

# Version Control of Speaker Recognition Systems

Quan Wang      Ignacio Lopez Moreno

Google

quanw@google.com

elnota@google.com

## Abstract

This paper discusses one of the most challenging practical engineering problems in speaker recognition systems — the version control of models and user profiles. A typical speaker recognition system consists of two stages: the enrollment stage, where a profile is generated from user-provided enrollment audio; and the runtime stage, where the voice identity of the runtime audio is compared against the stored profiles. As technology advances, the speaker recognition system needs to be updated for better performance. However, if the stored user profiles are not updated accordingly, version mismatch will result in meaningless recognition results. In this paper, we describe different version control strategies for different types of speaker recognition systems, according to how they are deployed in the production environment.

**Index Terms:** speaker recognition, version control, deployment, production

## 1. Introduction

*Speaker recognition* is the process of recognizing the personal identity of a spoken utterance. Depending on the number of speaker candidates to be recognized, it is often referred to as *speaker verification* (single candidate) or *speaker identification* (multiple candidates). According to the textual content of the spoken utterance being recognized, a speaker recognition task falls into three categories: *text-dependent* speaker recognition [1], where the text of the utterance is always the same (e.g. a keyword [2] or password), or from a very small set; *text-prompted* speaker recognition, where the text of the utterance is randomly selected from a pre-defined large set to prevent spoofing attacks; and *text-independent* speaker recognition [3], where there is no restriction on the text of the utterance.

Regardless of the number of speaker candidates, the text of utterance, or the specific underlying technology, all speaker recognition systems require two stages of user interaction when deployed to production environment: the *enrollment stage* and the *runtime stage*:

- During the enrollment stage, a user provides multiple audio samples to the system, and the system generates a *user profile* to represent the voice characteristics of this user, as shown in Fig. 1.
- Once the users have completed the enrollment, the system is ready for runtime recognition, where the voice characteristics of the runtime audio is compared against the enrolled user profiles, as shown in Fig. 2.

In both the enrollment stage and the runtime stage, the speaker recognition system needs to extract acoustic features such as PLP [4], MFCC [5], PNCC [6] or log Mel-filterbanks from the audio signals. After the acoustic features have been extracted, a speaker encoder model will be used to represent the audio by a speaker embedding, such as a GMM supervector [7],

speaker factors from joint factor analysis [8], an i-vector [9], or a neural network embedding [10, 11, 12]. In the context of this paper, the software that implements feature extraction and speaker encoder will be referred to as the *speech engine*, as they are the most computationally expensive components in the speaker recognition system.

## 2. The version control problem

After a speaker recognition system has been deployed to production environment, we may still want to update the system for many reasons, including:

1. Updating the feature extraction component for better performance (e.g. using more frequency bands).
2. Updating the underlying speaker encoder technology for better performance (e.g. migrating from i-vector model to neural network based model).
3. Based on the same technology, updating the speaker encoder model to use a different neural network topology, a different loss function during training, or different training datasets.
4. Software optimization and refactoring to improve system robustness, scalability and maintainability.

Because of the enrollment stage, speaker recognition is a **stateful** system — the recognition result of a runtime audio depends on the output of other audio (i.e. the enrollment audio). This is very different from other speech systems such as automatic speech recognition (ASR) and language recognition, where the systems are typically stateless.

As a consequence, the user profiles obtained during the enrollment process (Fig. 1) are “version dependent”. Once the speaker recognition system has been updated to a newer version, existing user profiles can no longer be used.

In the following sections, we will discuss strategies to re-enroll the user profiles based on a new version of the system in a production environment<sup>1</sup>. The strategies are different based on the type of deployment. According to where the speech engine runs and where the user profiles are stored, we categorize deployment solutions into three types:

1. **Device-side deployment:** The speech engine runs on user devices, and the user profiles are also stored on user devices.
2. **Server-side deployment:** The speech engine runs on cloud computing servers, and the user profiles are stored on cloud databases.
3. **Hybrid deployment:** The speech engine runs on cloud computing servers, but the user profiles are stored on user devices.

<sup>1</sup>Without loss of generality, we will refer to the new version of the system as the new “model” in the following sections for simplicity.

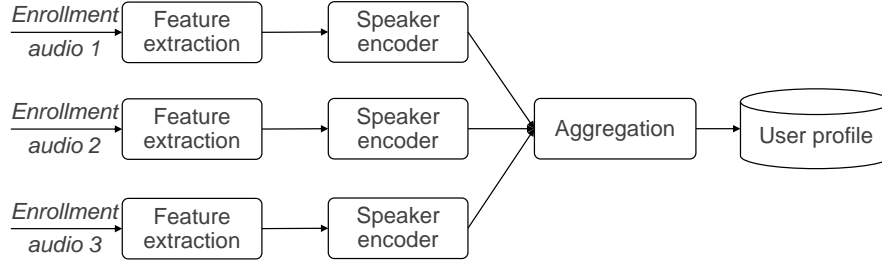


Figure 1: Workflow of the enrollment stage of a speaker recognition system.

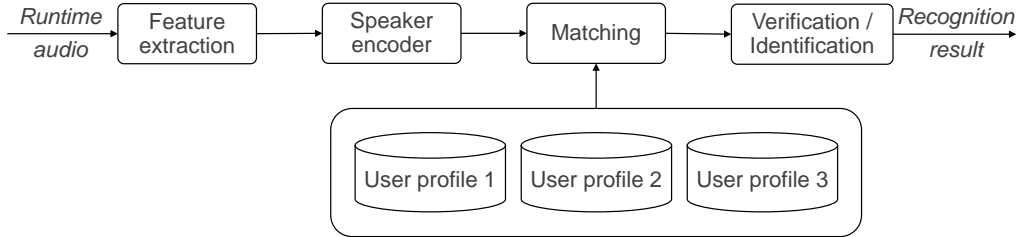


Figure 2: Workflow of the runtime stage of a speaker recognition system.

### 3. Device-side deployment

#### 3.1. Device-side architecture

In device-side deployment, both the speech engine execution and the user profile storage happen on the user device. The user device could be either smartphones, smart home speakers, or smart security devices. The biggest advantage of device-side deployment is that, it does not require any Internet communication with servers. This means both enrollment and runtime stages can perform smoothly even when there is no Internet connection.

One big challenge of device-side deployment is the limited computational resources, such as CPU, memory, storage, and power. In most use cases, the user device (*e.g.* a smartphone) needs to perform many other tasks in parallel, thus the resource budget for speaker recognition is usually very limited. There are many approaches to reduce the computational cost of the speaker recognition system, such as model quantization [13, 14], model compression [15], model sparsification [16], or implementing part of the system on specialized hardware (*e.g.* digital signal processors).

#### 3.2. Single version updating strategy

Version control for device-side deployment is relatively straightforward, as illustrated in Fig. 3. The user device only keeps a single model. After the enrollment stage, the user’s enrollment audio will be stored on the device. When there is a newer version of model available on the model storage server, the user device will download this newer model. Once the download completes, it will immediately trigger a process that uses the newly downloaded model to generate the new version of user profiles based on the enrollment audio. This process guarantees that the version of user profiles stored on the user device always match the version of the model.

### 4. Server-side deployment

#### 4.1. Server-side architecture

In server-side deployment, both speech engine execution and user profile storage happen on backend servers, which is the opposite of device-side deployment. The biggest advantage of server-side deployment is that, the user device only needs to perform very simple operations, such as obtaining the enrollment audio from the user, and communication with the servers. All complicated logic and resource-intensive tasks will be implemented on the servers.

The typical architecture of server-side deployment can be illustrated in Fig. 4:

- During the enrollment stage, the user device first uploads the enrollment audio to the backend database via the frontend reverse proxy server; next, the speech engine on the cloud computing server generates the user profile based on the enrollment audio; and finally, the user profile will be stored in the backend database. Both the enrollment audio and the user profile are stored together with the user’s unique ID.
- During runtime stage, the user device sends the runtime audio together with a set of candidate user IDs to the frontend server; the frontend will fetch the profiles for the candidate users from the backend database, and send them together with the runtime audio to the cloud computing server; finally, the speech engine on the cloud computing server will send the recognition result back to the user device.

The request and response schema for enrollment and runtime stages can be roughly described as below:

```

EnrollmentRequest {
  string user_id;
  vector<Audio> enrollment_audio;
}

EnrollmentResponse {}

```

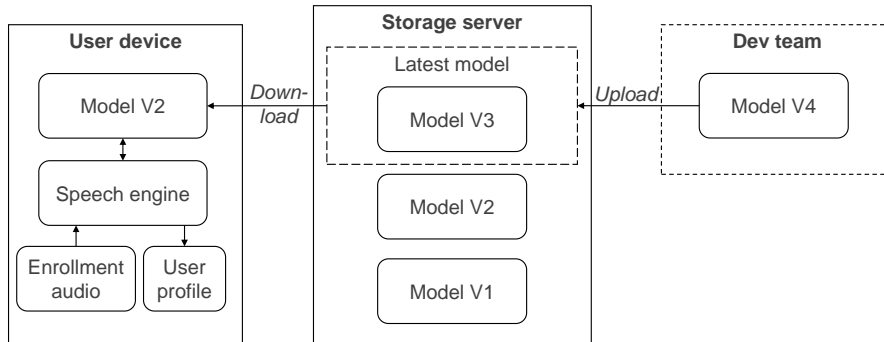


Figure 3: *Version control for device-side deployment. The storage server stores all historical models, and provides a shortcut URL for the user device to download the latest model. When the development team uploads a new model, the URL to the latest model will redirect to this new model.*

```

RuntimeRequest {
  Audio runtime_audio;
  vector<string> user_id;
}

RuntimeResponse {
  map<string, Result> user_id_to_result;
}

```

The problem with the above architecture is obvious: During runtime stage, if the model on the cloud computing server has been updated to a newer version, it will mismatch with the user profiles stored in the database. In the remaining of this section, we will introduce three different version control strategies to handle this problem.

#### 4.2. Single version offline updating strategy

Among all model updating strategies for server-side deployment, single version offline updating is the simplest one. Before we update the speaker recognition models in the cloud computing servers, the frontend server will first stop dispatching any new enrollment or runtime requests to the backend. Instead, the frontend will respond the user device with a special error message, indicating that the backend servers are currently being maintained and updated, and the user device should try again later.

Once the models in the cloud servers have been updated, a background process will be triggered to rerun the enrollment process for all users — the speech engine will process the enrollment audio for each user, generate a new user profile based on the new model, and replace the existing user profile in the database. Once this large-scale re-enrollment process has been completed, all user profiles in the database will have the same version as the models in the cloud computing servers, and the frontend could resume to accept new enrollment and runtime requests again.

Although this single version offline updating strategy is relatively simple and easy to implement, its disadvantages are also obvious:

1. It requires a downtime period of the entire speaker recognition service. If the users are geographically concentrated and the use cases are relatively simple, the updating can be typically scheduled to happen in the local late midnight when we expect very few requests. However, if the users are distributed across multiple time zones,

we may expect requests to the service 24 hours a day, thus the downtime will cause significant frustrations to the user experience.

2. Unlike device-side deployment, where each device only stores the profiles for the owners of the device, in server-side deployment, the database needs to store the profiles of all users. For large-scale applications, the number of users could be huge, thus rerunning enrollment for all users will be a very computationally intensive task. It may not complete within the scheduled downtime.

#### 4.3. Single version online updating strategy

To avoid the downtime issue in the single version offline updating strategy, an alternative solution is the single version online updating strategy. In this strategy, we associate each speaker recognition model with a unique *version identifier* string. During the enrollment stage, when we store the user profile in the database, it is stored together with the version identifier of the model that generated it. Then in the runtime stage, when the frontend server receives a new runtime request, it will first check whether the version identifier of the user profile in the database matches the version identifier of the model in the cloud computing server:

- If the version identifiers match each other, the frontend server will directly trigger the runtime logic as illustrated in Fig. 4b.
- If the version identifiers do not match, the frontend server will trigger another process to rerun the enrollment for the user. After the re-enrollment completes, the versions of the user profile and the model are guaranteed to match each other, and the frontend server will trigger the runtime logic.

As we can see, the single version online updating strategy postpones the re-enrollment process to an on-demand, per-request manner. This guarantees that the speaker recognition service will be available 24 hours a day without downtime.

However, this strategy also has one disadvantage. Once the model in the cloud computing server has been updated, the next runtime request from each user will always experience increased latency due to the re-enrollment. The significance of the latency increase depends on the efficiency of the re-enrollment process. However, since model updating typically happens every few weeks or months, this increased latency is possibly ac-

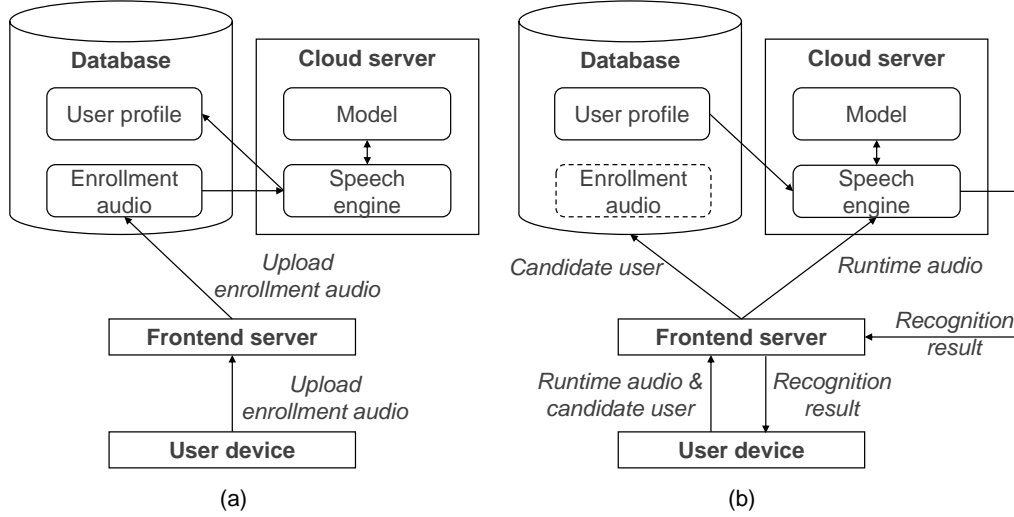


Figure 4: Architecture of server-side deployment of a speaker recognition system. The frontend server is the reverse proxy between the user device and the backend servers; the cloud server performs resource-intensive tasks such as feature extraction and neural network inference; and the database stores each user’s enrollment audio and profile. (a) Enrollment stage workflow. (b) Runtime stage workflow.

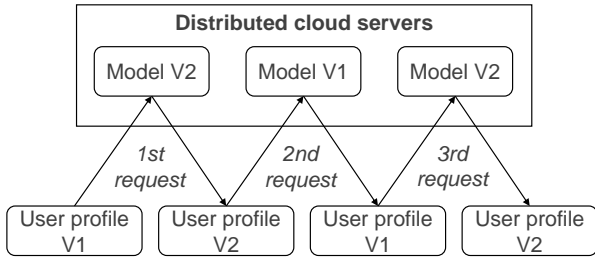


Figure 5: The version bouncing problem for distributed speaker recognition system.

ceptable for most applications — it only happens once for each user after each model update.

Additionally, for large-scale distributed systems, single version online updating strategy has another challenge known as **version bouncing**. In a distributed system, there will be multiple cloud computing servers, each serving a copy of the speech engine. When we update the models for the cloud computing servers to a newer version, the update process typically will not finish synchronously on different machines. This will result in a state that some of the cloud computing servers are serving the new model version, while the other cloud computing servers are still serving the old model version. If a user device sends runtime requests to different servers, the re-enrollment process may happen multiple times, upgrading and degrading the model version in turn, as illustrated in Fig. 5.

There are several methods to avoid the version bouncing problem:

1. The frontend server can periodically send synchronization requests to all cloud computing servers, and maintain a table to record the current model version of each cloud computing server. With this table, if a user profile has been updated, the runtime request will only be dispatched to a cloud computing server with the updated model.

2. The frontend server can implement a load balancing algorithm based on the hash value of the user’s ID, such that requests for each user are always dispatched to the same cloud computing server. This will guarantee that re-enrollment will only update user profile from old version to new version once.
3. Finally, we can store multiple versions of profiles for each user in the database. Once the re-enrollment for a user has completed, we will store both the old version and the new version of this user’s profile. For future runtime requests, no matter which version of model is served in the cloud computing server, no re-enrollment will be needed as both versions of profiles are available.

#### 4.4. Double version updating strategy

As we mentioned before, the single version offline updating strategy requires service downtime for each model update, and the single version online updating strategy will cause increased latency for runtime requests. Here we introduce the double version updating strategy, which will overcome these drawbacks.

In the double version updating strategy, we always serve two versions of models in the cloud computing servers at the same time, and always store two versions of user profiles in the database. During enrollment stage, we always enroll with both models; and during runtime stage, we use the “newest available model”. The coexistence of two versions guarantees that even if we have updated one model to a newer version, the other model is still available, allowing for a grace period for the user profiles to be updated.

There are typically two ways to simultaneously serve two models in the cloud computing servers. First, we could divide the cloud computing servers into two groups, each group serving one model. The group partition is fixed, so the frontend server does not need to periodically synchronize with the cloud computing servers. Alternatively, each cloud computing server could serve two models at the same time with separate processes.

Assuming different versions of models are served in differ-

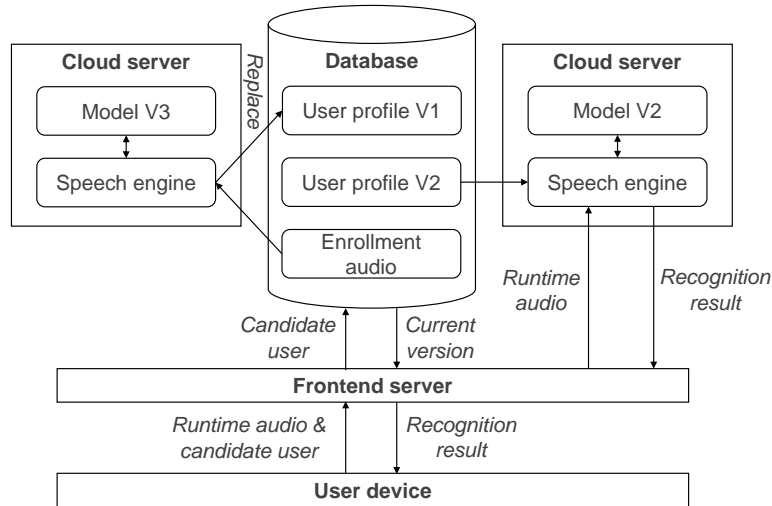


Figure 6: Double version updating strategy for server-side deployment.

ent groups of servers, we use Fig. 6 as an example to illustrate this strategy. Originally, the cloud computing servers are serving model V1 and model V2 simultaneously, and we store both user profile V1 and V2 in the database. When the development team releases a newer model V3, it will replace the group of cloud computing servers that are still serving the oldest model V1. During this process, the frontend is still handling all enrollment and runtime requests:

- Enrollment requests will be dispatched to both cloud computing servers serving model V2 and V3. User profiles for both V2 and V3 will be produced and stored in the database.
- For a runtime request, if the user profiles have not been updated (only V1 and V2), the request will be dispatched to a cloud computing server serving model V2. Because user profile V2 is available, the runtime recognition can be performed smoothly without additional latency (as is the case in Fig. 6). At the same time, the frontend will trigger a re-enrollment process in the background to replace user profile V1 by user profile V3.
- For a runtime request, if the user profiles have already been updated to V2 and V3, the request will be dispatched to a cloud computing server serving model V3 (another case not described in Fig. 6).

As we can see, in the double version updating strategy, while background processes are updating the models on cloud servers to the newer version, and updating user profiles to the newer version, the speaker recognition service will still be always available without additional latency. There will usually be sufficient time to update all user profiles until the next model release. Apparently, this is the most elegant version control solution for server-side deployment. However, the implementation of double version updating strategy is quite complicated, thus may not be the optimal solution for smaller projects with short development cycles.

## 5. Hybrid deployment

### 5.1. Hybrid architecture

In Section 3 and Section 4, we discussed the version control strategies for device-side and server-side deployment. Although device-side deployment is simple and requires no Internet communications, it's not available for many applications where the on-device computational resource budgets are limited. At the same time, storing user profiles on server-side databases may result in privacy concerns [17].

An alternative solution is the hybrid deployment, where the speech engine execution happens on cloud computing servers, but the user profiles are stored on user devices, as illustrated in Fig. 7:

- During enrollment stage, the user device first sends the enrollment audio to the frontend server; then the speech engine produces the user profile from the enrollment audio; finally, the frontend server will send the user profile back to the user device. Once the enrollment stage completes, the servers will immediately delete the user profile from the memory; the user device is responsible for storing the user profiles.
- In the runtime stage, the user device sends the runtime audio together with candidate user profiles to the frontend server; the speech engine will compare the voice identity of the runtime audio against the candidate user profiles; finally, the recognition results will be sent back to the user device. The user profiles are typically encrypted when being stored on the user device and communicated to the servers for security.

Similar to Section 4.1, we provide the rough request and response schema for enrollment and runtime stages of hybrid deployment as below:

```

EnrollmentRequest {
    string user_id;
    vector<Audio> enrollment_audio;
}

EnrollmentResponse {
    bytes profile;
}

```

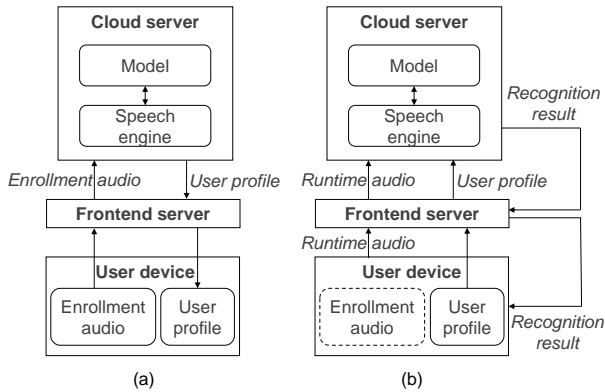


Figure 7: Architecture of hybrid deployment. (a) Enrollment stage workflow. (b) Runtime stage workflow.

```

RuntimeRequest {
  Audio runtime_audio;
  map<string, bytes> user_id_to_profile;
}

RuntimeResponse {
  map<string, Result> user_id_to_result;
}

```

The hybrid deployment is very similar to the server-side deployment, except that user profiles are stored in the user devices instead of in a backend database. Because all device-server communications can only be initiated by the user device, the servers cannot access the user profiles at any given time, which poses a new challenge to the hybrid deployment.

### 5.2. Single version online updating strategy

For hybrid deployment, we could use a single version online updating strategy that is very similar to the strategy we introduced in Section 4.3 for server-side deployment. When the user device sends a runtime request to the frontend server, it will first check whether the user profile version matches the version of the model in the cloud computing server. If the versions do not match, it will trigger the enrollment stage to update the user profile, then perform runtime recognition after the re-enrollment completes.

Similar to server-side deployment, single version online updating strategy will cause increased latency to the first runtime request for each user after the model has been updated. This could be mitigated by implementing a daily *handshaking* communication between the user device and the server, initiated by the user device. This handshaking communication will simply check whether the version matches between the device and the server; if they mismatch, it will silently trigger the re-enrollment in the background. The handshaking communication could happen at the late midnight in the device’s local time zone to minimize user interference.

### 5.3. Double version updating strategy

For hybrid deployment, we could also use a similar double version updating strategy as the one introduced in Section 4.4 for server-side deployment. In this strategy, the cloud computing servers always serve two versions of models, and the user devices also always store two versions of user profiles. During en-

rollment stage, the server always produce two versions of user profiles and send them back to the user device. At runtime, even if one server-side model has been updated to a newer version, the other model is still available for those devices whose user profiles have not been updated.

For hybrid deployment, even if we use the double version updating strategy, we still need to make sure that all user devices complete the update within a certain time frame. Otherwise, if some user devices missed two server-side model updates, both versions of user profiles stored on the device will not be usable. One solution is to implement a periodic handshaking communication between the user device and the server, as we mentioned in Section 5.2.

## 6. Conclusions

In this paper, we introduced the concept of version control in speaker recognition systems. Version control is a common and challenging problem when deploying speaker recognition systems to production environments. Based on how we execute the speech engine and how we store the user profiles, we categorize speaker recognition deployment into three types: device-side deployment, server-side deployment, and hybrid deployment. We introduced version control strategies for each type of deployment, and discussed the advantages and disadvantages of each strategy.

## 7. References

- [1] M. Hébert, “Text-dependent speaker recognition,” in *Springer Handbook of Speech Processing*. Springer, 2008, pp. 743–762.
- [2] G. Chen, C. Parada, and G. Heigold, “Small-footprint keyword spotting using deep neural networks,” in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2014, pp. 4087–4091.
- [3] T. Kinnunen and H. Li, “An overview of text-independent speaker recognition: From features to supervectors,” *Speech Communication*, vol. 52, no. 1, pp. 12–40, 2010.
- [4] H. Hermansky, “Perceptual linear predictive (PLP) analysis of speech,” *the Journal of the Acoustical Society of America*, vol. 87, no. 4, pp. 1738–1752, 1990.
- [5] S. Davis and P. Mermelstein, “Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 4, pp. 357–366, 1980.
- [6] C. Kim and R. M. Stern, “Power-normalized cepstral coefficients (PNCC) for robust speech recognition,” *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, no. 7, pp. 1315–1329, 2016.
- [7] D. A. Reynolds, T. F. Quatieri, and R. B. Dunn, “Speaker verification using adapted Gaussian mixture models,” *Digital Signal Processing*, vol. 10, no. 1-3, pp. 19–41, 2000.
- [8] P. Kenny, “Joint factor analysis of speaker and session variability: Theory and algorithms,” *CRIM, Montreal, (Report) CRIM-06/08-13*, vol. 14, pp. 28–29, 2005.
- [9] N. Dehak, P. J. Kenny, R. Dehak, P. Dumouchel, and P. Ouellet, “Front-end factor analysis for speaker verification,” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 19, no. 4, pp. 788–798, 2010.
- [10] L. Wan, Q. Wang, A. Papir, and I. Lopez Moreno, “Generalized end-to-end loss for speaker verification,” in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 4879–4883.
- [11] C. Li, X. Ma, B. Jiang, X. Li, X. Zhang, X. Liu, Y. Cao, A. Kannan, and Z. Zhu, “Deep speaker: an end-to-end neural speaker embedding system,” *arXiv preprint arXiv:1705.02304*, vol. 650, 2017.

- [12] D. Snyder, D. Garcia-Romero, G. Sell, D. Povey, and S. Khudanpur, “X-vectors: Robust DNN embeddings for speaker recognition,” in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 5329–5333.
- [13] R. Alvarez, R. Prabhavalkar, and A. Bakhtin, “On the efficient representation and execution of deep acoustic models,” in *Proc. Interspeech*, 2016, pp. 2746–2750.
- [14] Y. Shangguan, J. Li, Q. Liang, R. Alvarez, and I. McGrawn, “Optimizing speech recognition for the edge,” *arXiv preprint arXiv:1909.12408*, 2019.
- [15] P. Nakkiran, R. Alvarez, R. Prabhavalkar, and C. Parada, “Compressing deep neural networks using a rank-constrained topology,” 2015.
- [16] Y. LeCun, J. S. Denker, and S. A. Solla, “Optimal brain damage,” in *Advances in Neural Information Processing Systems*, 1990, pp. 598–605.
- [17] S. De Silva, A. Liu, and L. Nabarro, “Europe’s tough new law on biometrics,” *Biometric Technology Today*, vol. 2017, no. 2, pp. 5–7, 2017.

# Appendices

## A. Glossary

- **Speaker embedding:** A vector representing the voice characteristics of a spoken utterance.
- **Speaker encoder:** The algorithm that generates the speaker embedding from the acoustic features of an utterance.
- **Speech engine:** The software that implements acoustic feature extraction and speaker encoder.
- **User profile:** The aggregated speaker embedding generated from multiple enrollment audio samples provided by the user.
- **Model:** The model used by the speaker encoder algorithm. In deep learning based approaches, the model is usually a neural network.
- **Frontend:** In server-side and hybrid deployment, the reverse proxy server that dispatches requests from user devices to backend servers.
- **Cloud server:** In server-side and hybrid deployment, the backend server that runs the speech engine.
- **Database:** In server-side deployment, the backend database that stores enrollment audio and user profiles.