Discriminative Speaker Verification System

Mohammad Afshar (ma2510@nyu.edu)
Martha Poole (mep505@nyu.edu)

Abstract. This project attempts to create a speaker verification system using machine learning techniques. The goal of the system is to discern whether an individual's identity claim is valid, based on audio of their voice. A successful system should, with minimal error, verify an individual's identity claim or alternatively assert that the claim is false, that the individual is in fact an impostor.

1. Introduction and Motivation. This project attempts to implement a speaker verification system. A speaker verification system validates an identity claim for a given individual, based on a short audio sampling of their speech. Such a system has become increasingly useful in real world situations given the ever increasing computing power in phones, watches, and other everyday commodities (the "internet of things"). A speaker verification system could control access to these devices in a way that is seamless and natural to the user.

The machine learning task of verifying a speaker's identity has historically been implemented using a generative approach with Gaussian Mixture Models, as well as the discriminative approach of Sup- port Vector Machines. It has also been demonstrated that a simple Nearest Neighbors algorithm performs close to optimally. Applying transformations to the data using theoretical and heuristic filters and kernels is key to optimal performance.

2. Performance Criteria. We borrow our performance criteria from the paper *Kernel Based Text-Independent Speaker Verification* by Mariéthoz, Grandvalet, and Bengio. As the authors point out, a simple classification error rate does not adequately express loss in the system, since it does not account for the differing repercussions of a false acceptance versus a false rejection. Instead, the authors present the Half Total Error Rate (HTER) equation as the performance criteria for the test set

$$\Delta^* = \underset{\Delta}{argmin}(\alpha \cdot FAR(\Delta) + (1-\alpha) \cdot FAR(\Delta))$$

FAR is the False Acceptance Rate (the number of false acceptances over client accesses). FRR is the False Rejection Rate (the number of false rejections over the number of impostor accesses). is the deci- sion threshold, which is previously determined based on the training data.

3. Problem Formulation. In the paper by by Mariéthoz, Grandvalet, and Bengio, the problem is formulated as a binary classification problem: whether to accept or reject an individual's identity claim. The problem is thus defined as:

$$f(\bar{x}; V_i) > \Delta$$

Put plainly, we classify a sentence $\bar{x}$ as being said by the speaker if the discriminant function is greater than some previously defined decision threshold. V_i is a parameter that minimizes the loss on the training set. It is defined as:

$$V_i = \underset{\theta}{argmin} \sum_{(\bar{x},y)\epsilon\mathcal{L}_i} \ell(f(\bar{x};\theta), y)$$

Where y is the classification 1 or -1, $\bar{x}$ is the sentence that is our audio sample, and ℓ is our loss function.

4. Algorithm. In the paper, the authors discuss the Gaussian Mixture Model (with regard to the generative approach) and the Support Vector Machine and K Nearest Neighbors (with regard to the discriminative approach). The KNN and SVM performance do not differ by a large margin; they both perform well in minimizing the loss function, while the GMM yielded the worst performance. GMMs were very popular when solving speaker verification problems, but soon were overshadowed by more kernel-based methods, specifically using Fisher Kernels (to be able to specify similarity between pairs of examples). Prior to using a SVM, the dimensions of the target subspace must be observed and controlled through cross-validation. This will allow the model to avoid obtaining the very obvious bad solution; then the SVM is trained for each new client in the database. It is important to note that this method had very high performance when tested against the NIST dataset.

Preprocessing of the data was done using band-pass filtering using a discrete cosine transform (DCT) complimented with delta and delta-delta signals.

5. Baseline Method. We initially set our baseline to reproduce the results outlined in *Kernel Based Text-Independent Speaker Verification.* The baseline method for this project will be to have similar performance compared to the implementation of the engineers in the article. This seemed like a feasible task, as the software is freely available for download online. The NIST dataset, however, was not available. We emailed Samy Bengio, one of the authors of the paper, and he replied that he no longer had access to it. So, we attempted to produce results for a new dataset, using the same software.

Furthermore, we set the baseline as the goal of our own project - our attempt at approaching and potentially improving upon the space and time complexities of the algorithm the engineers in the article have obtained. However, the engineers used C++ as the language of focus with occasional usage of Python alongside various other machine learning frameworks, we intend to strictly use Python, scikit-learn, matplotlib and various other Pythonic machine learning libraries and packages.

6. Software. The software that the authors used in all experiments in the paper are freely available online. The software provided was comprised of shell and python scripts and C++ programs, and relied heavily on a handful of external libraries for scientific computing, linear algebra, signal processing, and machine learning algorithms. It implements the GMM, SVM and KNN text-independent speaker verification over a certain database. The software also relies on Torch, which provides a framework that is used. Requirements for the software based upon the authors' notes was a machine running Linux (or Mac OS, but untested) with Python, matplotlib, and zsh installed. While the documentation indicated that these libraries and programs were self-contained and could be installed in just several steps, we ran into many problems from the beginning. Though not mentioned in the documentation, the software configuration was finely tuned to only work on 32-bit architecture. We attempted several workarounds and installations on four separate machines before we were able to successfully compile. In the end, the best virtual machine that worked very well with the libraries and packages was 32-bit Lubuntu 14.4.3.

There were also several required libraries and packages aside from SVDLIBC, KISS_FFT, Torch3, and PyVerif that were not included in the documentation that we had to seek out and install after seeing compilation error messages. After several days effort we prevailed in installing the software. However, major modifications to scripts were required in order for us to process data other than the NIST dataset used by the authors. Since time did not allow for us to make these modifications, we attempted to run an initial script that transformed wav files into MATLAB files containing the computed LFCC (Linear Frequency Cepstral Coefficients) using the PyVerif library. We were able to succeed in this, though additional library installations and changes to the software directory structure were required.

7. Real-World Dataset. The dataset that was used in the paper is the NIST dataset. NIST has been working with Speaker Recognition Evaluations in order to promote the research and collaboration of text independent speaker recognition. It is well structured and has a .sph file extension. The scripts used by the authors converts this to a .wav file format, and then calculates the LFCC of each signal. Although there is no way of accessing the dataset used in the experiment due to New York University not being affiliated with NIST, a paper published by Tel Aviv University's Computer Science department outlines the data they used in their experiments regarding speaker verification. This consisted of using voice samples from popular animated sitcoms such as The Simpsons and Futurama, keeping the sampling rate at 48kHz so that no information is lost. The size of this dataset is 5GB.

However, due to the difference in the datasets used by Mariethoz and his team, all the scripts that were written were solely based on the format and structure of the NIST dataset; there are three different input directories, and the paths are passed in as input to the program, which then creates the output directory with all relevant output files. These scripts were also reliant upon the separation of the data into various different categories.

On the other hand, The Simpsons dataset is a labeled dataset that comes in a .wav file format. The subtitles are matched to the transcript lines of the individual characters. Characters that had more than one actor voicing them were eliminated from this dataset, with a certain exception made to characters that made significant appearances and were played by a single voice actor.

Additionally, the dataset had to be processed further. The engineers at Tel Aviv University identified and retrieved these utterances, and then filtered them to get rid of any white noise – there was a total of 29,733 utterances, with a total time length of 18.9 hours, and utterances an average of 2.29 seconds long. Then, spectogram frames using the sliding window method were generated with SoX for every utterance; the window width is 20ms and the step is at 10ms. Spectograms with different shapes are not used.

8. Evaluation. We evaluated the baseline method by running the LFCC script by Mariethoz and his team on The Simpsons dataset. This was tricky as it required us to modify the Python script in such a way that it would allows us to input a .wav file and get the LFCC for each signal. After modifying the code multiple times on multiple virtual machines, we were able to successfully produce the .mat files for the .wav signals. We had successfully accomplished the portion of the preprocessing that Mariethoz and his team accomplished before running it through the actual machine learning algorithms.

9. K-Means Clustering Optimization. In unsupervised learning, labels in the dataset are missing, which makes it more challenging to come up with a system that predicts the outcome of an unknown input value. However, clustering is a commonly used technique to understand more about the dataset that you have at hand – it is often used prior to running the data through a learning model. In particular, we used the K-Means clustering algorithm to in our dataset to understand more about the feature vectors that we were getting.

Initially, we first implemented K-Means using a toy dataset, where each of the 50 points was a 5-D vector. However, this implementation was very naive and randomized the centroid only once.

An upgrade to this implementation was another version of K-Means that randomized the centroids, but also remembered the best centroid that had the smallest minimum distance for given points. In other words, it was a significant improvement to the naive K-Means, running the algorithm an additional number of times.

An even better improvement to this algorithm was to use the K-Means technique introduced in a paper by David Arthur and Sergei Vassilvitskii, known as K-Means++. This algorithm is an elegant solution to the ungraceful improvement to naive K-Means. It selects a point from the input data set, chosen uniformly at random. It selects another point from the input data set with probability D^2; it then repeats this for k centroids. Then it runs regular K-Means on these centroids.

The scitkit-learn implementation of K-Means is also leaning on the algorithm defined by Arthur and Vassilivitskii. However, scikit's code has been optimized to take advantage of Python's many available libraries and packages including NumPy and SciPy. K-Means++ allows for a smarter way of selecting the centroids in order for the clustering algorithm to converge faster. I observed the K-Means++ algorithm implemented by scikit, and utilized many of their optimization techniques in our code after first running it.

For example, the most apparent technique that optimized performance was obtaining the distance between two points in Euclidean space. There are three possible functions that I used, each of which led to a better performance in running time compared to the naive way that I had implemented with the *math.sqrt()* and *math.pow()* functions in the math package of Python stdlib. Given two points, $P(x_{i1}, x_{i2}, ..., x_{in})$ and $Q(x_{j1}, x_{j2}, ..., x_{jn})$, then the Euclidean distance between the two points can be calculated using three distinct optimization functions, namely; using scipy: *from scipy.spatial import distance* and then *distance.euclidean(P, Q)*; using numpy: *import numpy as np* and then followed by either one of these function calls: *np.linalg.norm(P-Q)*, *np.sqrt(np.sum((P-Q)**2))*.

Another optimization technique that is commonly used in Python code (and follows Pythonic coding structures) is the use of vectorized and single line code consolidations in order to do multiple lines of computation. This also includes a programming paradigm I am very unfamiliar with called lambda architecture, where instead of defining a function to complete a task and return a value, you pass in a lambda expression that computes what you need on the spot. With regard to short-hand vectorization, for example, the *min()* function can also be utilized to calculate the best distance in a vector of distances between a point and all k centroids.

The difference in time complexity between the improvements and optimizations made and the initial ways of solving it was apparent by at least 3 seconds.

10. First Approach Optimization. Equipped with knowledge of K-Means clustering, we tackled the problem of feature extraction and first-approach implementation of our project. We are using the Simpsons/Futurama dataset. We took 20 signals, which were in .wav files, and represented them as audio time-series. We then took each signal and calculated the Mel Frequency Cepstral Coefficients (MFCC) for each of the signals. The default number of MFCC windows was 20, and the sampling rate we used was 16 kHz. We only focused on one feature in this first approach, but for further optimization, we intend you use multiple features, e.g. LFCC, STFT, etc. Our *learnVocabulary()* function took a path to a directory containing signals to be clustered, and we then took the MFCC of each signal – each MFCC matrix was a matrix of 20 points. The only issue that we ran into was that, for each point in the MFCC matrix of a given signal, the points were in different dimensions, ranging from 9 to 153. Therefore, we were unable to test our code further. Then, the *getbof()* function took a path to a directory of input signals, and then applied the *learnVocabulary()* function to each new signal. However, our implementation also consisted of creating a K-Means objective function, which calculated the average distance of a point from the dataset to it's centroid and returned this value – therefore, the best clustering was the one that returned the smallest value of the objective function. This was our implementation of GridSearch; this initial implementation consisted of clusters from 1-10.

Furthermore, the library we used for building most of the signal processing aspects of our pipeline was librosa. However, librosa uses *scipy.signal* as the default package for resampling audio signals, which takes a very long amount of time. It was recommended by the developers of librosa to install *librsamplerate* along with the connecting module, *scikits.samplerate*. However, these two modules had multiple dependencies, and each of those libraries was tedious to symlink due to problems with documentation. The libraries that depended on one another were *flac*, *libogg*, *libvorbis*, and *libsndfile*. Once these libraries and packages were installed and individually symlinked, then the *scikits.samplerate* was installed through the online link; it is important to note that installing the library using *pip* led to compilation errors in Python code. After installation, performance of audio loading, MFCC calculation, spectrogram analysis (for simply observing the MFCC's) had a noticeably reduced running time, averaging between 4-5 seconds per signal-load and MFCC computation.

REFERENCES

[1] Mariethoz J, Grandvalet Y, and Bengio S. 2009, *Kernel Based Text-Independent Speaker Verification*, IDIAP Research Institute, pp197-223

[2] Uzan L, Wof, L 2015, *I Know That Voice: Identifying the Voice Actor Behind the Voice*, 2015 International Conference on Biometrics, pp 46-51