SUR

Filip Banák
xbanak01@stud.fit.vutbr.cz

May 4, 2025

1 Audio data

1.1 Feature extraction

The features being classified are MFCCs. But before MFCCs are extracted, some preprocessing happens.

A simple voice detection algorithm is applied to extract voiced frames only. First, FFmpeg's *silenceremove* effect is applied on the ends of the signal, to remove digital zeros and frames that are undeniably just silent.

Then, audio is framed into non-overlapping windows, and either the logarithm of energy or the 0th MFCC is extracted from each window. The resulting sequence is smoothed using a median filter and normalized into interval $[0, 1]$. By default the 0th MFCC is used.

The first second of the recording is cut if it's not voiced as loudly as the rest on average.

Sound is then pre-emphasized, and an empirically chosen threshold of 0.5 is used to pick frames with (not pre-emphasized) energy greater than this threshold as voiced, the rest is discarded.

You can visualize this process and adjust some of its parameters interactively in the `data` notebook.

After the voiced recording is obtained, MFCCs are extracted, and for each recording, the mean of its coefficients is subtracted, to discard systematic effects of the recording devices.

First 23 shifted MFCCs are then used as features.

1.2 K-NearestNeighbors Model

The number of feature vectors of each class is first balanced by reduction. The number of vectors of classes that have more than the class with the least amount of vectors are reduced to have the same amount, vectors to keep are chosen randomly, without replacement.

An index for euclidean distances is then built using this reduced set. To obtain class probabilities of a single recording, the following is done.

The recording goes through exactly the same feature extraction process as the reference data. K is set to 3. For each frame, KNN are found. Then scores for every label are initialized to zero. For every frame the 1st NN gets K points, the 2nd gets K-1 points, and so on until the Kth gets just a single point. The total score of each label is then divided by the total sum of scores of all labels. This produces a quantity that is interpreted as a probability for the purpose of classification. The probability of a single class is thus determined by the number of occurrences in KNN and their distances.

More ranking strategies are implemented, but the one described above seems to perform best.

Trained using the original training and validation split, this classifier achieves accuracy of about 80 %.

1.3 Gaussian Mixture Model

For classification using GMM, the training data is balanced and features are extracted in the same way as for KNN classification. The priors for each class are uniform.

To obtain class probabilities of a single recording, the following is done. The recording goes through exactly the same feature extraction process as the reference data. For each frame, the GMM likelihood of each class is evaluated. Then the simplifying but false assumption of frame-independence is applied and the likelihoods are multiplied across frames and with the priors, producing a posterior probability for each class.

The best configuration found empirically makes up each GMM from two components using full covariance matrices. I tried modeling using diagonal covariance matrices, after applying LDA with no reduction of dimensionality, and while the diagonal strategy worked better with than without LDA, it did not work as well as the full strategy.

Trained using the original training and validation split, using 100 EM iterations, this classifier achieves accuracy of $>= 90$ %.

1.4 Hyperparameters

I tried playing with different combinations of described preprocessing steps, enabling and disabling voice activation detection, preemphasis, balancing, choosing between energy and 0th MFCC, different numbers of nearest neighbors. The combination described in individual classifiers seemed to work best more consistently than the others, but none of this led to a significant improvement. Maybe a more thorough search using crossvalidation could help, but was not executed.

2 Image data

2.1 Convolutional NN Model

For image classification, a convolution neural network model was employed. The basic architecture is the one from ResNet models with bottleneck blocks. It is most similar to the ResNet50 model, but the number of channels in each layer was reduced.

Unfortunately, the results are poor. Given the very limited amount of training data, I could not get the validation set classification accuracy above about 20 %. I tried many combinations of different data augmentations, adding dropout layers, changing the number of channels and blocks, using learning rate scheduling.

I was able to achieve perfect accuracy on training data, thus overfitting. I was able to overcome overfitting, using more augmentations and limiting model capacity, but that did not improve validation accuracy.

I am not sure what more could have been done, besides completely changing the architecture, to mitigate this. I failed to force the model to generalize.

AdamW was chosen as the optimizer, with no weight decay. Learning rate scheduling was also applied, decreasing it if the training loss plateaued. An optimizer step was taken after each epoch. Specific numbers are in the notebook.

Trained using the original training and validation split, this classifier achieves accuracy of about 20 %. I am not very proud of this one.

3 Reproduction

The project is implemented in Python, but with some additional dependencies. Everything interesting is in IPython notebooks (the latest cell outputs are included).

- UV Python package manager
- FFmpeg (read details below)

Tested on Linux only, with Python 3.13 (UV can download a build automatically).

3.1 UV

Please install UV using the instructions linked above. Then change directory to `SRC` and run the following commands.

- `uv sync --all-groups`
- `uv run ipython kernel install --user --env VIRTUAL_ENV $(pwd)/.venv --name=project`

This will install all needed dependencies (inlcuding JupyterLab, registering `project` kernel) except FFmpeg.

3.2 FFmpeg

FFmpeg is needed as a backend for PyTorch Audio and to apply some effects. But PyTorch and the desired effects support only some specific versions. The easiest way, that will be presented here, is to download a prebuilt FFmpeg distribution with shared libraries.

The webpage linked above contains a link to "64-bit static and shared builds", under releases download the shared build of version 6.1 (like this one).

Download and extract it somewhere, the distribution should contain a `lib` directory. The path to this directory needs to be stored in the `LD_LIBRARY_PATH` environment variable when running JupyterLab, so that PyTorch knows where to look for our FFmpeg shared libraries.

3.3 Run

The following command will open JupyterLab with all needed dependencies discoverable. Make sure to replace the path to the `lib` folder of FFmpeg with your downloaded one.

```
LD_LIBRARY_PATH=ffmpeg-n6.1/lib uv run jupyter lab
```

To reproduce results, use notebooks `voice-knn`, `voice-gmm` and `image`. Make sure to select the `project` kernel. Also, make sure the data is present in the directories hard-coded in the notebooks.

The voice detection step is interactively visualized in the `data` notebook.