RUTGERS

Senior Capstone Design Project

14:332:418

Open Ears Application

Project Members
Peter Lin
Stephen Shanko
Chris Slezak

Faculty Advisor
Dr. Athina Petropulu

Contents

Abstract

The Open Ears app is an initiative undertaken with the hopes of providing an easy and affordable way for pedestrians to stay safe as they listen to their music or other types of audio. For a few dollars, any Android user could download the application and instantly experience a higher level of safety with minimal effort. The app doesn't require any fancy hardware, simply plug in your microphone equipped headphones and go. The app will run in the background and listen to the microphone for you using powerful linear algebra techniques and alert you to any oncoming traffic by muting your music and vibrating your phone. The app has proven very successful in detecting cars with a false positive rate that is higher than desired. In terms of future work, a more rigorous training program for the machine learning portion of the app will make it much more accurate in detecting the difference between a noisy traffic filled environment and direct danger to the pedestrian.

Introduction

As technology becomes a greater part of our everyday lives, people are also becoming more distracted. Whether texting or listening to music, pedestrians are paying less attention to their surroundings. In 2013, 286 pedestrians were killed in automobile accidents in New York City. And that is a huge problem.

Our solution is the "Open Ears" Android App. The goal of the app is to alert the user whenever there is an approaching car. We assume the user to be wearing earphones with a built in microphone and to be listening to music on their phone. The microphone will be constantly recording, and the application will be processing the audio in real time. If a car is detected, the application will alert the user.

We used a Kernel Dictionary Learning based classifier to determine whether or not a car is present. As its input the classifier takes a vector containing six features that are extracted from the incoming audio. The algorithm outputs residuals for each classification, which would determine which class the input best matches.

Currently, we have a running Android application, which can distinguish between the silence and an approaching car. In the future, we would like for the user to be able to incorporate his or her own data into the app. They would be able to train on data they supply and teach the app to recognize any sounds they would like to be alerted to.

System Overview

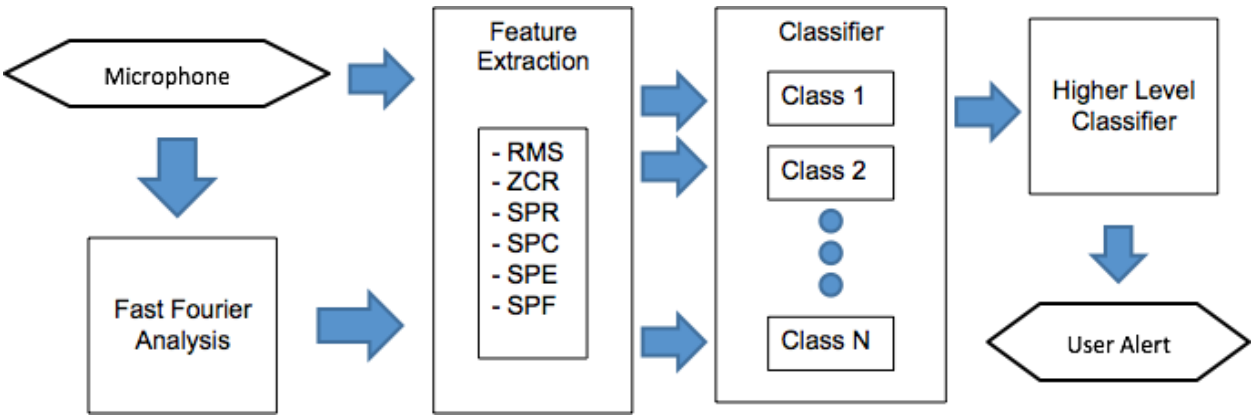

Figure 1. Block diagram for the Android app

When running, the app is constantly sampling from the microphone. Six features are extracted from each frame of audio. The features are then compiled in a six-dimensional vector, where each element is a result of one of the feature calculations for that frame. The feature vector is then used as input for a Kernel Dictionary Learning classifier. The output of the classifier can then be used by the app to determine which class (approaching car, background noise, etc.) the frame belongs to. A higher level classifier is used to average the results from several frames and

provide smoothing. Once the higher level classifier determines that a match has been made the user is alerted.

Phase 1 Algorithm Development

Feature Selection and Extraction

The audio features used in our app were selected based on their successful use in a similar system developed by Smaldone et al. (2010). They made use of six features total, two in the time domain and four in the frequency domain. The time domain features are zero crossing rate and root mean square energy. The frequency domain features are spectral roll-off, spectral centroid, spectral entropy, and spectral flux.

Audio Feature	Symbol	Domain	Description
Spectral Entropy	SPE	Freq	The entropy of the spectrum. Falls off sharply when there is a vehicle approaching.
Spectral Centroid	SPC	Freq	Weighted center of the spectrum. Increases when there is a vehicle approaching.
Root-Mean-Square Amplitude	RMS	Time	The per-frame audio energy. Used to drop frames to ignore irrelevant sounds, e.g., a parked bike. Vehicle approaches are often louder than other audio frames.
Zero Crossing Rate	ZCR	Time	Number of times the audio signal crosses zero within a frame. Typically higher for random noise.
Spectral Rolloff	SPR	Freq	Frequency bin below which 91% of the signal energy is contained. Increases slightly during vehicle approaches.
Spectral Flux	SPF	Freq	Relative change in the frequency weighted by magnitude. Exhibits a small increase at the onset of a vehicle approach.

Figure 2. Table of features from Smaldone et al. (2010)

Classifier

Our app made use of a Kernel Dictionary Learning classifier outlined by Nguyen et al. (2012). This classifier exploits sparsity in a signal, which allows a signal to be represented using a small number of atoms selected from a dictionary. It also makes use a of kernel function which projects the six-dimensional vectors obtained through feature extraction into a space of much higher dimension. This provides a greater degree of separability between classes.

Training

Before any classification can take place, the classifier must be trained. We begin training by selecting which classes of signals we would like to identify. During some of our trials for instance, we chose approaching car, background noise, and conversation. These classes were selected with the intention of alerting the user that a car is approaching while preventing a false alarm if the user is having a conversation with a friend or simply walking along a quiet road. Audio data that represents each of the classes can be obtained using the recording functionality built into the app. Care must be taken to ensure that the training data is as exhaustive as possible. As an example, training on data that only includes small cars may cause classification problems when a large truck is approaching.

Learning

Once the audio files that will be used for training have been recorded they can imported into MATLAB for the dictionary learning process. The first stage of this process requires the training data to be segmented into frames of a fixed length. We then run the feature extractor on each frame to transform it into a six dimensional feature vector. We then take all the vectors of a single class and construct a dictionary from them, so that with N classes we will have N dictionaries. A dictionary composed of all these vectors would be much too large, so a random permutation of all these vectors is taken and used to form the base dictionary. The atom representation dictionary is then learned using the KOMP procedure detailed by Nguyen et al. (2012). Together the base dictionary and the atom representation dictionary form a complete dictionary that can be used for classification.

Classification

Once the dictionaries for each class have been learned, they can be transferred to the phone so the app can use them for classification. When the app has a feature vector that is ready to be classified, it will reconstruct the input vector using each learned dictionary. The class whose dictionary produces the smallest residual (reconstruction error) is the class to which the input belongs. The higher order classifier will look at the reconstruction errors for each class across multiple frames and make the final decision to alert the user.

Selection of Classifier Parameters

The system had a number of parameters that could be tuned to increase performance. Incorrectly choosing these parameters could cause the system to function poorly or fail entirely. One of the first choices was the selection of a kernel function. The kernel function allows the classifier to exploit nonlinear structure in the data and thus is a fundamental part of the algorithm. Two kernel functions often used in machine learning applications are the Gaussian and polynomial kernels. We found that a Gaussian kernel with sigma equal to 1000 produced very good results, better than a polynomial kernel of any degree we tried. Polynomial kernels with too high of a degree caused the algorithm to fail and therefore were not suitable.

The next choice faced was the selection of a sparsity level. As a rule the sparsity level should be smaller than the dimensionality of the feature vectors, and we found that a sparsity level of 2 worked well for our data. The final choice we faced as far as the classifier was concerned was the number of atoms to include in the base dictionary. Each additional atom increases the amount of computation during classification time, but too few and our dictionary may not be representative of the entire class. This value was determined experimentally and eventually a balance was struck at 40.

Results

To calculate the Receiver Operating Characteristic, test and training data was recorded while walking along River Road and Hoes Lane West. A total of 51.8 seconds of training data was recorded that represented approaching cars, conversation, and background noise. 65 seconds of audio was used for testing. A true positive was when an approaching car or an approaching car with conversation in the background was successfully detected. A false positive consisted of background noise or conversation alone that was detected as a car. The sensitivity of the receiver was varied by multiplying the residual for the car class by a constant. A small number corresponded to low selectivity and a larger number high selectivity.

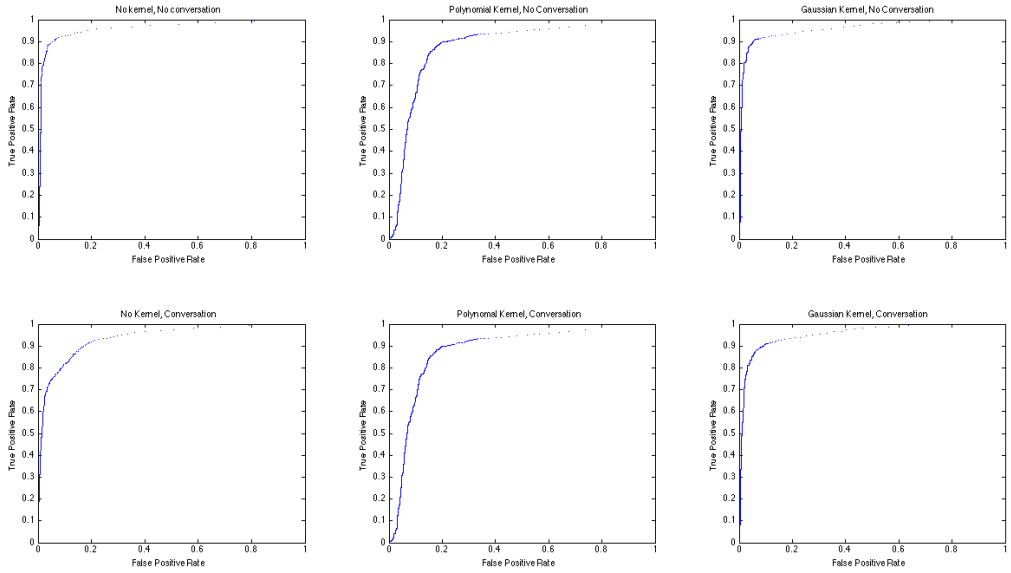

Figure 3. ROC curves

The above figure shows ROC curves for various kernel functions and in different test scenarios. In the “conversation” scenarios the test data included approaching cars with conversation in the background. These curves show that the use of a Gaussian kernel made the classifier more robust when challenging situations like this were present.

Phase 2 Android Application Development

App Organization

The android application for the Open Ears pedestrian safety system has been developed in such a way that it requires minimal user interaction and oversight in order to operate successfully. The user interface that is currently implemented is used for debugging purposes only. It will display the word Car in red letters when a car is detected. Additionally, as part of user interaction it will also mute the audio stream and vibrate the phone when an approaching car is detected. It also provides the user with an option of recording up to 2 minutes of .wav audio and the option of disabling the thread that is responsible for monitoring the microphone input stream.

The android application is comprised of 4 java classes in total, of which one is an Android Activity class and another extends the class Thread. The classes are named as follows, MainActivity, Audio, Classify, and WavProcessing. Following are brief descriptions of each class, their functionality, and a block diagram of the whole system.

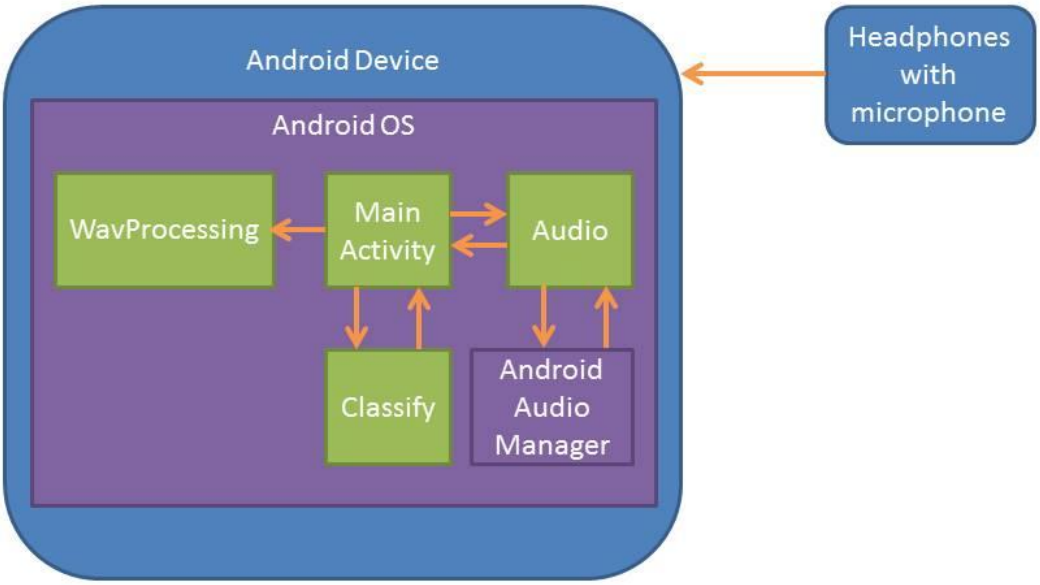

Figure 4. Android application block diagram and organization

MainActivity

This class is at the heart of the app as pictured in the block diagram. It is the only class that extends Activity, and therefore the only class that has a UI, in the entire app. The MainActivity receives data from the Audio thread and the Classify class. It also sends data to the WavProcessing class in the event that the user decides to record a wav file from the microphone input stream. Upon launching the app, an instance of the Audio thread is automatically started, and a message handler is registered to the MainActivity class. Whenever data is read from the buffer in Audio, it is passed to the message handler. The message handler then decides what

type of data is arriving (time domain, spectral, or wav writing) and acts accordingly. Within the message handler, the 6 key features for the classification algorithm are calculated, and the higher order classifier is implemented in order to keep the rate of update to the UI reasonable for a more comfortable level of usability. After receiving time domain and spectral data for a given frame of data, the MainActivity is responsible for passing that information to the Classify class in order to be compared against the dictionaries that we have complied previously.

Audio

The audio class is responsible for asynchronously gathering data from the Android AudioManager and sending messages to the MainActivity every time a full frame of data is recovered. A full frame is determined by the equality frameSize = samplingRate * .025. In other words depending on the sampling rate, the number of samples per frame will generally yield 25 milliseconds of audio. This amount of audio is found to be sufficient for a good match across the frequencies we are analyzing. The audio data coming from AudioManager is a 16bit PCM encoded stream that is received natively as a short data element. This is then converted to a normalized double that ranges in value between -1 and 1. The time domain sequence is then run through a Hamming window, a complex DFT is performed on the windowed sequence, and then the real parts of the DFT are obtained. These two double arrays, time and spectral, are then sent to the MainActivity for classification.

Classify

The Classify class is responsible for the implementation of the Kernel Orthogonal Matching Pursuit (KOMP) algorithm. Within the initializer, we read the dictionary data generated in MATLAB and packaged into a .dat file. Each instance of the Classify class uses a different dictionary based on the sounds classes we wish to identify. For example, we might have a car

class, a horn class, a police siren class, and an ambient noise class. In the aforementioned instance, we would like to alert the user to danger in any of the cases except the case of ambient noise. The Classify instance accepts an argument to define which dictionaries to read, as well as the double array of features that have been extracted in the message handler in the MainActivity. The KOMP algorithm, implemented using the JAMA matrix library, is a linear algebra intensive algorithm that is able to compare the current feature vector against the particular dictionary data and return a residual that represents the degree of similarity between the current audio frame and the frames identified as cars in the training data. The residual is returned to the MainActivity for analysis by the higher order classifier.

WavProcessing

The WavProcessing class was designed to gain a high level of certainty and customization for the training data collected by the user. In early attempts, we used third party apps to record uncompressed wav files for the purpose of training our algorithm on data collected in the field by the same smartphones that would ultimately run the classification. However, to our surprise we found that when Open Ears was run using the dictionaries obtained from said training data, the results were poor. After careful analysis of controlled and repeatable audio signals, we determined through the aid of .dat files and MATLAB that the data being collected by these third party apps was different than the data that came directly from the Android AudioManager. Our conclusion is that effects such as automatic gain control, audio compression, and signal smoothing were implemented in third party apps to obtain a more pleasant sounding audio track. However, for our purposes a computer is just as willing to listen to a highly dynamic signal as it is to listen to a nicely compressed and evenly loud signal. Our solution to this problem was the WavProcessing class. Based on the documentation for the wav file header and wav file data

format, we were able to construct the class so that it wrote the data directly from our short[] buffer directly to hex values in the wav file. By using a random access file writer in Java we were able to write the data into the file first, and then complete the header before finalizing the file. This way, we had very precise control over the entire process from microphone to MATLAB. Once this class was created, we were able to record our own training data from directly inside the app. Using dictionaries created from this data we found that the ability of Open Ears to detect oncoming vehicles and other sounds accurately was greatly increased.

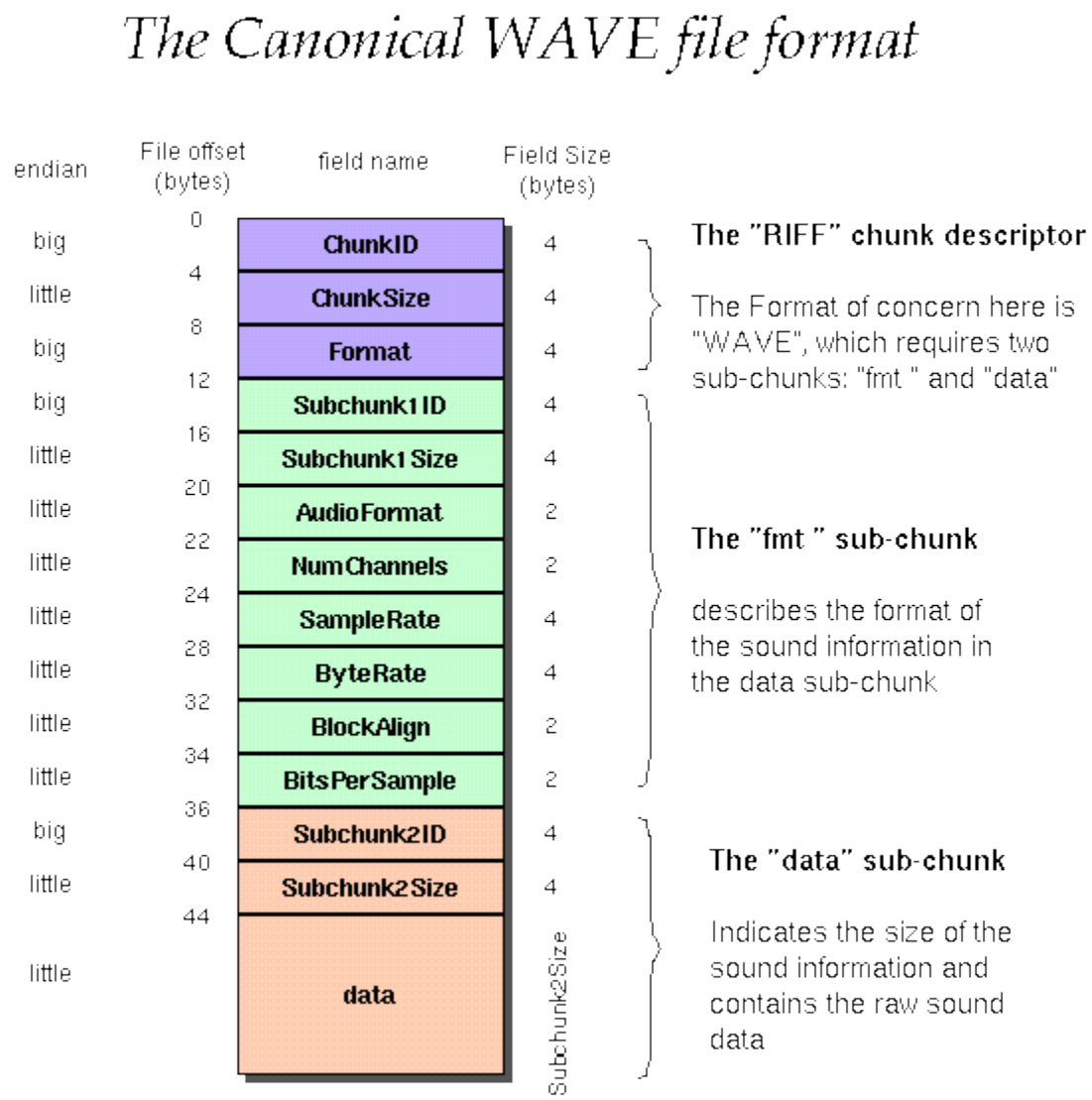

Figure 5. WAVE file format taken from Microsoft RIFF specification

Future Work

The app was tested under realistic traffic conditions on Hoes Lane West, bordering the west side of Busch campus. Due to the traffic conditions on this road, it was possible to observe single cars passing with few other sources to cause false positive or negative results. Based upon our training data, the car class is built upon the unique sound of road noise. There are certain advantages and disadvantages to this approach. One advantage is that the speed of the vehicle and sound of the engine are somewhat less influential in our classification of an approaching car. However, to our disadvantage classification based on road noise is very susceptible to error in cases where there are very high traffic areas and road noise is constantly present. Even on Hoes Lane West, we observed that as cars passed by, the sound of road noise was audible for many seconds after the cars had passed by. In the future, there are two ideas which if implemented could greatly increase the viability of this app as a safety feature. To increase the robustness of the algorithm, data of more vehicles must be collected in order to reliably detect vehicles of different kinds. Notable examples of vehicles that are currently hard to detect with our limited data are motorcycles and heavy trucks. Also, careful consideration of what constitutes an approaching car is needed to improve the detection rate of the app. A more refined approach based on factors such as speed of approach, geo-location, engine noise, brake squealing, and/or horn sounds could dramatically increase the practicality of this app, helping us to eliminate the problem of false positives induced by lingering road noise.

Results

During the testing phase, we observed 21 cars passing by us on Hoes Lane West. We were sometimes walking and sometimes stationary during the trials. We were able to successfully detect 20 of the 21 cars using the Open Ears app. This yields a success rate of just over 95%.

However, the price paid for such a high rate of successful detections came with the rate of false positives. On 8 occasions we detected cars when there were none that had passed by recently. As we were walking along the road, the reason for this rate of false positives was not always clear, but on some occasions we did hear the sound of road noise and vehicles in a way that would also be perceived by the human ear as the sound of a car. For this reason we believe that the app is indeed accurate, but needs more refinement in some of the ways listed above to become more discerning as a human being might be.

Cost/ Sustainability Analysis

The software that our team used included MATLAB, Java, and the Android SDK libraries. Java and all Java libraries are open source and all freely available. However, a license of MATLAB was required , priced at $149 for personal use. It was assumed that at least 1 developer on the team had an Android device, that could be used for testing. Otherwise, no other physical materials were used.

This project was a semester long project, which lasted approximately 5 months. Our team consisted of three developers. Assuming we were paid, a fair wage would have been approximately $15/ hour. If we considered that the developers worked part time, for approximately 15 hours a week, for 5 months, the total wage one developer would have been $4500. The total wage for three developers would have been $13,500 for the length of the project.

We can estimate that around 56% of the total funding will go into overhead costs, back into Rutgers to pay for the use of their facilities. We can also estimate that 30% of the total funding will go into benefits, including insurance. This means that a total of 14% of the original

funding will go to wages. This means that project would have required a total of approximately $97,000 in funding.

Even though we received no funding , we were able to create a working product purely from open-source software. Because we already had access to MATLAB from Rutgers, we were able to create an Android App at no cost. If we were to release our App onto the Google Play Store, a fair selling price would be at $1 per download. In the future, we could add additional features, and charge a premium for them.

Works Cited

Nguyen, H.V.; Patel, V.M.; Nasrabadi, N.M.; Chellappa, R., "Kernel dictionary learning," Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on , vol., no., pp.2021,2024, 25-30 March 2012

Smaldone, Stephen, et al. "Improving Bicycle Safety through Automated Real-Time Vehicle Detection." (2010)