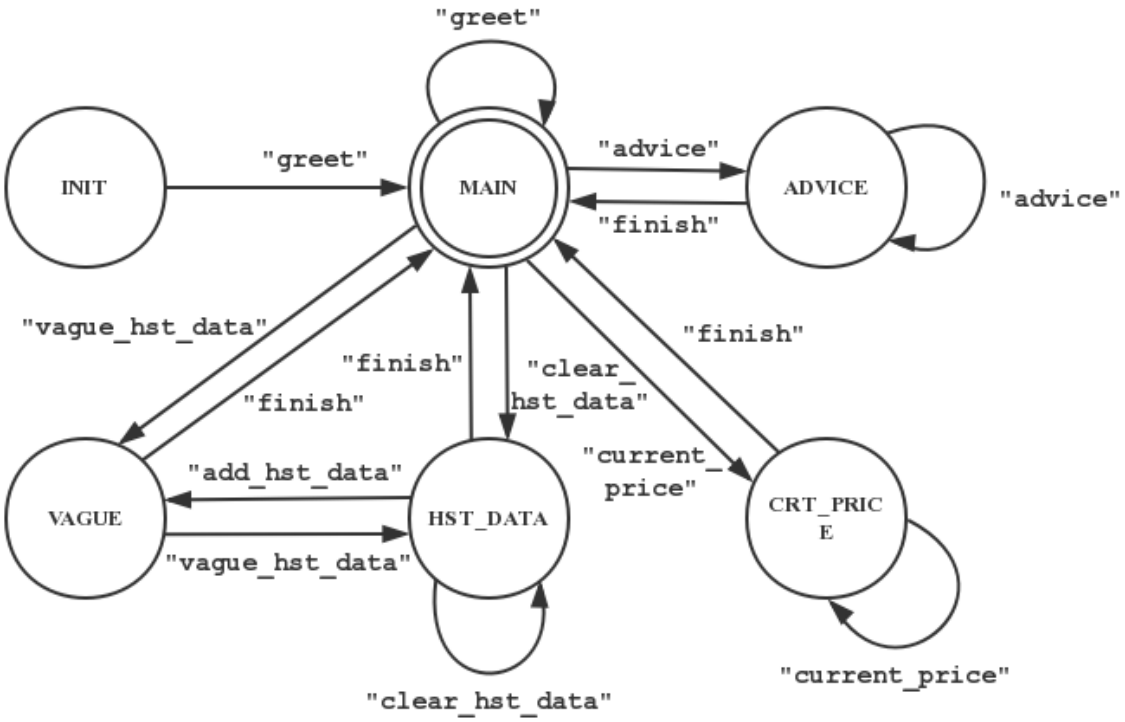

Figure 2: NFA for the stock chatbot

The chatbot will give a corresponding response (e.g. The chatbot will query the current price of a certain stock if the chatbot extract "current_price" intent from the user's text) based on the intent and entities from the user's sentence.

There will also be cases that the chatbot cannot get enough information about a specific action, then the chatbot will launch a new round of query to the user. For instance, ff the chatbot find the intent `"vague_hst_data"`, then it will ask for additional information. Then it will:

1. Received intent `"add_hst_data"`(i.e. additional info is given): Do the query based on the sum of information from the two rounds.
2. Received other intent: Exit this round of query to wait for further access.

5 Summary

The chatbot is designed to help clients with stock information.

With this chatbot, clients can query various stock indicators conveniently. And it can also give brief investment suggestions.

The chatbot is associated with *Wechat* app via *wxpy* package. It uses RasaNLU to train the model. The following techniques or methods are implemented:

- Multiple selective answers to the same question and provide a default answer.
- Intent recognition based on `sklearn` and `spacy`.
- Named entity recognition using conditional random fields.
- Single-round incremental query for multiple times based on incremental filter.
- Multiple rounds of multi-query technology on state machines, and providing explanations and answers based on contextual issues.
- Handling pending state transitions and pending actions.
- Complex pandas Dataframe processing and data cleaning, and producing a corresponding matplotlib figure.

References

[1] Margaret Rouse.*"What is a chatbot?"*. Retrieved from techtarget.com.

[2] Orf, Darren. *"Google Assistant Is a Mega AI Bot That Wants To Be Absoutely Everywhere"*. Retrieved from gizmodo.com.

[3] *"2017 Messenger Bot Landscape, a Public Spreadsheet Gathering 1000+ Messenger Bots"*. 3 May 2017.

[4] Business Insider Intelligence (Dec. 14, 2016) *"80% of businesses want chatbots by 2020"*. Retrieved from www.businessinsider.com.

[5] Beaver, Laurie (July 2016). *The Chatbots Explainer*. BI Intelligence.

[6] *"Chatbots Take Education To the Next Level – Chatbot News Daily"*. Chatbot News Daily. 2016-09-29.

[7] Bhavani Ravi(Sep 7, 2018) *Intent Classification-Demystifying RasaNLU*, Retrieved from hackernoon.com.

[8] Burr Settles, *Biomedical Named Entity Recognition Using Conditional Random Fields and Rich Feature Sets*, JNLPBA '04 Proceedings of the International Joint Workshop on NLP in Biomedicine and its Applications, Pages 104-107

[9] *Finite-state machine*. Retrieved from wikipedia.org

Apendix: Parts of source code

This code part is used to build the training set.

Program 1: `Training set examples`

```
{
   "text": "hey",
   "intent": "greet",
   "entities": []
},
{
   "text": "tell␣me␣the␣current␣price␣of␣Tesla",
   "intent": "current_price",
   "entities": [
     {
       "start":29,
       "end":35,
       "value":"TSLA",
       "entity":"company"
     }
   ]
}
```

This code part is used to define the policy rules (i.e. the transition rules of FSM).

Program 2: `Policy_rules examples`

```
# Received clear historical data information
(MAIN, "clear_historical_data"): (MAIN, response_group[5], None),
# Received vague information, ask for more
(MAIN, "vague_historical_data"): (MAIN, response_group[4], HST_DATA),
(HST_DATA, "vague_historical_data"): (MAIN, response_group[4], HST_DATA),
# Get additional information to finish the query
(MAIN, "add_historical_data"): (HST_DATA, response_group[5], None),
(HST_DATA, "vague_historical_data"): (HST_DATA, response_group[5], None),
# Finished, return to main menu
(HST_DATA, "finish"): (MAIN, med2, None),
# Response to provide advice
(MAIN, "advice"): (MAIN, response_group[10], None)
```

This code part is used to handle with pending action cases.

Program 3: `send_message() function to deal with pending action`

```
def send_message(state, pending, message):
    new_state, response, pending_state = respond(state, message)
    if pending is not None:
        new_state, response, pending_state = policy_rules[pending]
    if pending_state is not None:
        pending = (pending_state, get_intent(message))
    return new_state, pending, response, get_intent(message)
```

This code part is used to query corresponding information after identifying the intent.

Program 4: `Action of the chatbot after identifying the intent`

```
# If client is querying current price
   if get_intent(message) == 'current_price':
       response = policy_rules[(state, get_intent(message))][1].format(entity,
                                get_current_price(entity))
```

This code part is used deploy the chatbot onto *Wechat* platform.

Program 5: `Deploying chatbot on Wechat`

```
#------ Wechat bot part ------
from wxpy import *
# Create a new Bot object
bot = Bot()
# Set target client account
my_friend = bot.friends().search('Chatbot_KennethSun')[0]

#------ Rrgister pre-defined reply function ------
@bot.register(my_friend, TEXT)
def auto_reply(msg):
    state = MAIN
    pending = None
    print(get_intent(msg.text))
    state, pending, final_response, message_intent = send_message(state, pending,
msg.text)
    msg.reply(final_response)
    # Save matplotlib figure to local path and send it via chatbot
    if message_intent == 'clear_hst_data' or message_intent == 'add_historical_data':
        msg.reply_image('fig.png')
    return final_response
```