David Matkovic - 7038335
Jona Hoppe - 7034173
Hanna Helbig - 7008726
Maximilian Schmidt - 2356

Introduction

Preprocessing

The initial phase of our project focused on preprocessing the audio data, which is a crucial step that encompasses several subtasks. The preprocessing aims at preparing the raw material for further analysis.

Data Acquisition

The first step before we could start with the preprocessing involved acquiring the songs that we wanted to use for our project. We wanted to ensure a diverse and representative dataset; thus, we searched for publicly available audio collections of Linkin Park. As we did not find a complete collection, we sourced the audios from YouTube. In total, we curated a corpus consisting of eight albums, each containing an average of 12 tracks. This collection formed the foundation for our subsequent processing and analysis steps.

After collecting the data, we started with the preprocessing. Preprocessing is of critical importance for singing voice synthesis, especially when working with raw audio data, which is typically noisy and includes irrelevant or disturbing elements, such as instrumentals, audience noise, and silences, all of which can have a negative impact on model performance [**?**].

As we wanted to train the model to recognize the specific patterns of Chester Bennington's voice, including his pitch, timbre and vocal fry, a good preprocessing is important to ensure that all relevant information is captured by the model and all irrelevant information (noise, other voices, instrumentals) is excluded to provide the best learning. Including instrumentals and noisy data would potentially lead to the model learning these unwanted features, resulting in a bad overall model output. This being said, we tried to preprocess our data, such that we excluded instrumentals and noise and included clean singing voice data, which was later passed through the model.

Segmentation

After the vocal extraction, the next step was to segment the data. Segmenting the audios into smaller chunks is important as it can significantly enhance the model's performance and the reliability of the forced alignment that followed this step [**?**]. The quality of the alignment heavily depends on the segmentation, as a wrong segmentation will lead to a wrong alignment. The effectiveness of our model is, therefore, closely tied to the goodness of the audio segmentation.

The best way to ensure a good segmentation would be to do it manually, however, as our dataset is quite big, this would have been out of the scope of this project, so we searched for automatic segmentation methods.

We tried different methods for automatic segmentation. We first used pydub, which is a Python library and a known powerful tool for preprocessing audio and text data, widely utilized in speech and text recognition applications [**?**]. We also tried pydub.silence, which is a module in the Python pydub library that provides functions to detect and handle silent segments in audio files. After that, we used Whisper, which is a powerful automatic speech recognition system developed by OpenAI

[?]. Whisper is trained on 680,000 hours of diverse, multilingual, and multitask data from the web, which makes it a robust model that can handle accents, background noise, and technical or domain-specific language, making it a suitable choice for our project, as speech clarity and variability are some of our main challenges. Whisper supports transcriptions for multiple languages. During the description, Whisper detects voice activity and attenuates background noise or music. In order for Whisper to work, PyTorch and ffmpeg must also be installed (those are its dependencies) [?]. Whisper is not intended for real-time transcription and is designed to process audio containing at least one complete sentence, ideally under 30 seconds in length.

Whisper's main challenge lies in balancing accuracy and resource efficiency, as it can produce hallucinated transcriptions and larger models demand high computational power, which limits their usability on resource-constrained devices [?].

We also used whisper-timestamped, which is an extension of whisper and provides more efficient and accurate word timestamps. Whisper-timestamped uses a Dynamic Time Warping approach, which allows it to create word timestamps and a more accurate estimation on speech segments. As a result, the estimated start and end times of speech segments are more precise. Another advantage over the base model is that Whisper Timestamped can process longer files [?].

Whisper-timestamped turned out to be the best model to segment our audios. It works with different model sizes: base, small, medium, and large. We used the small model as it was sufficiently well.

Following an initial phase of trial and error, we implemented a custom script utilizing whisper-timestamped to segment our audio data. The script first verifies the file format, ensuring that only audio files (with the ending wav or mp3) are processed and that only those files whose names conformed to our naming convention (i.e., filenames containing either six or nine digits) were included. Once the appropriate files were selected, we used whisper-timestamped to generate segment-level transcriptions along with time-aligned boundaries. Most of the segments were fine, but some segments were extremely short (i.e., less than two seconds). To avoid these short segments and to enhance the usability of the resulting segments for annotation purposes, we introduced a minimum segment duration of four seconds. Thus, segments falling below this threshold were automatically concatenated with the subsequent segment to ensure that each snippet was sufficiently long to carry meaningful context for the automatic labeling. We also had some issues with hard cuts, where word endings were cut off too strictly. To mitigate this problem, we introduced a small buffer between consecutive segments (0.1s). This slight overlap helped preserve the natural flow of speech and ensured smoother transitions, which improved both the accuracy of our transcriptions and the overall quality of the audio snippets.

After segmenting our data, we had to align our audio data with textual/phonetic annotations. Here, we encountered another challenge as some segments were silent, resulting in an error, thus, we excluded segments with zero variance before aligning the data.

Alignment Tools

We initially used the Montreal Forced Aligner (MFA), which is a widely recognized tool for automatic speech-text alignment. The MFA is built upon the Kaldi speech recognition toolkit. It is designed to provide robust phonetic segmentation by leveraging triphone acoustic models and

speaker adaptation techniques [**?**]. MFA has been successfully applied to various languages and datasets; however, in our case, the MFA did not reveal convincing results. We attributed this to the fact that the MFA is primarily designed for spoken language, while singing poses some additional challenges, such as extended phonemes and pitch variations.

To improve alignment quality, we explored alternative tools and came across SOFA (Singing-Oriented Forced Aligner), which functions similarly to the Montreal Forced Aligner (MFA) but is specifically designed for singing voice. According to its Github page, SOFA offers simpler installation, better performance, and faster inference speed compared to MFA. After testing it on a few examples, we observed slightly improved results [**?**]; however, the alignments were still not consistently convincing. We continued our search and discovered LabelMakr, "a GUI tool to help users easily generate SVS phoneme-level labels" [**?**] for singing voice synthesis (SVS), particularly intended for use with DiffSinger. While initial tests with a small sample also fell short of expectations, we observed that LabelMakr's performance significantly improved when it was applied to our full dataset, such that we ended up with some meaningful alignments that we used for our training.

Data Conversion with nnsvs-db-converter

We used the nnsvs-db-converter, a Python tool for converting vocal databases with HTS mono labels into DiffSinger format to make them compatible with variance models for predicting pitch and timing [**?**]. It handles sample segmentation by ensuring that each sample has a maximum length and an acceptable number of pauses between segments, and converts silent phonemes (e.g., sil, pau) to standardized representations (SP and AP for breaths). The script requires minimal external dependencies, making it easily accessible. In addition, it supports language definitions that specify vowels and liquids for better segmentation and timing handling, and it can estimate MIDI pitch and timing information for the use of variance models.

In our project, we used this converter to convert our .lab files, which contained the phoneme alignments, into .ds files to ensure they were compatible with the DiffSinger framework and could utilize variance-based pitch and timing predictions.

SlurCutter Exploration

We also had a look into SlurCutter, which is a specialized tool that is designed to assist in the creation of datasets for DiffSinger. Its primary function is to edit MIDI sequences that are stored in .ds files to facilitate the preparation of data for variance model training [**?**]. However, it requires a lot of time and effort as this process is done manually, so we decided not to do it on a large scale.

Manual Segmentation

To include more data and improve model performance, we additionally included two live albums in our dataset. As many segments contained screaming from the audience, we went through them and cut out unwanted parts manually with Praat, a tool for speech analysis and phonetic research.

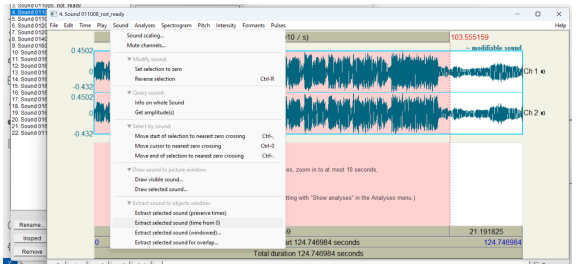

Figure 1: excluding noisy screams at the end of the audio snippet

Manual Alignment

In addition to our segmented audio snippets, we fully labeled two complete songs — Over Each Other and The Emptiness Machine — using LabelMakr, which we later used during inference. Since these full-song annotations were created after the short segments had already been labeled, the initial alignments were expectedly poor. To address this, we manually refined the boundaries: we shifted misaligned ones, added missing boundaries, and removed others — particularly SP (speech pause) and AP (aspiration pause) annotations, which frequently appeared in inappropriate positions.

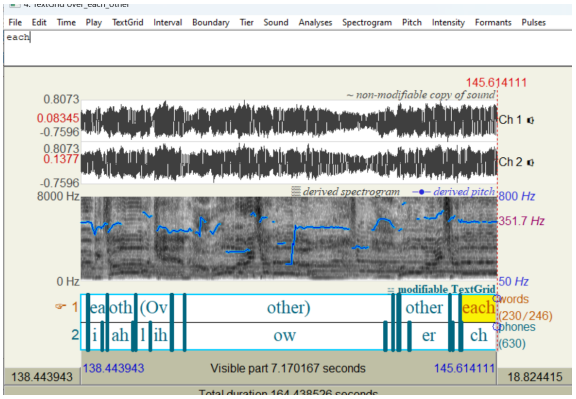

Figure 2: bad alignment

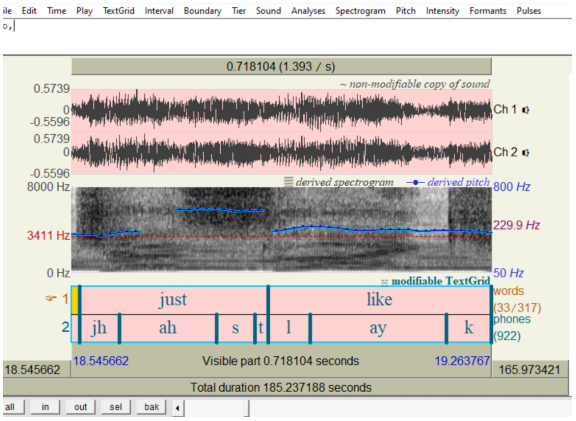

Figure 3: manual alignment

Refined Preprocessing

As our initial training revealed noisy outputs, we decided to improve our dataset to enhance model performance. We began by manually manipulating two albums, where we removed segments that contained excessive noise, overlapping voices, or rap sections. The result was a cleaned subset of data, which we stored as our clear training data.

To scale this cleaning process across the entire dataset, we wrote several scripts to automate the filtering process. First, we analyzed how different types of data snippets varied across the corpus. We manually selected representative examples of each category—screaming, multi-voice, clear voice, and rap—and stored them in separate folders to serve as our reference data.

Initially, we attempted to map each song snippet to the category it was most similar to. However, this yielded inconsistent and often unreliable classifications, such that the categories had a large overlap and seemed to be meaningless. We then tried to identify dominant acoustic features that could meaningfully distinguish between categories. We found that fundamental frequency was a strong discriminator between screaming and clear singing. After analyzing our reference examples, we set a threshold of 230 Hz to filter out heavy screams and background noise—this removed many problematic segments while retaining the majority of clean vocals. The remaining screams in the clean voice examples were retained as they were less problematic than the really heavy ones and the background noises.

To distinguish between rap and singing, we found that syllable rate is a suitable discriminator. Since our target singer, Chester Bennington, does not perform the rap sections, we applied a threshold of six syllables per second to exclude the fast-paced vocal segments.

For segments with multiple overlapping voices, we explored various characteristics and thresholds, but none of them seemed to be a reliable discriminator. Therefore, we returned to the similarity-based approach—this time using only a single, well-chosen example per category to guide the classification, which surprisingly seemed to work better than with the bigger example set. While the categories were still imperfect, this method provided the most usable approximation.

Ultimately, we combined the resulting clear voice set (without screams, rap, multiple voices) with other data to train three of our models (see model training).

Training

Manual Augmentation

Data augmentation plays a crucial role in improving the fine-tuning and accuracy of speech attributes like pitch, loudness, and duration, enhancing the quality of AI-generated singing [?]. To enhance the diversity and robustness of our audio dataset, we implemented several data augmentation techniques: pitch shifting, loudness adjustment, noise reduction, and speed alteration. These methods are widely recognized for their effectiveness in improving speech recognition models by simulating speaker variations [?]. As singing is especially prone to these elements (e.g., pitch, loudness, and speed change drastically during a song), these augmentation techniques appeared to be especially useful for our purposes.

- **Pitch Shifting**: We wrote a script to adjust the pitch of each audio file by a semitone both upwards and downwards, creating two additional variations per original file. This technique helps the model become invariant to pitch differences as the pitch changes heavily between Chester's different singing modes (e.g., compare harmonic singing versus screaming). We also tried to make more drastic pitch shifting, however, then the voice seemed to be too different from the original, so we did not include it in our model training.
- **Loudness Adjustment**: The audio files' loudness was modified by increasing and decreasing the volume by 10%. This approach aids the model in handling variations in speaking volume and recording conditions.
- **Noise Reduction**: We applied a noise reduction technique to suppress background noises such as ambient sounds and audience noise. This process aimed to enhance the clarity of the singing voice and thus, enhance model performance. However, we found that the denoising also suppressed the screaming parts quite a bit, so we decreased the initially high penalty to only 30% noise reduction and used a less strict high-pass filter. For some parts, the screaming was still denoised too heavily, however, we used the denoised parts too to make the model more robust.
- **Speed Alteration**: The speed of the audio recordings was adjusted by $\pm 10\%$, creating versions that are slightly faster and slower than the original. This method exposes the model to variations in singing rates, improving its ability to generalize across different singing tempos.

By incorporating these augmentation strategies, we aimed to increase the robustness of the model, leading to better overall performance.

Evaluation

Mixing and Mastering

Conclusion

Conclusion The goal of our project was to develop an SVS model that could replicate the unique voice of Chester Bennington. Our approach focused not just on the training itself, but on creating a comprehensive pipeline that would allow others to have a guide that covers everything from data processing, data augmentation, training, evaluation and post-processing of the data to be able to use DiffSinger with their own data. So our intention is not just to create a replica of Chester's voice, but to provide an understandable guide to creating a DiffSinger model.

After creating our dataset with the unprocessed audio, we began pre-processing, an important step that determines the quality of our model. The first step was to extract the vocals to ensure that the model was only trained on relevant vocal data. This was followed by the segmentation and alignment of the data, as well as the creation of ds-files, which are needed for the use of DiffSinger.

This first preprocessing was followed by a second, which focussed on improving the quality of the data. For this purpose, some songs were edited manually so that disturbing noises and inappropriate

sequences (e.g. wrong singer) were cut out. Some scripts also filtered out faulty segments. The alignment and ds-file creation steps were then repeated.

For the data augmentation process, we explored both DiffSinger's built-in augmentation functions and manual augmentation methods. While the built-in methods included pitch shifting and time stretching, we also applied pitch shifting, volume adjustment, noise reduction and velocity change to simulate real-world variations in vocal performance.

Setting up the training presented us with some challenges in terms of configuring the environment, setting up the hardware and dealing with outdated documentation. We initially tried to use Docker, but ran into issues with the module locations, so we decided to switch to a more reliable Anaconda environment. Once we had configured our system, we trained our models on the LST servers at Saarland University using HTCondor to schedule training jobs. The training progress was monitored with TensorBoard.

We trained different models experimenting with different parameters such as the size of the dataset, different quality of the dataset, the use of data augmentation, different vocoders and different learning rate and fine-tuning settings.

Despite the many different experiments, the results of the models are relatively comparable. While the data augmented models seem to outperform the small and large model, and the fine-tuned vocoder also improved model performance, the final results are still disappointing in quality. Some segments are processed well by (almost) all models,however, the output of other segments is inadequate, as the voice sounds distorted and rough or other background noises appear to be involved. In many segments, the phonemes or phoneme boundaries also appear to be incorrect. This is probably due to the alignment. To address this issue, aligning the data manually would almost certainly lead to a better model as many boundaries are inadequate. However, this step requires a non-negligible amount of time, offering potential for future research. There can also be made improvements within the data augmentation as the project has highlighted the importance of careful and balanced augmentation to avoid degrading the model's performance.

The main contribution of this project is the creation of a comprehensive guide for setting up and training a DiffSinger model, from data pre-processing to fine-tuning the model and post-processing. By documenting each stage of the process, we provide a valuable resource for others who wish to emulate or improve upon our work in speech synthesis. This guide serves as a step-by-step reference for anyone interested in training a speech synthesis model with DiffSinger, covering not only technical details, but also the challenges we encountered and how they were overcome.

To summarise, this project has provided valuable insights into the use of DiffSinger for voice replication, particularly in terms of data pre-processing, augmentation, model setup and post-processing. Our project also highlights the shortcomings of the results, which are mainly due to poor alignment and insufficiently high-quality data (e.g. an insufficient number of acapellas). Therefore, further refinements are needed to fully capture the nuances of a unique voice like Chester Bennington's. Nonetheless, our efforts have established a solid foundation for future research and development in the field of voice synthesis, with the guidelines presented here providing a robust framework upon which others can build.

References