Error detection of text queries transcribed from voice input

STUDENTS: Aravind Narayanan, Zhuoran Zhou, Chingpo Lin

Problem Statement

- People use voice for in-car navigation systems. However, the system might have trouble processing what exactly the user says, depending on noise from surroundings as well as accents.
- The system will come up with a few strings for what the user could have said. For example, if the user said "university," then the system's voice processor might hear two possibilities: "university" and "universe city".
- Our goal is to rank these possible queries in order of likelihood of what the user actually meant.

Requirements

- Given a list of possible input strings, order a list of the same strings ordered correctly based on relevance.
- Eliminate errors from the input queries that were transcribed from speech (nonsensical phrase and misspelled words should be filtered out.)
- Our program should be organized into a Java library (and be able to be run through a JAR file).
- We need to create an Android test application in order to run the library.
- Finally, we need a full report and documentation detailing our entire implementation.

Implementation - Algorithm

- Going back to the example mentioned before, to us humans it is obvious that "university" should be ranked above "universe city" because "universe city" is a nonsensical phrase. However, a machine would have no way of knowing that without having some training data to work with.
- In order to train our program, we use a file called "local.txt" which includes a ton of searches by different users.
- Below is a snippet of "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 cirlce   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
```

Implementation - Algorithm (Cont'd)

- Now, how do we rank input queries when given a training data set like this? We can rank phrases by their frequency, or how often they are mentioned.
- Our algorithm counts the frequency of every bigram (set of two words) in "local.txt". Some examples of bigrams from the snippet above would be "golf club" and "club calabash".
- When prepositions like "of" or "in" are encountered, we include the word after the preposition as well. For example, we count the full phrase "town of detroit" instead of splitting it up into two bigrams. This is done to not give too much importance to common filler words.
- When full queries themselves are just a single word like "Gas" we just store that word alone.
- Our dictionary is stored as a text file called "word.txt". Here is a snippet of the dictionary. 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
```

- Once we have a dictionary of phrases, we can rank queries based on their score. If a query is longer than two words, its score is calculated by getting the sum of the counts of all of the bigrams in that query. For example, let us take the query "wayne state university".

```
wayne state  state university
54           646
```

- Above are snippets from "word.txt". You add the counts of "wayne state" and "state university" to get the score for "wayne state university". 54+646 = 700.
- Once all of the input queries have scores, they can just be ranked from highest to lowest score.
- For example, let's say the user inputs the following list of queries to be ordered:

university | wayne state university | state university

- This input list will be ordered in the following way. The score for each query is shown after the query:

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

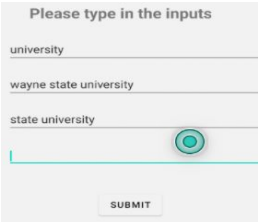
Implementation - Library and Android App

- After algorithm implementation, we build a .jar file as a library for ease of use in both Android development and command-line-execution.
- As shown below, calling the jar library through command line requires a quite a few lines of code every time.

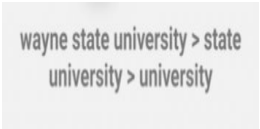
```
linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % cd /Users/linbob/Desktop/
EE497/telenavProject/out/artifacts/query_sorter_jar
linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % ls
query-sorter.jar
linbob@hayashiyasuhiros-MacBook-Pro query_sorter_jar % export CLASSPATH=.:/Users
/linbob/Desktop/EE497/telenavProject/out/artifacts/query_sorter_jar
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
```

- To simplify the user interface, we also built a simple Android App for our program.
- As shown below, the program takes a number of possible text queries as inputs and sorts them by their score.



- The input screen is shown on the left. The output screen is shown on the right.



Future Work

- The Android App should include a function to support the .jar file library so that it can save plenty of time by not re-loading the same dictionary whenever it runs.
- We need to improve our address checker, which checks whether queries are addresses or non-addresses. Currently, many edge cases are not handled.

Acknowledgements

- Industry Mentors: Srinivasa Parvathareddy, Changzheng Jiang, Akira Zhang, Kumar Maddali
- Faculty Advisors: Lillian Ratliff, Mari Ostendorf
- TA: Haobo Zhang