An Open-Source Based Speech Recognition Android Application for Helping Handicapped Students Writing Programs

Tong Lai Yu, and Santhrushna Gande
School of Computer Science and Engineering
California State University,
San Bernardino
tyu@csusb.edu, 004580341@coyote.csusb.edu

Ronald Yu
Department of Computer Science
University of Southern California,
Los Angeles
ronaldyu@usc.edu


Keywords: Speech Recognition, Android STT, Open Source, Learning Aid

Abstract

We describe in this paper how to use open-source speech recognition technologies to design and implement an Android application that helps students with physical disabilities write programs in classrooms. Google Voice Recognition (GVR)[13], which is a free and open Android tool, is utilized to convert the speech of a user to text. To fully utilize GVR, the Android phone has to be connected to the Internet.

In a typical setup, a handicapped student sits in front of a workstation with a large computer screen like the one shown in Figure 6. The student speaks to an Android phone, which converts speech to text using GVR. The text is then sent to the workstation through Wi-Fi for parsing and analysis. The processed text, which is a code segment of a program, is displayed on the workstation screen. Since the Android main activity thread cannot handle too many activities, we save the text of the speech in a buffer and use another thread, named communication thread, to send it to the workstation using the standard socket API. The producer-consumer paradigm is employed to synchronize the generation of text data by the main activity thread and the sending of the data by the communication thread[15, 18, 21].

The server program that runs on the workstation is written in C/C++. The main thread listens at a port. When it detects data, it creates two threads to handle the data. One thread created reads in the data, parses them into words, and puts the words in a circular queue. The other thread, named processing thread, simultaneously retrieves words from the queue, processes them to generate a code segment, saves the code in a file and displays it on the screen. A condition variable[7, 12] is used to synchronize the tasks between these two threads.

The keywords and symbols of the programming language that the student is using, which are saved in a file are loaded into a table. The processing thread uses a hashing and mapping scheme to obtain the proper keywords and symbols from the table; as humans often speak with inconsistency, several different words may map to the same keyword. For example, when one tries to say the word import, they may say it slightly different from the standard pronunciation and the recognizer generates the word important. The scheme will map important to the same location as import to retrieve the correct keyword.

1. Introduction

In recent years the number of mobile applications has been growing with tremendous speed. Mobile devices have become ubiquitous and in the last few years, Android, an open-source software stack for running mobile devices, has become the dominant platform of many mobile devices such as tablets and smart phones[8].

Open-source software has been playing a critical role in recent technology developments. A lot of breakthroughs in technology applications such as Watson’s Jeopardy win[4] and the phenomenal 3D movie Avatar[3] are based on open-source software. It is a significant task to explore the usage of available open-source or free tools to develop software applications for research or for commercial use[22]. We report in this paper the design and development of an Android application, based on free or open-source tools, which helps physically handicapped students write programs.

In a large university such as California State University at San Bernardino (CSUSB), which has more than 30,000 students, there is always a small but significant fraction of
students who have certain physical disabilities that make typing difficult for them. When these students take a programming course, they require special services from the institution, as they may write or type too slowly to follow the pace of the lecture. Very often, the institution offers an assistant to help a handicapped student in class, typing code segments in a workstation for the student and executing these to see how the code works. These students have to overcome tremendous physical and mental barriers to finish a degree in science or engineering that often require some programming classes. This has been the situation in CSUSB for many years. The Android mobile application presented here would alleviate the disadvantages of these students by providing tools that allow them to write programs in a workstation effectively with very little or no typing. Using this application, a student sits in front of a workstation with large screen display and speaks to an Android phone to dictate a program, which is displayed on the workstation’s monitor. (A mobile phone’s display is too small for any beginning student to learn programming on it.)

The application consists of two components: a client and a server. The client runs in an Android device, which connects to the Internet via Wi-Fi. It accepts speech from the student and converts it to text using Google Voice Recognition (GVR)\[13\], a free tool with open API. The text is sent to the server, which runs in a workstation using the free open-source Linux operating system and is provided to students to write programs in a classroom.

The server is written in C/C++. It reads in the incoming text, and makes analysis of it to form a syntactically correct program segment of a specified programming language such as Java. It then displays the code segment on the screen and saves it to a file.

In rare cases, when a workstation is not available, the text is processed by another Java server program residing in the Android device. The Java server program uses a different technique to map ambiguous speech text to more specific keywords or symbols from a pre-constructed file; it saves the code segment in a file. In this paper, we mainly describe the normal operation of the application, which displays programs on a workstation screen.

Speech recognition (SR) by machine, which translates spoken words into text has been a goal of research for more than six decades. It is also known as automatic speech recognition (ASR), computer speech recognition, or simply speech to text (STT). The research in speech recognition by machine involves a lot of disciplines, including signal processing, acoustics, pattern recognition, communication and information theory, linguistics, physiology, computer science and psychology. Figure 1 shows a general block diagram of a task-oriented speech recognition system.

![Figure 1 A Typical Speech Recognition System](image)

Nowadays, speech recognition (SR) mobile products are ubiquitous. There are many third party SR apps that support Android. We have chosen Google Voice Recognition (GVR)\[13\], which is preinstalled in many Android devices, as our recognition engine. GVR makes use of neural network algorithms to convert human audio speech to text and works for a number of major languages but we use English as our example in our description.

A neural network consists of many processors working in parallel, mimicking a virtual brain. The usage of parallel processors allows for more computing power and better operation in real-time, but what truly makes a neural network distinct is its ability to adapt and learn based on previous data. A neural network does not use one specific algorithm to achieve its task; instead it learns by the example of other data. Though GVR may work in some Android phones offline, it normally accesses through Internet its large database for voice recognition attempted by previous users. It also looks at previous Google search queries so that the voice recognition engine can guess which phrases are more commonly used than others. This way, even if the user does not speak a certain word clearly, GVR can use the context of the rest of the spoken phrase or sentence to extrapolate what the user is most likely trying to say.

In general, a neural network can learn from two major categories of learning methods—supervised or self-organized. In supervised training, an external teacher provides labeled data and the desired output. Meanwhile, self-organization network takes unlabeled data and finds groups and patterns in the data by itself. GVR learns from its own database through the self-organization method.

2. Android Threads Synchronization

The Android client involves a few tasks, including interfacing to the user, accepting text from the GVR engine, and communicating with the server. The GVR itself also has to
connect to the Internet through Wi-Fi to interact with the Google cloud database. The execution time for each task is never a constant. In particular, the bandwidth of a Wi-Fi communication can fluctuate widely, depending on the traffic of the environment. So in the application, we use two threads to handle the tasks independently so that they won’t interfere with each other. The main activity thread interacts with the user and calls the GVR engine to convert any spoken words to text and saves it as strings in a shared queue. The other thread, the communication thread, reads the strings from the queue and sends it to the server, which resides in a workstation. This is shown in the block diagram of Figure 2.

![Figure 2 Android Client](image)

To ensure that the two threads will not interfere with each other’s task, we employ the producerconsumer paradigm[15, 18], a well-studied synchronization problem in Computer Science, to synchronize the tasks between them. A classical producer-consumer problem has two threads (one called the producer, the other the consumer) sharing a common bounded buffer. The producer inserts data into the buffer, and the consumer takes the data out. In our case, the buffer is a queue where strings are entered at the tail and are read at the head. Physically, the queue is a circular queue. Logically, one can imagine it to be a linear infinite queue[21]. The head and tail pointers are always advancing (incrementing) to the right. (To access a buffer location, the pointer is always taken the mod of the physical queue length, e.g. $\text{head mod queue length}$.) If the head pointer catches up with the tail pointer (i.e. $\text{head} = \text{tail}$), the queue is empty, and the consumer must wait. If the difference between the $\text{head}$ and the $\text{tail}$ is equal to the length of the buffer, the queue is full, and the producer must wait. In the application, the main activity thread is the producer and the communication thread is the consumer. In this way, the main thread can interact with the GVR engine and the user while the communication thread is sending data at the background.

A user pushes an image button presented by the client program to start GVR. The user then speaks to the phone, which is presenting the GVR interface as shown in Figure 3. When a user speaks a sentence or a word with ambiguity, GVR may suggest up to 5 choices. Based on our experience, the first suggested one is most likely the one we want. For simplicity, the application just sends the first choice, and discards the rest. If necessary, the user can issue a discard command to the server (see description below), which discards the previous sentence, and the user can repeat the speech. Figure 4 shows the Android interface of the application and the five words, Java, Chava, tava, cava, and kava, suggested by GVR when one of the authors spoke the word ‘java’.

![Figure 3 UI of GVR](image)

![Figure 4 Android Client Interface](image)
3. The Workstation Server Threads

The text converted from speech by the Android client is sent to a server program running in a workstation placed in front of the student. The server is a multi-threaded program implemented in C/C++. Instead of using the POSIX threads, we have used the open-source cross-platform SDL threads[14], well-known by its robust characteristics, in our implementation. The SDL threads are significantly simpler than the POSIX threads but have enough features that satisfy all the requirements of our application. The C/C++ standard template library (STL) is used to facilitate the implementation.

We use the socket API function `read()` to read in the data as a stream of bytes from the network. The function `read()` is a blocking command, inhibiting the thread to proceed while it is waiting for data to come. Therefore, we create two threads to read and process the data. One thread, the reading thread, reads in the text using `read()`, obtains a word, puts it in a string buffer, which is a STL `deque` (double-sided queue), and continues to read in more text. The other thread, the processing thread, retrieves a word from the buffer and processes it. The two threads work independently and will not interfere with each other’s activities.

The synchronization between them is done using a condition variable[12], which can help solve problems that could be complicated to solve using semaphores[7]. Supported by both POSIX and SDL, a condition variable is a queue of threads (or processes) waiting for some sort of notification. A condition variable queue can only be accessed with two methods associated with its queue, typically called wait and signal. Threads wait for a guard[9] statement to become true to enter the queue and threads that change the guard from false to true could wake up the waiting threads. In practice, it always works with a mutual exclusion variable. The following code segment shows how such a variable is utilized to synchronize between the reading thread and the processing thread with some minor details omitted. In the code, the variable mutex, representing mutual exclusion, is a binary semaphore for locking and unlocking a code section, and the variable strQueue is the thread condition variable; the routine `read_data()` reads a word, puts it in the character array `a`, and returns the number of characters read.

```c
#include <SDL/SDL_thread.h>
#include <deque>
deque<string> strArray; // string buffer
SDL_mutex *mutex;
SDL_cond *strQueue;
.....
Reading_thread:
    char a[200];
    while (read_data(a) > 0) {
        string s(a);
        SDL_LockMutex(mutex);
        strArray.push_back(s);
        SDL_CondSignal(strQueue);
        SDL_UnlockMutex(mutex);
    }
```

Processing_thread:
```c
    SDL_CondWait(strQueue, mutex);
    string s = strArray.front();
    strArray.pop_front();
    SDL_LockMutex(mutex);
    strArray.push_back(s);
    SDL_CondSignal(strQueue);
    SDL_UnlockMutex(mutex);
    ..... 
```

Note that in this example, the accessing of the string buffer `strArray` is guarded by the statement `strArray.size() == 0`.

The command `SDL_CondWait()` sends the thread to sleep and releases the lock `mutex`. When it is awakened by the other thread, it will try to acquire the lock `mutex` again.

This code is significantly simpler than the circular buffer technique we used in the previous section of the Android client. The main disadvantage of this method is that it only allows one thread to access the string buffer at one time while the circular buffer allows both the producer and the consumer threads to access the buffer simultaneously as long as the head and tail do not point to the same slot. Since the server program is written in C/C++ and runs in a workstation, which has much more computing power and resource than that of a mobile phone, the technique could read and process data seamlessly and would not cause jitters in presenting the data.

4. Searching by Hashing

We use a hashing scheme to lookup keywords and commands in our applications. Hashing is a very fast searching method, with time complexity O(1). Its main disadvantage is that a table with preset size is needed to store the keys and associated data. The required table could be huge if the key space is large. However, the number of keywords in common computer languages such as Java and C/C++ is relatively very small as compared to a natural language. Therefore, in our application, hashing is an ideal candidate for looking up keywords or commands.

In our scheme, we save all the possible speech text for the keywords, which include the language keywords, symbols, and any made-up sentences, and load them into a table. Figure 5 shows a sample segment of this table. The first column shows the indices that the corresponding speech words
will map to; the second column shows the number of words that will map to the index and the third column is the keywords that will be retrieved.

<table>
<thead>
<tr>
<th>Idx</th>
<th>Count</th>
<th>Keyword</th>
<th>Speech text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>import</td>
<td>import Import important impact</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>java</td>
<td>java Java Chava tava cava kava</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>public</td>
<td>public Public puppet poppet</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>b</td>
<td>b B bee Bee be</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>.</td>
<td>dot Dot thot doct</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>=</td>
<td>equal Equal eco Eagle Eagle Ecol</td>
</tr>
</tbody>
</table>

**Figure 5** Hashing Table (Idx=Index)

For example, when the application receives any of the words, *equal, Equal, eco, Eco, eagle, Eagle, or Ecol*, the symbol “=” is retrieved. We also need another similar table that stores some commands we manually created. For example, we make the word *variable* a command. If users want to create in the program a variable called *beeb*, they have to first say *variable*, and then spell out ‘b’, ‘e’, ‘e’, ‘b’. The command *upper* capitalizes a word or a number of words. In the program, each command is handled differently so that the application knows how to process each command accordingly.

Normally, when the application receives a word, it first searches the command table. If the app cannot find the word, it is not a command. The app then searches the keyword table. If it still cannot find the word, the speaker has to say the word again. In the worst situation, the user can always use the command *variable* to create a keyword by spelling its letters out. One can also use certain commands to discard the previous word or line.

A keyword shown in the third column of the table does not need to be a real keyword or symbol of the computer language. It can be any text that would help the user in the development process. For example, it could be a statement like,

```
public static void main(String[] args) {
}
```

When a user says “main” or “Main”, the whole statement is retrieved. Similarly, the set of keywords can contain some other commonly used statements such as “for ( int i = 0; i < N; i++)” or “System.out.println()”. When saving such a keyword (which contains white spaces) in a file, one may use a special symbol like the ‘@’ character as a marker, to mark the beginning and the end of the keyword, and removes the markers when loading the keyword into a table. Alternatively, one may handle keywords with multiple spaces separately, saving them in a separate file and in a different format.

The hashing scheme handles these cases just in the same way it handles a simple short keyword. In fact, the set of keywords can be tailored for any particular programming class. For example, the instructor can provide some programming templates to students as part of the keywords. A student retrieves the templates by speaking one word to the phone.

The following code, which has omitted some minor details, shows how this hashing scheme can be implemented using C/C++, assuming that the file pointer *fpi* points to the text file that contains the data in the format shown in Figure 5.

```c++
#include <ext/hash_map>
using namespace __gnu_cxx;
const int MaxSize = 1024;
string keyWords[MaxSize];
hash_map<int, int> Map;
hash<const char*>* H;
....
int k = 1, n;
char buf[100];
int count;
//build the mapping table
while (true) {
    if (fscanf(fpi, "%d %s", &count, buf) == EOF) break;
    keyWords[k] = string(buf);
    for (int i = 0; i < count; i++) {
        fscanf(fpi, "%s", buf);
        n = H(buf); //hash a word
        Map[n] = k; //map to index,
        // starting from 1
    }
    k++;
}
```

In the code, the hash function *H* hashes a string (an array of characters) to an integer. The hash map, *Map* maps the integer to an index of the keyword table, starting from the value 1. If an integer *i* has not been mapped to an integer, the value *Map[i]* is 0. From the code we see that all the words in the same row will map to the same index *k* and the corresponding keyword is given by *keyWords[k]*. We have to use namespace *__gnu_cxx* because the standard C/C++ libraries have not implemented this hashing scheme. It is supported by the external open-source GNU libraries.

Note that by using this hashing scheme, one can easily modify the hashing table of Figure 5, which is saved in a file, to allow the user to speak in another language other than English. As long as the speech text points to the correct keyword, it always produces the same final program.
5. Results and Discussions

Figure 6 shows a typical classroom for a programming class at CSUSB. Linux workstations with large screen and Internet connection are provided to students to write programs in a class session. As shown at the lower left corner of the figure, the mobile phone lying on the table is an Android phone, which is provided to students who need it to write programs. However, it has a relatively very small screen, inappropriate for direct program development. The student must make use of the Linux workstation to help him or her to do the tasks.

The room would become too noisy if all students used speech to write their programs. However, if only one or two students who are physically handicapped use speech to write their programs, the environment works well. We have carried out experiments on this situation and found that other students are not bothered by the speech input. Moreover, the lecturer is too far from the cell phone and won’t interfere with the student’s speech input to his or her mobile phone.

In conclusion, we have developed an Android application, using open source or free tools, that assists students who are physically handicapped to write programs in a programming class. The producer-consumer paradigm is used to synchronize tasks in the client, which runs in an Android phone. On the other hand, a condition variable is used to synchronize tasks in the server, which runs in a Linux workstation. A hashing scheme is used to simplify and retrieve keywords of a language, which we have only considered for the Java programming language. Though there are still many minor details that need to be improved and fine-tuned, the application works well in a typical programming classroom. In the future, we shall extend the application to include other common languages such as C/C++ and Perl. Moreover, the data files can be easily modified to let users speak in another language such as Chinese, Korean or Japanese but still generate the same program.

References

[14] https://www.libsdl.org/


