

# **Error detection of text queries transcribed from voice input**

Team members: Chingpo Lin, Zhuoran Zhou, Aravind Narayanan

Company: Telenav

## **Table of Contents:**

Introduction/Abstract -----	3
Teams, Roles, Responsibilities -----	3
Project Schedule -----	4
Project resources/budget -----	6
Realistic Constraints -----	6
System Requirements -----	6
Operating Hazards and Requirements -----	6
System Specifications -----	7
Design Procedure/Methods -----	7
Hardware/Software design -----	11
Test/Experiment Design -----	16
Test Procedure and Results ----	19
Results and Analysis -----	20
Impact and Consequences -----	23
Conclusions and Recommendations -----	23
References, Acknowledgements, Intellectual Property -----	24
Appendices -----	25
Industry Mentor Comments -----	25

## **Introduction/Abstract:**

People tend to use voice to navigate, search addresses, order food and so on via car. However, the noise from surroundings and accent from users sometimes negatively affect the input speech to text and output text.

Our goal is to detect and eliminate such errors. Given a list of possible input queries, we want to output them in order of relevance based on what the user most likely meant. For the final product, we rerank the input queries transcribed from speech using specific techniques, and come up with a Java library that can be used on Android and one android test application to run that library. Also, we will wrap up the whole program into a jar file which could be called in the command line/terminal with commands.

## **Teams, roles, and responsibilities:**

### **Chingpo Lin: Team Leader and Java Developer**

- Writing code to create dictionaries txt files in word frequency format  
connecting to address API opencage using gradle
- Writing code in JDBC as a alternative of these txt files
- Implementing the .jar file

### **Zhuoran Zhou: Machine Learning Leader and Java Developer**

- Researching machine learning models which can fit our project goal.
- Getting in touch with machine learning faculties
- Writing codes based on natural language processing model
- Developing the Android testing App

## Aravind Narayanan: Java Developer and Project Manager

- Setting up for JDBC
- Writing code to test accuracy based on word frequency on a given input query file
- Writing code to test accuracy based on distance on given user locations.
- Writing code to incorporate bigrams and trigrams

**Faculty mentor:** Prof.Lillian Ratliff

**Industry mentors:** Akira Zhang, Kumar Maddali, Changzheng Jiang,

Srinivasa Parvathareddy

### Project Schedule:

#### Gantt Chart:

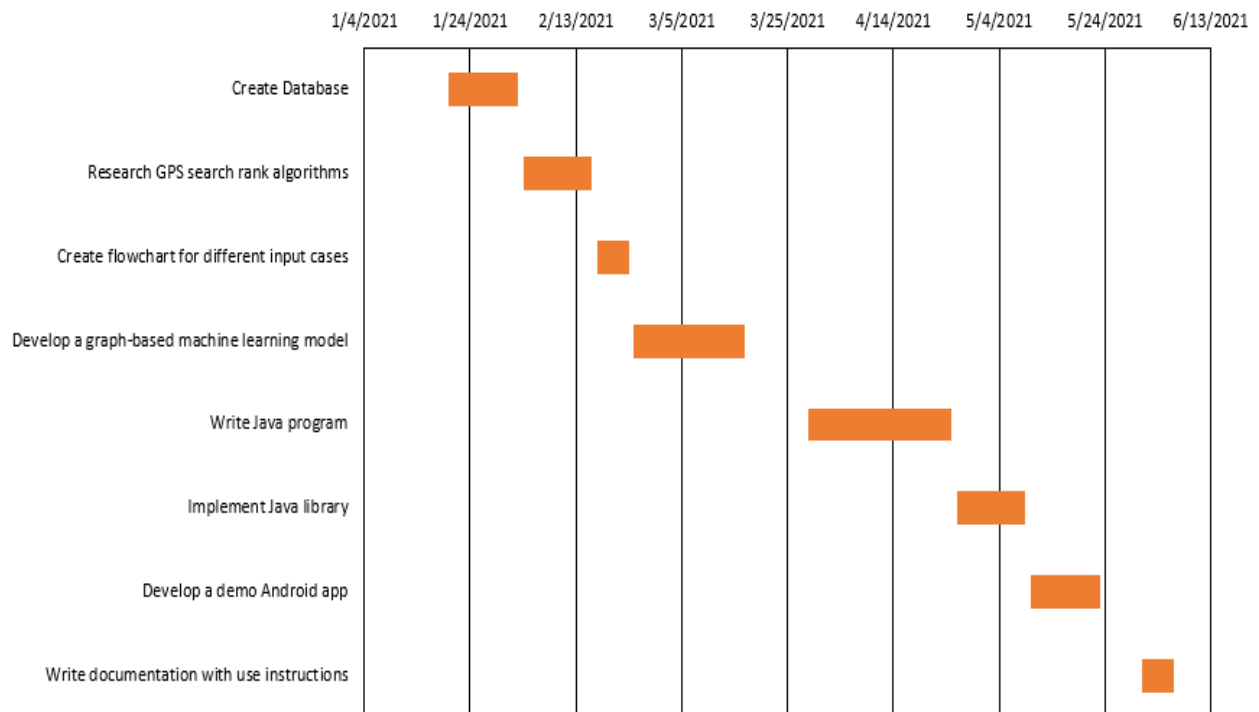


Figure (1): Gantt Chart of Timeline

Our final Gantt chart also matches our original Gantt chart. Our timeline didn't change since the end of last quarter.

### Work breakdown structure in detail:

WP A		Chingpo	Aravind	Zhuoran	Status	Timeline
All Machine Learning techniques					Done	✓ Jan 25 - Mar 10
implement JDBC					Done	✓ Jan 25 - Feb 8
Using txt file to replace JDBC					Done	✓ Feb 8 - 22
test based on frequency					Done	✓ Feb 8 - 22
connect location API					Done	✓ Feb 22 - Mar 8
test based on distance					Done	✓ Feb 22 - Mar 8
Get advise form professor					working on it	Mar 8 - 11
implement machine learning					Future steps	Mar 20 - Apr 11
implement java library					Future steps	Apr 11 - 25
get Android test app					Future steps	Apr 25 - May 31

Figure (2): Work Breakdown Structure in detail

Above was our work breakdown structure at the end of last quarter. There have been some changes since then. First of all, all of the steps are "Done". Second, we didn't really use any machine learning, but we did use a technique commonly used in machine learning called N-grams. I implemented the transition to N-grams. Also, Charlie worked on the Android test app instead of me.

### **Project Resources/Budget:**

Our budget for this project was \$0, as we did not have to buy any hardware or conduct any lab experiments. So, we do not have to be reimbursed for anything.

### **Realistic Constraints:**

There are a few constraints which set limits on our project, causing us to incur errors or imperfect outputs. First of all, the inputs given by the program might already be mistaken or divergent from the desired voice input so that it may also cause errors with erroneous inputs. As the program which processes voice recognition and generates the inputs is not revealed, we don't know its quality of processing, especially when it comes to noisy environments, different dialects and such complicated situations. Another constraint we have is that our training data set, a file with user searches called "local.txt", is limited in size. Some phrases which are actually common may not be found in this file, and could bias our results. The limited size could also lead to many ties in frequencies, because many phrases may appear only once or twice. I think that a much larger sample size will lead to more accurate results.

### **System Requirements:**

The following are the company requirements for our project:

- Given a list of possible input strings, output them in order based on relevance
- Eliminate errors of input query text from transcribed from speech
- The artifacts should be one Java library which can run our program easily
- One android test application to run the library
- Test report and documentation

### **Operating Hazards and Requirements:**

There are no operational hazards with our project, as we don't use any dangerous items.

### **System Specifications:**

Our final deliverables did not differ from the System Requirements, as we have a program, Java library, and test app. However, we don't really "eliminate" any of the queries from an input query set. We output all of the input queries in a ranked order; the worst queries are just shown last. We thought that this would be more transparent than eliminating some queries. However, maybe in the future there can be a spellchecker incorporated which checks all of the input queries and automatically discards any misspelled queries, because the voice processor obviously misheard those.

### **Design Procedure/Methods:**

We are given a master document of user searches in a file called "local.txt". We decided we could use this document in order to rank the input queries. We started out thinking that we would score queries by adding up the sum of the frequencies of every single word in the query. The frequencies of each word would be calculated by counting up the number of times the word appeared in the master document. However, we soon ran into some issues with this approach. For one, larger queries with a lot of filler words automatically got ranked higher, just because they contained more words. For example, "airport town of detroit" would get ranked higher than "detroit airport" even though the latter is more concise. We didn't want this to happen. Another issue was that we couldn't distinguish between queries which contained the same words in different orders. For example "coffee san jose" and "san jose coffee" would both have the same score even though they are different queries. We initially considered moving to a machine learning approach to solve these problems. Here is our Pugh Matrix from the end of last quarter:

Pugh Matrix:

Criteria	Importance Weighting	Manual implementation	ML
Effectiveness	5	3	4
Speed	4	3	3
Ease of implementation	2	4	3
Cost	2	5	2
Sustainability	4	2	5
Score:		53	62

Figure (3): Pugh Matrix

From our Pugh Matrix, machine learning got a higher score than manual implementation when accounting for many factors, suggesting that we should probably try an ML implementation. However, we ended up just using a commonly used technique in ML, without actually moving to ML itself. The technique is called N-grams. These are basically sets of words. Unigrams are a single word, bigrams are sets of two words, trigrams are sets of three words. By using bigrams, we were able to distinguish between queries which contained the same words in different orders. For queries with filler words, we used trigrams. For example, when calculating the score of the query “airport town of detroit”, we added up the sum of the frequencies of “airport town” and “town of detroit”. Those two phrases are very unlikely to be found in the master document, so



the query “airport town of detroit” gets a low score even though it contains many words. You can see that “of” is a filler word so a trigram is used there. Some queries are only a single word anyway, so for those words, unigrams are used. Transferring from using single word frequencies to N-grams massively improved our program, solving most of the issues we encountered previously.

Below is a snippet of the master document “local.txt”. As you can see, it contains various searches as well as the location of the user during the search, in latitude and longitude.:

```
Ruins 32.783 -96.784
direction to siteton mo 39.184 -82.509
6225 nw 48th St KCMO 64151 39.169 -94.559
Meadowlands golf club calabash 33.823 -78.659
Meadowlands golf club calabash 33.823 -78.659
Smoothie king 28.292 -81.583
Barley circlce 39.803 -77.037
Smoke 28.005 -82.73
Gas 33.861 -118.316
88 main street ashburham ma 42.33 -71.111
Chase 42.859 -85.696
Clearwater smoke shop 27.991 -82.73
Josefeen crossing 45.766 -108.581
```

Figure (4): Snippet of master document “local.txt”

Below is a snippet of the dictionary which we use to store the frequency of the different unigrams, bigrams, and trigrams taken from “local.txt”. This dictionary file is called “word.txt”.

The count of each phrase is shown below the phrase:

```

1
blirbonnet bay
1
blise id
2
blisne target
1
blisnemn
1
bliss
7
bliss army
1

```

Figure (5): Snippet of dictionary “word.txt”

Everything above only applies for non-address query sets, or query sets which contain queries which are not addresses. For addresses, we use a different procedure altogether. Instead of counting any frequencies, we measure the distance between the user’s current location and the address in question. The address is converted to a latitude and longitude, and we also keep track of the user’s location as a latitude and longitude. The distance is then calculated using the following equation:

$$a = \sin^2(\Delta\phi/2) + \cos \phi_1 \cdot \cos \phi_2 \cdot \sin^2(\Delta\lambda/2)$$

$$c = 2 * \text{atan2}(\sqrt{a}, \sqrt{1-a})$$

$$d = R * c$$

Where  $\phi$  represent the latitudes, and  $\lambda$  represent the longitudes.

Figure (6): Distance calculation formula

We use an API called OpenCage for this process.

Of course, at first, input query sets need to be classified as either addresses or non-addresses. We made an address checker that deals with this by seeing whether queries start with numbers and end with words such as “rd” or “st” etc. However, it is not working very well in its current state because many edge cases get misclassified.

### **Hardware/Software Implementation:**

Using the master document, we update a dictionary called wordCount which adapts unigram, bigram, and trigram to store the number of times each word/phrase in the document appears. We use unigram on single word, bigram on phrase, trigram on phrase containing prepositions word of the master document.

```
if (prep.contains(secondWord) && (i != wordList.length - 2)) {
    String thirdWord = wordList[i + 2];
    thirdWord = thirdWord.replaceAll(regex: " ", replacement: "");
    String combined = firstWord + " " + secondWord + " " + thirdWord;
    if (wordCount.containsKey(combined)) {
        wordCount.put(combined, (wordCount.get(combined)) + 1);
    } else {
        wordCount.put(combined, 1);
    }
    i++;
} else { //deals with queries that don't have preposition
    String combined = firstWord + " " + secondWord;
    if (wordCount.containsKey(combined)) {
        wordCount.put(combined, (wordCount.get(combined)) + 1);
    } else {
        wordCount.put(combined, 1);
    }
}
```

Figure (7): java code to get frequency of all words in the master document

For each non-address query, we iterate through all the words in the query, and look up each word in the wordCount dictionary with the same pattern we use to store words (which gram to use).

We then sum up the frequencies of all of the words. Then we order the query by such frequency

```
if (allWord.length == 1) {
    String lookup = allWord[0];
    if (dictCount.containsKey(lookup)) { // if the query is only one word, look
        frequencySum += dictCount.get(lookup);
    } else {
        dictCount.put(lookup, 0);
    }
} else {
    for (int j = 0; j < allWord.length - 1; j++) {
        String lookup;
        if (prep.contains(allWord[j + 1]) && j != allWord.length - 2) { // if
            lookup = allWord[j] + " " + allWord[j + 1] + " " + allWord[j + 2];
            j++;
        } else { //if the query doesn't contain preposition, look up the bigram
            lookup = allWord[j] + " " + allWord[j + 1];
        }
        if (dictCount.containsKey(lookup)) {
            frequencySum += dictCount.get(lookup);
        }
    }
}
frequencySumMap.put(word, frequencySum);
```

Figure (8): java code for calculating frequency score for non-address queries

Below is the code we use to get the distance from the current location to the input query location,  
if the query is an address.

```
public static double distance(double lat1, double lon1, double lat2, double lon2) {  
    if ((lat1 == lat2) && (lon1 == lon2)) {  
        return 0;  
    }  
    else {  
        double theta = lon1 - lon2;  
        double dist = Math.sin(Math.toRadians(lat1)) *  
            Math.sin(Math.toRadians(lat2)) + Math.cos(Math.toRadians(lat1)) *  
            Math.cos(Math.toRadians(lat2)) * Math.cos(Math.toRadians(theta));  
        dist = Math.acos(dist);  
        dist = Math.toDegrees(dist);  
        dist = dist * 60 * 1.1515;  
        return (dist);  
    }  
}
```

Figure (9): java code to calculate the distance for address queries

But, the code above are just interfaces, we use different interfaces for creating dictionaries and ordering queries. For creating dictionaries, we have below constructor:

```
private File inputFile;  
private File outputFile;  
  
public CreateDictionary(File inputFile, File outputFile) {  
    this.inputFile = inputFile;  
    this.outputFile = outputFile;  
}
```

Figure (10): constructor for creating dictionaries

And when we construct a CreateDictionary object with given path for input file (master file) and output file (dictionary file), we will use two methods inside to originate or do increment of our dictionary file:

```
// input is local.txt, output is word.txt
public void originateDictionary() throws IOException {
    updateFrequency(inputFile, outputFile, update: false);
}

// input is local.txt, output is word.txt
public void incrementalUpdateDictionary() throws IOException {
    updateFrequency(inputFile, outputFile, update: true);
}
```

Figure (11): methods can be called by CreateDictionary object

For the originateDictionary method, we will originate a new dictionary file, if there is one existing file, we will overwrite it. For the incrementalUpdateDictionary method, we will increase the original dictionary frequency by the new frequency.

For the interface to order query, we use following constructor:

```
private Map<String, Integer> map;

public OrderQueries(String path) throws FileNotFoundException {
    this(new File(path));
}

public OrderQueries(File dictionaryFile) throws FileNotFoundException {
    this.map = getFrequencyFile(dictionaryFile);
}
```

Figure (12): constructor of orderQuery interface

By converting our dictionary into a map, we can use the dictionary multiple times with one loading by asking the object to call methods below.

```

public String sortQueries(String[] words) {
    Map<String, Integer> dictCount = map;
    Map<String, Integer> frequencySumMap = new HashMap<>();
    for (int i = 0; i < words.length; i++) {
        String word = words[i];
        word = word.toLowerCase().trim();
        String[] allWord = word.split(regex: " ");
        int frequencySum = 0;
        if (allWord.length == 1) {
            String lookup = allWord[0];
            if (dictCount.containsKey(lookup)) { // if the query is only one
                frequencySum += dictCount.get(lookup);
            } else {
                dictCount.put(lookup, 0);
            }
        } else {
            for (int j = 0; j < allWord.length - 1; j++) {
                String lookup;
                if (prep.contains(allWord[j + 1]) && j != allWord.length - 2)

```

Figure (13): methods can be called by OrderQueries object

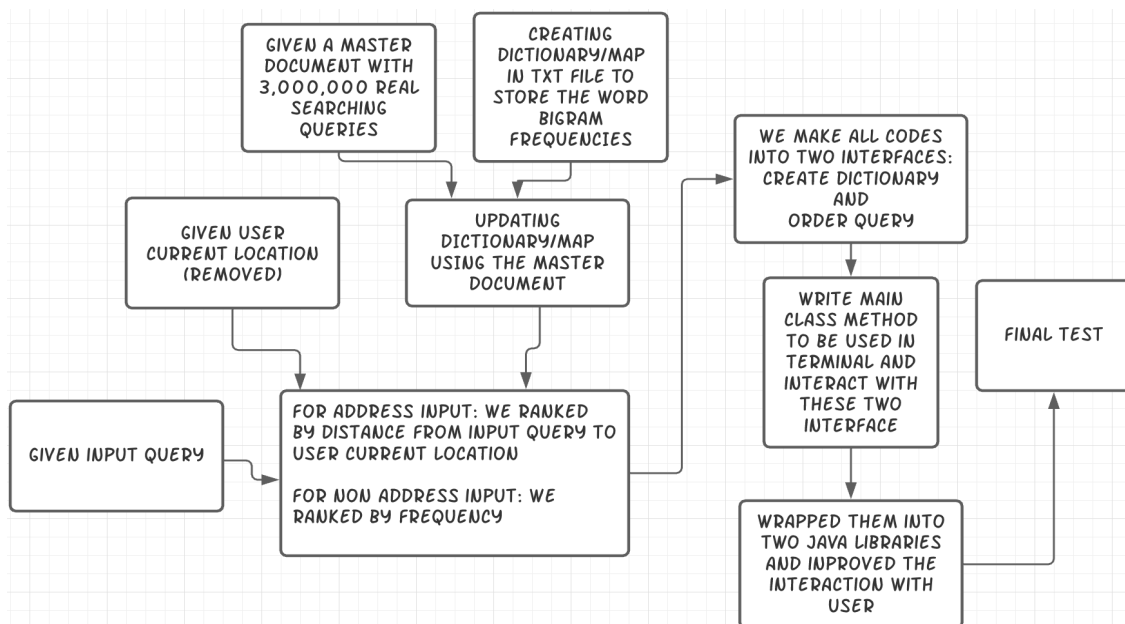


Figure (14): big-picture flowchart

Here is a basic introduction on how our algorithm performs in steps. After receiving the human voice input, the model would do the pre-processing operations to generate four most possible inputs which are relatively close to the voice input. Our task is to figure out the most likely/accurate text query from these four to meet clients' demands. Thus, a machine learning model composed of a Bayes classifier and an N-gram frequency counter would be applied to categorize and analyze all the queries generated. In this process, information regarding the text queries such as address, location name, plain words or compounded phrases is known. The N-gram algorithm is designed to count the search frequency of certain phrases such as "coffee san jose". Afterwards, we will evaluate and output according to the category of the texts in order. If all the texts are addresses, we will sort out the outcomes based on geographical coordinates and distance between the client and the destination to guess the desired answer. Otherwise, as long as all the texts are location names or merely plain words or phrases, we could assess with a criteria of searching frequency, or searching trends. The more clients search for a certain place or item recently, the higher priority the text query has on the list and meanwhile it implicates the more likely the correct answer it is. The remaining text queries are considered as compounds which consist of inter-categorical components. In this case, we will assess them based on the rule: address first, location name second, last words, and their corresponding sorting standard to rank them. Then, the first resulting output will be one the clients demand most.

### **Test/Experiment design:**

We already tested our current frequency counter program by feeding in more and more data. Initially, we fed it a list including 20 items to justify its function. After that, we gradually increased the amount of testing words up to 5,000 until we verified its capacity. We plan to test



the whole machine learning model in the same way. We may first try to manually check the outputs in a small scope and compute the accuracy. Gradually, as long as the accuracy is stable and high enough, we will be able to verify its capacity and claim its uses. As we now estimate, an accuracy of 80 percent would be expected and desired.

Then, we intend to design a simple testing Android mobile application to see how the algorithm performs in a different environment like a mobile phone. The structure would be straightforward, four lines to fill in the input text queries, one submit button to direct to the output, and a string sorting the outputs by frequency from high to low.

Finally, we would wrap the whole algorithm into a jar library which could be called in the console with command lines. We wish that users are able to create the dictionary file and seek the most likely query simply in the terminal by harnessing a jar file.

To test the functionality of the interface, we convert them into java libraries and use the command line to call them. We use a main file to implement the interaction of user and terminal.

And it looks like below:

```

if (args.length != 2) {
    System.out.println("Please give two File path with input and output");
} else {
    CreateDictionary create = new CreateDictionary(args[0], args[1]);
    print();
    Scanner sc = new Scanner(System.in);
    String userInput = sc.nextLine();
    while (userInput.length() != 0 && !userInput.trim().equals("3")) {
        if (userInput.trim().equals("2")) {
            create.originateDictionary();
        } else if (userInput.trim().equals("1")) {
            create.incrementalUpdateDictionary();
        }
        print();
        userInput = sc.nextLine();
    }
}
}

```

Figure (15): main file of using creating dictionaries

```

OrderQueries wordOrder;
if (args.length != 0) {
    wordOrder = new OrderQueries(args[0]);
} else {
    wordOrder = new OrderQueries( path: "word.txt");
}
Scanner sc = new Scanner(System.in);
System.out.println();
System.out.println("Please enter your input Strings split by |: (press enter to quit)");
String input = sc.nextLine();
while (input != null && input.length() != 0) {
    String[] test = input.split( regex: "\\|");
    System.out.println("the output is: " + wordOrder.sortQueries(test));
    System.out.println("Please enter your input Strings split by |: (press enter to quit)");
    input = sc.nextLine();
}

```

Figure (16): main file of using order queries

### Test Procedure and Test Results:

As the mentors provided us a list full of input queries and expected outputs, we implemented a testing algorithm which is capable of taking in all the text queries from the list and printing out whether the results are adhered to their expectation. We also enumerate and test a few edge cases which include all kinds of possible categories of the input text queries, and check out if the results tightly correspond to the numbers in the frequency dictionary. For example, we would test out text inputs including addresses, locations, words and even compounded words which consist of two categories of words.

We have a branch of testing cases, and we can enter them one by one in terminal or use a method that can test a branch of queries in one time and output the result in order as below:

```
Correct
detroit airport: 56
airport detroit: 7
airport town of detroit: 0

Correct
coffee san jose: 888
san jose coffee: 884

Correct
passport pizza: 1
ford pizza: 0

Correct
fort wayne: 365
ford: 130

Correct
huron national forest: 84
national florist: 0
```

Figure (17): output testing file

Also, we can test in terminal as below:

```
[linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % cd /Users/linbob/Desktop/EE497/telenavProject/out/artifacts/query_sorter_jar]
[linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % ls]
query-sorter.jar
[linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % export CLASSPATH=./Users/linbob/Desktop/EE497/telenavProject/out/artifacts/query_sorter_jar]
[linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % java -cp query-sorter.jar test.TestDictionary /Users/linbob/Desktop/EE497/telenavProject/word.txt]

Please enter your input Strings split by |:
pizza | new york pizza
the output is: new york pizza | pizza
```

Figure (18): test using command line

As you can see, both work very well, and command line testing will be mainly used by user because it is more familiar and reachable in real life.

### **Results and Analysis:**

Here is an input example, the program would like to seek which word the user wants the most. It will figure out the most likely text that the user speaks among these three.

**university | wayne state university | state university**

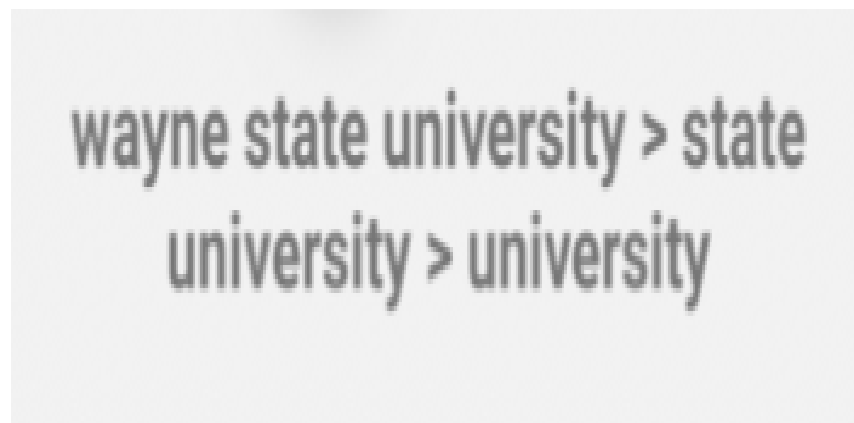
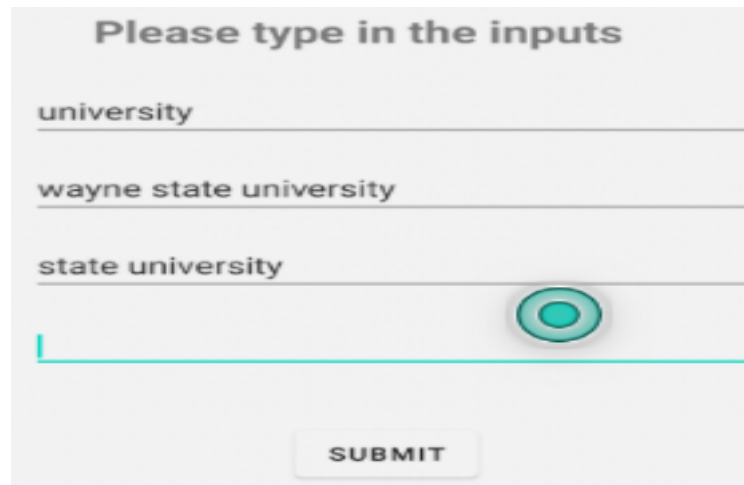
It initially looks into the dictionary file for all their scores. Wayne state university has a score equal to the sum of its two bigrams, wayne state and state university.

<b>wayne state</b>	<b>state university</b>
<b>54</b>	<b>646</b>

Therefore, wayne state university has the highest score among the three which implies the highest preference of the user. This result adheres to our expectation as users are more likely to ask for a specific location instead of general places like university.

```
wayne state university: 700  
state university: 646  
university: 8
```

In our testing Android App, typing in the inputs, it will sort them by score in a descending order.



\_\_\_\_\_Figure (19) and Figure (20): What the test app looks like

In the console, entering the inputs split by ‘|’ and running the jar file will result in the same conclusion. Here, New York pizza has a higher score than word pizza.

```

[linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % cd /Users/linbob/Desktop/EE497/telenavProject/out/artifacts/query_sorter_jar
[linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % ls
query-sorter.jar
[linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % export CLASSPATH=./Users/linbob/Desktop/EE497/telenavProject/out/artifacts/query_sorter_jar
[linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % java -cp query-sorter.jar test.TestDictionary /Users/linbob/Desktop/EE497/telenavProject/word.txt

Please enter your input Strings split by |:
pizza | new york pizza
the output is: new york pizza | pizza

```

Figure (21): Terminal running the jar file

As shown in figure in above page, most of test we got is correct, but we still have some small problems due to following reasons:

- (1) No enough words in our master document Ex:

```

Need more cases in training data to distinguish
annie's children center: 7
.....
annies children center: 7
.....

```

Figure (22): first kind of problem in output

- (2) For the words that has few frequency, we will meet 'cold start' problem and cannot recommend a proper result Ex:

```

First two order should be reverse
berkeley: 3
berkley: 2
.....
berkeley twp: 1
town of berkley: 0
.....

```

Figure (23): second kind of problem in output

To fit these two problems, I think we just need to enlarge our training dataset. With more user using this, we will definitely have a higher frequency, and thus there will be a huge difference between the hot words and cold words, and we can pick the better one for user as a result.

### **Impact and Consequences:**

As it is mentioned in the section of realistic constraints, without enough censorship of the inputs, illegal contents including sexual, violent, racial biased words may be involved, leading to possible negative social impacts and fallacies.

Besides, since the output produced is not guaranteed to be absolutely accurate, a wrong output might be fallacious and misleading so that it causes misconduct and unacceptable consequences in extreme situations. For instance, a driver intends to drive a patient to the nearest hospital urgently. Due to his strong accent, the program provides the information of a farther hospital and thus the patient misses the perfect time in the first aid treatment.

Another issue is the fact that our program takes in users' locations as inputs. One possible consequence is that these locations could get leaked and cause privacy/safety concerns. A solution to this problem is to encrypt the users' locations using a hashing method, so that there is no way to get the original input from the hashed output. This way, the data will be secure.

### **Conclusions and Recommendation:**

Our program works as intended for the most part with non-address query sets; most edges are handled correctly. Also, the user interface for the app is very intuitive. However, there are still quite a few factors that need improvements. Currently, we only have a small set of prepositions that we consider "filler" words. One possible solution is to include more words to this set, but a better way of dealing with this issue is to not hard-code the set, but rather, use machine learning to identify filler words. We can employ a common technique called TF-IDF scores to find which words are least important. We can classify words below a certain score as filler words. Another area for improvement is making the Android App more efficient. In its current state, re-loads the

same dictionary every time it runs. If we include a function for the app to support the .jar file, it can save plenty of time. The final issue, and also the most important one, is the address checker. Currently, many edges are not handled properly. Sometimes, non-addresses are classified as addresses, and sometimes, addresses are classified as non-addresses. We need to either add more conditionals to the address checker or resort to a machine learning classification method like linear regression or decision trees.

### **References, Acknowledgements, and Intellectual Property:**

#### **References:**

Licensing information for OpenCage, an API that we used for geocoding:

<https://github.com/moberwasserlechner/jopencage/blob/master/LICENSE>

We used OpenCage in order to calculate the distance between the user and a given address. This link provides a formula to calculate the distance when given longitude and latitude:

<https://www.movable-type.co.uk/scripts/latlong.html>

#### **Acknowledgements:**

Industry Mentors: Srinivasa Parvathareddy, Changzheng Jiang, Akira Zhang, Kumar Maddali

Faculty Advisors: Lillian Ratliff, Mari Ostendorf

TA: Haobo Zhang

#### **Intellectual Property:**

We haven't used any hardware or done any physical experiments for our project, and so there are no engineering standards relevant to us. We also have not infringed upon any patents as far as we know.



**Appendices:**

The whole program will be posted on our [github page](#) with [thorough instructions](#). Under the directory telenavproject/src/main/java, there are two folders containing two different instantiations. One is for dictionary creation and the other is to test its availability. Two interfaces: dictionary creator aims to create the dictionary file with all bigrams and their frequency; query sorter is able to sort the queries by frequency of words.

**Industry Mentor Comments:**

The team provided one solution that is simple and works well with testing data. Team developed Java library, command line tool and android app. We hope that they learned programming, enjoyed problem solving and collaboration with each other during this program. Appreciate each member for their hard work and contribution.