Project report for CAP 5610 Machine Learning Traffic Information Command Voice Recognizer

Yudong Guang
FIU CIS
PID: 4012835
Email: yguan004@fiu.edu

Huibo Wang
FIU CIS
PID: 4837350
Email: hwang033@fiu.edu

***Abstract*—In this project report, we have recorded our exploration and implementation of a traffic information voice command recognition system. The system is designed to work as a module that the outer system can use to transform user's voice to a URL that it can use to retrieve the corresponding information that the user is interested in. In this writeup, we introduced our motivation of doing this project, our design of the system, and most importantly, recorded our exploration on various machine learning tools and algorithms. We have completed the modules in the design and have a runnable version. We did an evaluation on the accuracy of the system and also listed the possible future work.**

I. Introduction

With the exponentially growth of the natural language processing technology, more and more softwares provide voice command recognizer as an import interacted method with the user. Besides by touching and pressing keyboard, user can interact with the software by simply telling the software what they want to do. The main advantage is that it facilitates the communication between human and software, especially when user can not use their hands to operate the software. The main goal of implementing voice command control is to get efficient way for humans to communicate with computers.

ITPA(Informed Traveler Program and Applications[1]), which we are currently working on, is a project that would provide personalized, timely information and advice regarding the most efficient and cost effective travel paths for consumers, specifically for the university area around FIU.

The ITPA will provide a smartphone-based interface for users to acquire, request and manipulate the information needed. Apart from the common interactions between the user and APP via physically touching the screen, it is very helpful to allow users to interact with the APP by voice.

The project implemented the traffic information voice command recognizer (will be referred to as TCVR in the following context) for the ITPA APP. Briefly, TCVR takes the users voice as input, recognize the words in the sentence, and extract the semantic meaning of the command, which will allow the APP to send the corresponding request to the backend. TCVR should be able to recognize the common requests of users, including but not limited to getting parking information, getting direction to certain location and sending public transit request.

II. Motivation

In some scenarios, commanding by voice is the most 'natural' and intuitive way, which can let users interact with computer with the minimum or even without learning. For example, using voice-controlled personal computers for dictation is an important application for physically disabled or layers; another application is environmental control, such as turning on the light, controlling the TV etc. In some other cases, freeing hands from devices can let people concentrate more on the task instead of controlling. For example, when people are driving and using cellphone to search for point of interest, it is safer and easier to let user speak instead of typing.

Hence, apart from the common interactions between the user and APP via physically touching the screen, it is very helpful to allow users to interact with the APP by voice. For example, the user can speak to the APP to get the parking information of the nearest parking garage in campus, or the user can speak to the APP to send a transit request. Allowing interacting with the APP by voice can prevent the users from typing on the phone while driving and thus improve the overall safety of traffic on campus, especially FIU is one of the most populated university in America.

III. Architecture

A. ITPA architecture

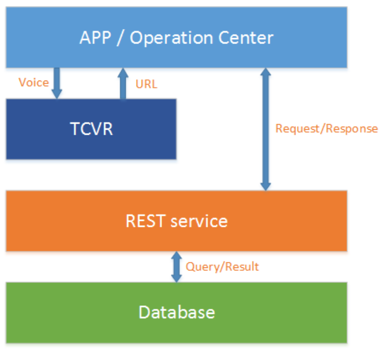


Fig. 1: ITPA architecture

Figure 1 shows the architecture of ITPA system and the interaction between ITPA and traffic command voice recognizer(TCVR). The front-end uses Rest-full service to communicate with back-end. The work flow of ITPA system is:

1). The APP/Operation Center receive user's command then send HTTP request to the Rest-full service. If the command is send using voice, TCVR will analyze the voice command and return the correct HTTP request;
2). Rest-full service query the database and get the result. The result is wrapped into geo-json format and send back to front-end via http response;
3). Front-end get the response and show the result.

So due to the TCVR, user can choose the way to send the request by voice command or press the button on the screen.

B. *TCVR architecture*

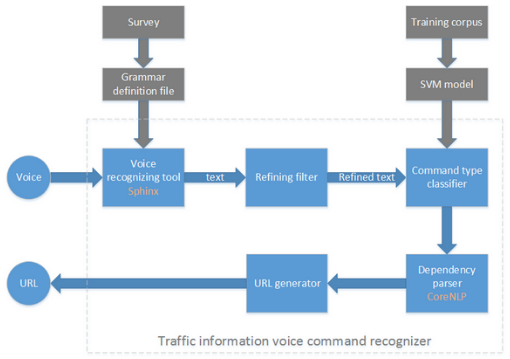


Fig. 2: The architecture of traffic command voice recognizer

Figure 2 shows the architecture of ITPA traffic command voice recognizer. There are 5 modules in the system. They are voice recognizing tool, refining filter, command type classifier, dependency parser and URL generator.

The input of this system is voice, and output is the corresponding URL. ITPA can call the URL to retrieve and display the requested transit information.

Voice recognizing tool transform the input voice to text format. In this module a grammar file is needed to define the questions' type. This grammar file is generated from a survey. Different person may have different way to express the same meaning. So we used a survey to collect how people asked questions. This module is based on CMU Sphinx speech recognition package.

Refining filter module is used to refine the text generated from voice recognizing tool. Bus route, bus stops, garage name and abbreviation will be replaced by a unified format. The refined text file will improve the accuracy of classifier.

Command type classifier leverages support vector machine (SVM). There are 12 types of questions, such as parking occupancy information, user position and bus location. The classifier will predict the question type for each refined text based on the trained SVM model. The model is trained using the questions from the Grammar file.

Dependency parser provides the grammatical structure of sentences, for instance, which words are subject or object of a verb. After the question type is determined, the next step is extracting the parameters. For example, user may request the bus location from FIU main campus to engineer center. Dependency parser will extract the engineer center and FIU main campus and the "from-to" relationship between them.

The last module is called URL generator. Each question type has a corresponding URL, and each URL may have some parameters. The question type is calculated from command type classifier and the parameters is extracted from the dependency parser. In this module is trying to map the question type to a URL and give the parameters.

IV. Implementation

A. *Voice recognition tool*

1) Our choice on voice recognition tool: Since our system will eventually be a part of the ITPA mobile App, one important concern we had when choosing the tool to use was the portability and platform compatibility, which means we hope the voice recognition tool to be small, efficient in computation and most importantly cross-platform.

With that said, the deep learning based tool Kaldi has a significant drawback to us in terms of the size, efficiency and the platform compatibility. Its ideal computing environment is a cluster of Linux machines (any major distribution) running Sun GridEngine (SGE), with access to shared directories via NFS or some similar network filesystem. In the ideal case, some computers on the grid will have NVidia GPUs which you can use for neural net training.

The CMU Sphinx is a much better choice for our system because it is written in Java and it has a portable version called 'pocketsphinx', which has some features missing, but ideal for a system which recognizes a dozen commands without the Internet connection. For our project, we have chosen to start with the full-fledged version Sphinx4 because we hoped to finish the workflow first and provide a runnable version, and leave the porting to mobile device as a future work.

The current Sphinx system has a dramatic reduction in speech recognition errors by improving feature representations, using multiple codebook semi-continuous hidden Markov models, between-word senones, multi-pass search algorithms, and unified acoustic and language modeling. Sphinx system has higher quality of feature extraction by taking temporal information into consideration. It has the most detailed modeling possible by means of parameters sharing with SCHMM. It increased the search quality by incrementally apply all available acoustic and linguistic information in three search phases.[2], [3], [4]

a) Required models: According to the speech structure, three models are used in speech recognition to do the match:

(1) An *acoustic model* contains acoustic properties for each senone. There are context-independent models that contain properties (most probable feature vectors for each phone) and context-dependent ones (built from senones with context).

(2) A *phonetic dictionary* contains a mapping from words to phones. This mapping is not very effective. For example, only two to three pronunciation variants are noted in it, but it's practical enough most of the time. The dictionary is not the only variant of mapper from words

to phones. It could be done with some complex function learned with a machine learning algorithm.

(3) A *language model* is used to restrict word search. It defines which word could follow previously recognized words (remember that matching is a sequential process) and helps to significantly restrict the matching process by stripping words that are not probable. Most common language models used are n-gram language models-these contain statistics of word sequences-and finite state language models-these define speech sequences by finite state automation, sometimes with weights. To reach a good accuracy rate, your language model must be very successful in search space restriction. This means it should be very good at predicting the next word. A language model usually restricts the vocabulary considered to the words it contains. That's an issue for name recognition. To deal with this, a language model can contain smaller chunks like subwords or even phones. Please note that search space restriction in this case is usually worse and corresponding recognition accuracies are lower than with a word-based language model.

These three models are combined together in an engine to recognize speech. In order to make a voice recognizer that can understand certain set of commands in specific domain, we can use the general acoustic model and phonetic dictionary, but we need to provide a language model that can significantly increase the accuracy of the commands we expected to support.

b) Language Model: There are two types of models that describe language - grammars and statistical language models. Grammars describe very simple types of languages for command and control, and they are usually written by hand or generated automatically with scripting code. Grammars usually do not have probabilities for word sequences, but some elements might be weighed. Grammars could be created with JSGF format and usually have extension like .gram or .jsgf.

Statistical language models describe more complex language. They contain probabilities of the words and word combinations. There are many ways to build the statistical language models. CMU language modeling toolkit can help to build the statistical language model if the data set is large; there is also an online version for small data sets.

c) The survey: In order to build the language model, we firstly need to know which commands exactly we need to support. Conceptually we knew that we need to support the following categories of commands:

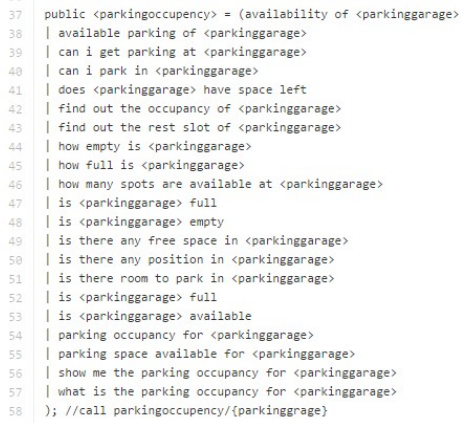
- Dynamic bus position
- Estimate arrival time for each bus
- Show bus station
- The estimate arrival time per bus stops
- Nearest bus stop
- Show bus route
- Show nearest bus route
- Show parking garage
- Show parking occupancy
- Recommend a parking place
- Show user's position

There are many ways of saying a command for each category of question. In order to better enumerate the ways of saying each command, we have designed a survey to collect people's opinion. We handed out the survey to out ITPA team and also the neighbor team, which totaling 16 people. We received 16 completed surveys and used these surveys to define our language model.

d) JSGF: Since the set of commands to support is not large in our case, we have decided to use the grammar definition file instead of the statistical language models. The JSpeech Grammar Format (JSGF[5]) is a platform-independent, vendor-independent textual representation of grammars for use in speech recognition. Grammars are used by speech recognizers to determine what the recognizer should listen for, and so describe the utterances a user may say. JSGF adopts the style and conventions of the Java Programming Language in addition to use of traditional grammar notations.

We completed the building of the language model in JSGF format and fed to the Sphinx. The result shows that Sphinx can recognize the sentences that are defined in the grammar file, which made the subsequent processing possible. As to the accuracy of the Sphinx voice recognizer, we haven't got the chance to do a more detailed evaluation, and may be done as a future work.

Figure 3 shows the defined sentences for parking occupancy class. In this class, availability of $< parkinggarage >$ can be extended to " availability of parking garage pg1 ", " availability of parking garage pg2 ", etc.



```
public <parkingoccupency> = (availability of <parkinggarage>
| available parking of <parkinggarage>
| can i get parking at <parkinggarage>
| can i park in <parkinggarage>
| does <parkinggarage> have space left
| find out the occupancy of <parkinggarage>
| find out the rest slot of <parkinggarage>
| how empty is <parkinggarage>
| how full is <parkinggarage>
| how many spots are available at <parkinggarage>
| is <parkinggarage> full
| is <parkinggarage> empty
| is there any free space in <parkinggarage>
| is there any position in <parkinggarage>
| is there room to park in <parkinggarage>
| is <parkinggarage> full
| is <parkinggarage> available
| parking occupancy for <parkinggarage>
| parking space available for <parkinggarage>
| show me the parking occupancy for <parkinggarage>
| what is the parking occupancy for <parkinggarage>
); //call parkingoccupency/{parkinggrage}
```

Fig. 3: Small class grammar

B. Refining filter

The output of the voice recognition tool is the sentence that Sphinx thinks right. One little problem we had with Sphinx is that we were using the generic acoustic model and dictionary. This brought a problem that we cannot use the words that are not in the provided generic vocabulary. However, we have a bunch of words that are not generic, such as "PG5", "GPE", "MMC". Fortunately, these words are all abbreviations. So in the JSGF grammar file, we have expressed these words the same as its pronunciation. For example, “PG5” can be expressed as “p g five”; “MMC” can be expressed as “m m c”.

With this workaround, we can let the Sphinx recognize the words that are not actually in the dictionary. Yet this also brought a little problem for the subsequent processing. For example, the “MMC” as a word is meaningful semantically, but “m m c” may be confusing for the semantic extraction tools. Hence we have implemented a filter to transform these words back to the original form as they should be.

This part is simple because it only replaces the recognized abbreviations to the form originally should be. We have used a simple string replacement algorithm to implement this module.

The output of this module are refined sentences with the right named entities. The semantic extraction can use the output of this module to retrieve the information in the text.

C. Voice command type classifier

Initially there was no such module because the straightforward way is to feed the recognized text into a dependency parser and process the parsed tree to extract the semantics. However, We found the processing in the extraction would be too complicated. Because we have about a dozen of categories to support, which would introduce a lot of conditions checking and thus a lot of manual work would be required.

After reviewing the collected surveys, we found that there are very obvious differences in the vocabulary in different categories of commands. For example, only the category of “bus ETA” has the word “ETA”. Then it came to our mind that a classifier may very well distinguish the types of commands. And if we can identify the type of a command, it would be very easy to do the following processing. For example, if we can know the command is about the real-time bus location, then the only thing we need to do is to extract the bus route that the user is interested in. It would required less work or is even trial to achieve.

1) Classification methods: As to the text classifier, we initially thought of three methods.

- Decision Tree [6]
- Random Forest [7]
- SVM [8]

2) Training:

a) Training data: To work with the text classifier, we need to provide a proper set of training samples to the algorithms. There were two ways of getting the training data: (1) Gathering from survey. (2) Generating from JSGF grammar file.

The first way is easy and intuitive. But it was not the best because the surveys were ‘raw’ in terms of the quality of the provided commands. Each people who filled in the survey had a different attitude and different level of understanding of ITPA project. Some of the commands they provided may not be easy to implement. So a data cleaning is very necessary. Then it may require a lot of manual work.

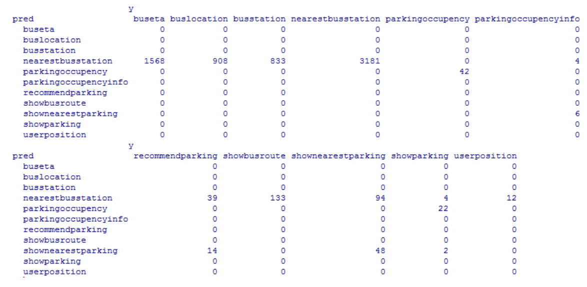
Then we came up with the second option, which is to generate the commands we expect to support from the JSGF grammar file we wrote. This is easier because with an automated script we can almost enumerate all the possible commands we will support. With a simple python program, we generated around 7000 commands that we need to support. And each command has the expected label attached, which is the type of command that it should be.

To express the commands in the form of vectors, we have processed them in the following way. Firstly make a closure of the vocabulary of all the commands. This means finding all the words that may occur in the commands we need to support. The result of this step found that there are about 170 words in total, which is not bigger than expected, and we’ll refer to this number as N in the following context. The second step is to convert each command to a vector of N dimensions. Each dimension is the number of occurrence of the word. Then after doing this we have converted the text commands into a set of vectors of the same length.

b) Training process: Once we have the training data, we can feed them to the algorithms. For each method, we have chosen an open-source library. These libraries require the training data in different formats. The QuickML library provides Java implementations for both decision tree and random forest. The libsvm is a well-known open-source implementation of SVM.

The training process is to feed the training vectors to the algorithms. To evaluate the accuracy of the different algorithms, we used R to do the evaluation. The model is built with the whole training data, then test with the whole training data. Decision tree is from ”party” library. Random forest is from ”randomForest” library. Support Vector Machine is using the ”e1071”(”libsvm”) library. The training accuracy is listed below:

c) Decision Tree: Figure 4 shows the classify result for each class. Training accuracy using decision tree is 0.4733719 that is very low. Almost every data is classified as nearest bus station class.

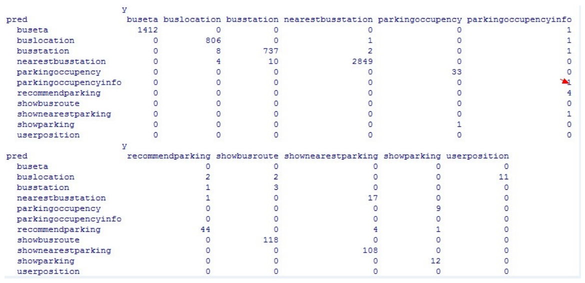


```
                     y
pred                  buseta buslocation busstation nearestbusstation parkingoccupency parkingoccupencyinfo
  buseta                   0           0          0                 0                0                    0
  buslocation              0           0          0                 0                0                    0
  busstation               0           0          0                 0                0                    0
  nearestbusstation     1568         908        833              3181                0                    4
  parkingoccupency         0           0          0                 0               42                    0
  parkingoccupencyinfo     0           0          0                 0                0                    0
  recommendparking         0           0          0                 0                0                    0
  showbusroute             0           0          0                 0                0                    0
  shownearestparking       0           0          0                 0                0                    6
  showparking              0           0          0                 0                0                    0
  userposition             0           0          0                 0                0                    0
                     y
pred                  recommendparking showbusroute shownearestparking showparking userposition
  buseta                             0            0                  0           0            0
  buslocation                        0            0                  0           0            0
  busstation                         0            0                  0           0            0
  nearestbusstation                 39          133                 94           4           12
  parkingoccupency                   0            0                  0          22            0
  parkingoccupencyinfo               0            0                  0           0            0
  recommendparking                   0            0                  0           0            0
  showbusroute                       0            0                  0           0            0
  shownearestparking                14            0                 48           2            0
  showparking                        0            0                  0           0            0
  userposition                       0            0                  0           0            0
```

Fig. 4: Accuracy of decision tree

d) Random Forest: Figure 5 shows the random forest classify result for each class. Training accuracy for random forest is 0.9863014 when the number of tree is 100. Although the result seems very good, the accuracy for parking occupency info class is 0.11. Only one training example is classified correctly. For user position class, the accurate rate is 0. Random forest is not doing well on small class.

Figure 6 shows the Relationship between error rate and the



```
                     y
pred                  buseta buslocation busstation nearestbusstation parkingoccupency parkingoccupencyinfo
  buseta                1412           0          0                 0                0                    1
  buslocation              0         806          0                 1                0                    1
  busstation               0           8        737                 2                0                    1
  nearestbusstation        0           4         10              2849                0                    0
  parkingoccupency         0           0          0                 0               33                    0
  parkingoccupencyinfo     0           0          0                 0                0                    1
  recommendparking         0           0          0                 0                0                    4
  showbusroute             0           0          0                 0                0                    0
  shownearestparking       0           0          0                 0                0                    1
  showparking              0           0          0                 0                1                    0
  userposition             0           0          0                 0                0                    0
                     y
pred                  recommendparking showbusroute shownearestparking showparking userposition
  buseta                             0            0                  0           0            0
  buslocation                        2            2                  0           0           11
  busstation                         1            3                  0           0            0
  nearestbusstation                  1            0                 17           0            0
  parkingoccupency                   0            0                  0           9            0
  parkingoccupencyinfo               0            0                  0           0            0
  recommendparking                  44            0                  4           1            0
  showbusroute                       0          118                  0           0            0
  shownearestparking                 0            0                108           0            0
  showparking                        0            0                  0          12            0
  userposition                       0            0                  0           0            0
```

Fig. 5: Accuracy of random forest

number of the trees. Different color of line denotes different class. So enlarge the number of trees may help reduce the error. But for some small class, even using 100 trees, the error rate is still very high.

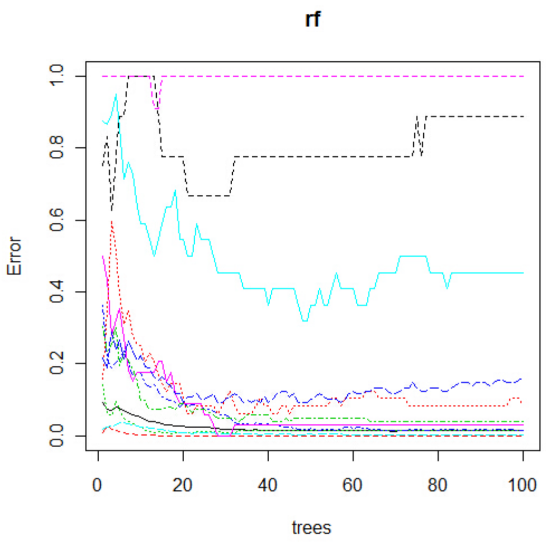
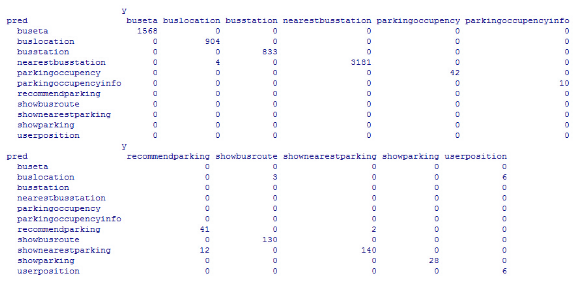


Fig. 6: Relationship between error rate and the number of the trees

e) SVM: Figure 7 shows the SVM classify result for each class. Training accuracy for SVM is 0.9960926. SVM performs very well on this data set, even for some small class. The accuracy for parking occupancy info is 1.0.



```
                     y
pred                  buseta buslocation busstation nearestbusstation parkingoccupency parkingoccupencyinfo
  buseta                1568           0          0                 0                0                    0
  buslocation              0         904          0                 0                0                    0
  busstation               0           0        833                 0                0                    0
  nearestbusstation        0           4          0              3181                0                    0
  parkingoccupency         0           0          0                 0               42                    0
  parkingoccupencyinfo     0           0          0                 0                0                   10
  recommendparking         0           0          0                 0                0                    0
  showbusroute             0           0          0                 0                0                    0
  shownearestparking       0           0          0                 0                0                    0
  showparking              0           0          0                 0                0                    0
  userposition             0           0          0                 0                0                    0
                     y
pred                  recommendparking showbusroute shownearestparking showparking userposition
  buseta                             0            0                  0           0            0
  buslocation                        0            3                  0           0            6
  busstation                         0            0                  0           0            0
  nearestbusstation                  0            0                  0           0            0
  parkingoccupency                   0            0                  0           0            0
  parkingoccupencyinfo               0            0                  0           0            0
  recommendparking                  41            0                  2           0            0
  showbusroute                       0          130                  0           0            0
  shownearestparking                12            0                140           0            0
  showparking                        0            0                  0          28            0
  userposition                       0            0                  0           0            6
```

Fig. 7: Accuracy of Support Vector Machine

3) Choice on classification method: As shown in the evaluation, SVM has the best accuracy. So we have decided to move on with SVM. We used the training data and got a model after training. After tuning the parameter, the training accuracy is 99.985528% (6909/6910). Only one training data is misclassified. The model will be used to classify the command recognized by the previous processing. The introducing of the text classifier has made the semantics extraction much easier because the amount of information to extract in the following steps would be significantly reduced.

D. Dependency parser

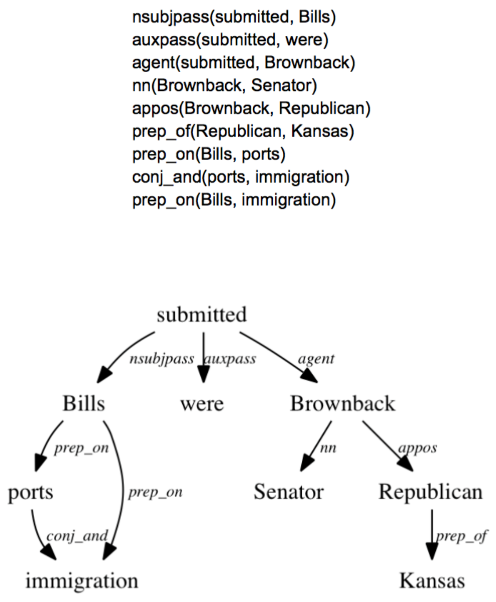
a) Systems or Softwares:

1) *Apache OpenNLP.* The Apache OpenNLP library is a machine learning based toolkit for the processing of natural language text. It supports the most common NLP tasks, such as tokenization, sentence segmentation, part-of-speech tagging, named entity extraction, chunking, parsing, and coreference resolution. These tasks are usually required to build more advanced text processing services. OpenNLP also includes maximum entropy and perceptron based machine learning.[9]
2) *Stanford CoreNLP.* Stanford CoreNLP provides a set of natural language analysis tools which can take raw text input and give the base forms of words, their parts of speech, whether they are names of companies, people, etc., normalize dates, times, and numeric quantities, and mark up the structure of sentences in terms of phrases and word dependencies, indicate which noun phrases refer to the same entities, indicate sentiment, etc. Stanford CoreNLP is an integrated framework. Its goal is to make it very easy to apply a bunch of linguistic analysis tools to a piece of text. Starting from plain text, you can run all the tools on it with just two lines of code. It is designed to be highly flexible and extensible. With a single option you can change which tools should be enabled and which should be disabled. Its analyses provide the foundational building blocks for higher-level and domain-specific text understanding applications.[10]

As to the choice between OpenNLP and CoreNLP, there is no big difference in terms of the functionality. We found CoreNLP has very detailed documentation so we decided to use CoreNLP in our project. Since the previous steps have got the command text, and the category of the command (from the classifier), the task in this step has been minimized. For example, if the command is asking for nearest bus station from “MMC” to “BBC”, the previous steps would recognized the command text, and also know it is about “nearest bus station”, then the only thing we need to do is to identify the station names that the user is interested in.

One difficulty in this step is to identify the relationship between the entities that we are trying to find. In the previous example, it is easy to identify the station names, but accurately tell the the “from” station is “MMC” but not “BBC” requires the understanding of the text, which goes beyond simple string matching.

b) Stanford dependencies: The Stanford dependencies provide a representation of grammatical relations between words in a sentence. They have been designed to be easily understood and effectively used by people who want to extract textual relations. Stanford dependencies (SD) are triplets: name of the relation, governor and dependent. The standard dependencies for the sentence "Bills on ports and immigration were submitted by Senator Brownback, Republican of Kansas" are given below(8), as well as a graphical representations: the standard dependencies (collapsed and propagated) in which each word in the sentence (except the head of the sentence) is the dependent of one other word.



```
nsubjpass(submitted, Bills)
auxpass(submitted, were)
agent(submitted, Brownback)
nn(Brownback, Senator)
appos(Brownback, Republican)
prep_of(Republican, Kansas)
prep_on(Bills, ports)
conj_and(ports, immigration)
prep_on(Bills, immigration)
```

Fig. 8: Stanford dependencies for example sentence

With the help of the dependencies information, we can easily identify the relationships between words. For example, the "from MMC" would be identified by the dependency parser, as well as the "to BBC". Since there are many ways to express a command, there are also a lot of different kinds of dependencies for the same meaning. We ran the dependency parser on all the sentences that were used to generate the training data for the classifier, and went through the parsed dependencies. We summarized all the dependencies for specific information, such as the dependencies for "arrival station", and "departure station", etc.

The result of this step can be finally used by the URL generator to generate the url, which is the output of the whole system.

E. URL Generator

The previous steps have recognized the command text, refined the text, classify the command, and resolved the dependencies between the parts of the sentence. All these information together are the input of the URL generator.

The URL generator takes these information and construct a URL for the caller of our system to retrieve the desired information for the user.

The logic in this part is relatively easy. Firstly we check the type of the command, for some of the types of commands, we do not need more information to generate a request, such as 'user position'. For some other categories, such as 'nearest bus station', we need the help of the dependency parser to get the arrival station, departure station and route name. This was made easy with the output of the dependency parser.

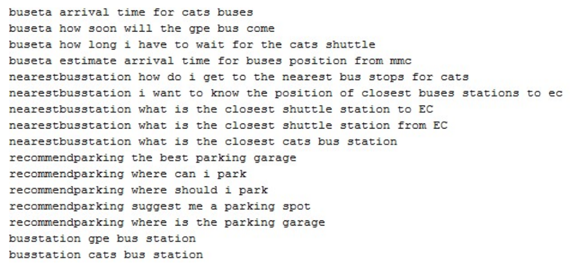
The URL generator will output a string that has the proper parameters filled in, so that the caller of this system can use it to make an HTTP request, with which the information that the user is asking for is attached.

By now, all the implementation details of the modules along the designed pipeline have been introduced. The input of the system is a voice snippet captured by the microphone, and the system will output a URL that reflects the user's actual request.

V. Evaluation

To measurement the result, we ran 2 times black box test. First time, we tested the system using frequently asked questions. There are 33 questions in total, each category has 3 questions. The system generated the correct URLs for all of them. Second time, we randomly generated 51 questions, each category has 4 to 5 questions. And test each of them on our system.

Figure 9 shows the test set. The main difference between test set and training set is that in test set we changed each pattern to specific name. For example, $< platform >$ is replaced by MMC.



```
buseta arrival time for cats buses
buseta how soon will the gpe bus come
buseta how long i have to wait for the cats shuttle
buseta estimate arrival time for buses position from mmc
nearestbusstation how do i get to the nearest bus stops for cats
nearestbusstation i want to know the position of closest buses stations to ec
nearestbusstation what is the closest shuttle station to EC
nearestbusstation what is the closest shuttle station from EC
nearestbusstation what is the closest cats bus station
recommendparking the best parking garage
recommendparking where can i park
recommendparking where should i park
recommendparking suggest me a parking spot
recommendparking where is the parking garage
busstation gpe bus station
busstation cats bus station
```

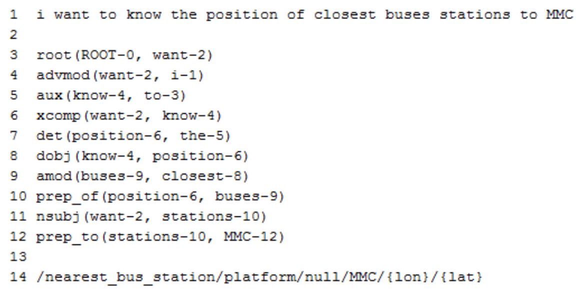
Fig. 9: Black box test set

We read each questions and verified the result. Here is an example of the output(Figure 10) of the software. Line 1 shows the input data. Line 3 to 12 is the result of dependency parser. Line 14 is the output URL.

Figure 11 shows the black box test result. The accuracy is 0.9607843 (49/51). There are two errors. It classified "gpe bus station" to /bus_eta/routenm/GPE. And marked "garage PG6" to /user_position.

VI. Future work

While developing the system we have used multiple tools, libraries, and algorithms. These tools, libraries and algorithms have all kinds of parameters to tune to achieve the best results.

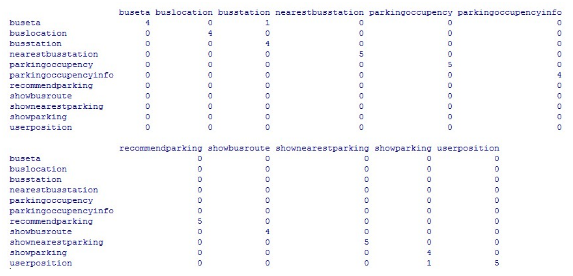


```
i want to know the position of closest buses stations to MMC

root(ROOT-0, want-2)
advmod(want-2, i-1)
aux(know-4, to-3)
xcomp(want-2, know-4)
det(position-6, the-5)
dobj(know-4, position-6)
amod(buses-9, closest-8)
prep_of(position-6, buses-9)
nsubj(want-2, stations-10)
prep_to(stations-10, MMC-12)

/nearest_bus_station/platform/null/MMC/{lon}/{lat}
```

Fig. 10: Evaluation result



```
                     buseta buslocation busstation nearestbusstation parkingoccupency parkingoccupencyinfo
buseta                    4           0          1                 0                0                    0
buslocation               0           4          0                 0                0                    0
busstation                0           0          4                 0                0                    0
nearestbusstation         0           0          0                 5                0                    0
parkingoccupency          0           0          0                 0                5                    0
parkingoccupencyinfo      0           0          0                 0                0                    4
recommendparking          0           0          0                 0                0                    0
showbusroute              0           0          0                 0                0                    0
shownearestparking        0           0          0                 0                0                    0
showparking               0           0          0                 0                0                    0
userposition              0           0          0                 0                0                    0

                     recommendparking showbusroute shownearestparking showparking userposition
buseta                              0            0                  0           0            0
buslocation                         0            0                  0           0            0
busstation                          0            0                  0           0            0
nearestbusstation                   0            0                  0           0            0
parkingoccupency                    0            0                  0           0            0
parkingoccupencyinfo                0            0                  0           0            0
recommendparking                    5            0                  0           0            0
showbusroute                        0            4                  0           0            0
shownearestparking                  0            0                  5           0            0
showparking                         0            0                  0           4            0
userposition                        0            0                  0           1            5
```

Fig. 11: Black box test set result

In the developing of this project, we have finished the work to implement all the components to make the system work. However, there are a lot to be done to make it yield the best result. The future work of the system can be stated by each module in the system.

A. About voice recognizing tool Sphinx

The tool CMU Sphinx is one of the most important parts in the system. It is the premise of the subsequent processing. When choosing the CMU Sphinx, one important concern we had was the efficiency and portability to mobile device, because finally we will integrate this system into an App as a module. The App will be run in a mobile device with significantly less storage space and computational power than a PC. My current implementation of the system is using a PC version of the Sphinx to quickly make the system work. So a significant future work in this part is to change the Sphinx to 'pocketsphinx' which is also part of the CMU Sphinx toolkit. This transformation not only involves the use of the tool, but also involves the change of the acoustic model, phonetic dictionary. The design of pocketsphinx is to give the mobile App the ability to recognize a few commands off-line. So the default acoustic model and phonetic dictionary I'm using will not be appropriate because they are for general purpose. They are too large (larger than 300MB) and thus too CPU-intensive for a mobile device. Hence the future work about Sphinx is to build or tailor a subset of acoustic model and phonetic dictionary that are just enough for the commands we wish to support.

B. About command type classifier

The classifier has helped a lot in understanding the command text. We evaluated three different classification methods: decision tree, random forest and SVM. This evaluation, however, is preliminary because we didn't tune much about each method. The reason why we moved forward with SVM was because it has yield enough high accuracy (99%) that is almost unbeatable.

However, during the evaluation of the whole system, we also found some commands that were not properly classified. This means that there might be some work to be done with the classification. This may include the following improvements:

1) Parameters tuning for the SVM model.
2) Another way to reduce the classification error might be finding more than one possible label for a command, instead of labeling a command with only one label. Then the multiple labels should be predicted with a confidence or probability, which may provide more flexibility for the subsequent processing.
3) Do dependency parsing before classification, and have the result of the dependencies as new features of the command vectors. With more features, the SVM may better classify the commands.

C. About dependency parser

Currently our use of the dependency parser is to get the dependencies only. While actually the Stanford CoreNLP is a tool-set that has many tools to understand a sentence, including the part-of-speech (POS) tagger, the named entity recognizer (NER), the parser, the co-reference resolution system, the sentiment analysis, and the bootstrapped pattern learning tools. The future work may try to use more CoreNLP tool to analyze the sentence.

References

[1] "Partnering for 21st century prosperity," http://government.fiu.edu/_assets/docs/UniversityCityProjectOverview.pdf, accessed: 2015-04-18.

[2] X. Huang, F. Alleva, H.-W. Hon, M.-Y. Hwang, K.-F. Lee, and R. Rosenfeld, "The sphinx-ii speech recognition system: an overview," *Computer Speech & Language*, vol. 7, no. 2, pp. 137–148, 1993.

[3] L. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[4] R. P. Lippmann, "Review of neural networks for speech recognition," *Neural computation*, vol. 1, no. 1, pp. 1–38, 1989.

[5] "Jspeech grammar format," http://www.w3.org/TR/jsgf/, accessed: 2015-04-18.

[6] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[7] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[8] C. J. Burges, "A tutorial on support vector machines for pattern recognition," *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[9] A. OpenNLP, "Welcome to apache opennlp."

[10] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2014, pp. 55–60.