SUR projekt

Autori: Jakub Kasem, Adam Dzurilla

Spustenie

Riešenie projektu sa skladá z troch častí.

- Detektor tváre (`src/img/`)
- Detektor hlasu (`src/speech/`)
- Kombinovaný detektor (`src/combined.py`)

Každú z týchto častí je možné samostatne spúšťať.

Pred spustením riešenia projektu je potrebná inštalácia závislostí, ktoré sú vypísané v súbore `src/requirements.txt` . Odporúčame vytvoriť *virtual environment*.

Pred samotným spustením riešenia je potrebné v skripte `src/combined.py` upraviť objekt `DATA_DIR` na cestu k evaluačným dátam. Taktiež je potrebné zo zložky `src/img` spustiť skript `augmentData.py` , bez ktorého by nebola možná klasifikácia tvárí.

Vytvorenie virtual environment a inštalácia závislostí:

V `src` zložke:

1. Vytvorenie virtual env

```
python3 -m venv .venv
```

2. Aktivácia virtual env

Unix:

```
source .venv/bin/activate
```

Windows:

```
.venv\Scripts\activate
```

3. Inštalácia závislostí

```
python3 -m pip install -r requirements.txt
```

4. Vyhodnotenie súborov

```
python3 combined.py
```

Výsledné súbory sú generované do repozitára `src/predictions` .

Detektor tváre

Systém, ktorý klasifikuje tváre je implementovaný v jazyku Python ako konvolučná neuronová sieť pomocou rozhrania Keras v knižnici Tensorflow.

Popis systému

Architektúra neuronovej siete:

- Konvolučná vrstva - 32 filtrov, 3x3 kernel
- Relu aktivácia, 2x2 maxpooling
- Konvolučná vrstva - 32 filtrov, 3x3 kernel
- Relu aktivácia, 2x2 maxpooling
- Konvolučná vrstva - 64 filtrov, 3x3 kernel
- Relu aktivácia, 2x2 maxpooling
- Plne prepojená vrstva - 64 neurónov
- Relu aktivácia, 0.5 dropout
- Výstup siete, sigmoid aktivácia

Techniky a vyhodnocovanie systému

Počas vývoja som experimentoval s pridávaním/odstraňovaním vrstiev a ich modifikáciou. Taktiež som experimentoval s rôznymi *optimizers*, počtom epoch a počtom *batches*. Prikláňal som sa k riešeniam, ktoré na vybranom počte epoch najlepšie konvergovali a nemali moc (relatívne) veľký rozdiel medzi trénovacou presnosťou a validačnou presnosťou, aby sa predišlo *overfitting*.

Pred samotným experimentovaním som obohatil zadané dáta o augmentácie a to pomocou skriptu `src/img/augmentData.py` . Augmentácia dát sa veľmi pozitívne podpísala na úspešnosti modelov.

SPEECH

This speech documentation describes the approach that we used for deciding if the .wav file was recorded by the target person or not. It outlines methods that we use for training and how we evaluate the test data or any unseen data.

How to run the program

1. Go to the directory `speech`.
2. Install dependencies using command below or install dependencies from parent directory:

```
python3 -m pip install -r requirements.txt
python3 -m pip install -r ../requirements.txt
```

3. You have two options:
    1. You can run training
    2. You can run an evaluation of the test set

Running training:

```
python3 train_speech_model.py
```

Running evaluation:

```
python3 evaluate.py
```

If you want to run an evaluation, first you need to change `speech.utilities` import and `parent_dir` variable as described in the comments.

Train speech model code

First, what you need to do when you want to train the model, you need to set up the correct path to the config file, the path is in the `train_speech_model.py` file in the main function.

In the config file, you can change training parameters.

1. Loading files and data augmentation We used the `wav16khz2mfcc` function from the provided codes and added data augmentation for different stretch speeds, we also augmented the file with random noise. With these two techniques, we created six new augmented data (five with stretching speed and one with adding noise). The function is in the `utilities.py` file. Data augmentation is also described below. Also, we implemented trimming the first 1.5 seconds and trimming silence from both train and dev speeches using the `librosa` library.
2. Decide the number of Gaussian mixture components used for the target and nontarget models. We tried to change the number of Gaussian mixture components used for the target and non-target models, firstly we started with 5 components for both target and non-target models. Without data augmentation, the model worked fine but after augmentation was added, the correctness of the model rapidly decreased. We then tried to set the numbers to 25 which led to prolonged training but improved the model correctness back to the 90%+ as was before with 5 components and without data augmentation. We then decreased the number of components to 15 to avoid slow training and measured model correctness which was again above 90%.
3. Initialize all variance vectors. We kept this code from the provided codes.
4. Generating weights Here we tried two different approaches as initial guesses for the weights. Firstly we used uniform distribution and then we tried random weights distribution. The distinct distributions were examined on non-augmented data.

    ```
    # Use uniform distribution as initial guess for the weights
    Ws_target = np.ones(M_target) / M_target
    Ws_non_target = np.ones(M_non_target) / M_non_target

    # Use random distribution as initial guess for the weights
    Ws_target = np.rand(M_target)
    Ws_non_target = np.rand(M_non_target)
    ```

    Uniform weights distribution:

Score border	Correctness of target	Correctness of non-target	Correctness average
-200	**100.00%**	87.00%	93.50%
0	99.00%	88.83%	93.92%
200	**100.00%**	90.83%	95.42%
400	97.00%	94.17%	**95.58%**
600	93.00%	**97.00%**	95.00%

    Random weights distribution:

Score border	Correctness of target	Correctness of non-target	Correctness average
-200	**100.00%**	86.67%	93.33%
0	**100.00%**	89.33%	94.67%
200	99.00%	91.50%	95.25%
400	95.00%	94.33%	94.67%
600	96.00%	**95.33%**	**95.67%**

If the score is above `Score border` value, the model predicts that the `tst` file is target, otherwise it is not target. All the score border tests were calculated as average from 10 different runs. Weight distribution is not significant for model correctness.

5. We are running n iterations (n is specified in the config file as `iterations`) to get distinct results. We set the number of iterations for each training before starting. The program runs n iterations and calculates the correctness of the target model, the correctness of the non-target model, and also the average correctness which is calculated as `(correctness_target + correctness_non_target) / 2` and this average correctness gives us a better understanding of how good the parameters are. For each iteration:
    1. Initialize mean vectors to randomly selected data points from corresponding class. This code is from the provided codes.
    2. We measure the time for training the two GMMs for target and non-target models. In this step, we tried different numbers of iterations, with a maximum of 100. This training was slow and took 650 seconds on Windows which is almost 11 minutes. From the log information about total log-likelihoods, we discovered that the convergence is mostly around the 30th iteration which is the same as in the provided codes. 30 iterations took approximatelly 198 seconds on Windows. We also tried to eliminate EM algorithm training, but the correctness decreased. Increasing the number of iterations in EM algorithm training led to slower training but slightly better results. After convergence, the results also converged.

After 90 iterations of the EM algorithm to train the two GMMs from target and non-target, we have had:

```
Iteration: 90  Total log-likelihoods: -2314243.0034861485 for target; -17922917.091317676 for non targets.
```

The `TTL_target` value (-2314243.0034861485) is significantly higher (less negative) than the `TTL_non_target` value (-17922917.091317676), indicating that the model is much better at explaining the data from the target class compared to the non-target class.

We see that there could be a reason to set the score border to a different number than 0. The model is better at recognizing when speech is the target, when it is not, it is a little prone to guessing that the input is the target. This information encourages us to push the boundary a little higher, but there is a risk of overtraining and worse generalization. Paradoxically, our best fit was with a threshold below 0.

3. After training parameters, we evaluate test data for both target and non-target data. The results we save to `score_CLASS` lists with scores for each data. Then we try to apply different borders to discover which can split the data most accurately. We log information about the target model correctness, the non-target model correctness, and the average correctness.
4. We also remember the highest `max_avg_correctness` and if some border and parameters outperform the current maximum, we save these parameters into `.txt` files for evaluation. We also save that `border` and `max_avg_correctness` data into files, border for evaluation and maximum correctness for the next training.

Below we describe three main approaches that we tried to use to decide if the tested data is target or not. The best was the `Score` approach closely followed by the `Average probability` approach. The `Max probability` approach was not working. In the Score section we also summarize the results of the best run.

Score

```
ll_target = logpdf_gmm(tst, Ws_target, MUs_target, COVs_target)
ll_non_target = logpdf_gmm(tst, Ws_non_target, MUs_non_target, COVs_non_target)
score_res = sum(ll_target) - sum(ll_non_target)
score.append(score_res)
```

Correctness is described in the weights distribution tables above (The scoring approach was used there). Our best found is described in the table below.

Score border	EM algorithm training iterations	Gausian mixture components	Average correctness
-150	30	15 for both	99.17%

Average probability

```
ll_target = logpdf_gmm(tst, Ws_target, MUs_target, COVs_target)
ll_non_target = logpdf_gmm(tst, Ws_non_target, MUs_non_target, COVs_non_target)

# Compute log-odds ratio
log_odds_ratio = ll_target - ll_non_target

# Apply sigmoid function to get probability score
probability_score = sigmoid(log_odds_ratio)

# Aggregate probabilities across frames
final_probability_score = np.mean(probability_score)    # Average probability
```

If final_probability_score is >= 0.5, it is target file.

Correctness of target: 100.00%
Correctness of non-target: 88.33%

The average probability is worse than `score prediction`.

Max probability

```
ll_target = logpdf_gmm(tst, Ws_target, MUs_target, COVs_target)
ll_non_target = logpdf_gmm(tst, Ws_non_target, MUs_non_target, COVs_non_target)

# Compute log-odds ratio
log_odds_ratio = ll_target - ll_non_target

# Apply sigmoid function to get probability score
probability_score = sigmoid(log_odds_ratio)

# Aggregate probabilities across frames
final_probability_score = np.max(probability_score)    # Max probability
```

If final_probability_score is >= 0.5, it is target file.

Correctness of target: 100.00%
Correctness of non-target: 0.00%

This is because in every .wav file there is probability somewhere around 0.99 that the frame is from target file. So this approach is useless for our problem.

Data augmentation

We implemented data augmentation for training data, you can find the code for data augmentation in `speech/utilities.py` in functions `augment_audio`, `augment_add_random_noise` and `wav16khz2mfcc` where the augmentation is called.

In the beginning, the results were bad, the correctness of non-target data dropped to somewhere around 50-60%. We solved this by increasing the `M_target` and `M_non_target` Gaussian mixture components used for the target and non-target models. It had a negative impact on the training time of the model, but the average correctness percentage between the target and non-target data went back to 90%+.

We added new augmented data with different stretch speeds. As we can see, we used 5 different stretch speeds for augmentation and we also created an augmented audio file with noise. So instead of having 20 target data and 132 non-target data, we have 140 target data and 924 non-target data.

In the end the data augmentation ended increasing the model correctness by around 5%.

Evaluate data

Speech data evaluation can be called only from the parent directory (running only speech evaluation is possible, but you need to change two things in the `src/speech/evaluate.py` file, the two things you need to change are described in the comments), the `evaluate_speech_data` function expects one argument: directory path with the data for evaluation. There is also one optional argument to return probabilities instead of scores. The score is more precise because the best decision border was found for the data. However, the disadvantage of the decision boundary is perhaps a worse generalization.

In a combined solution we use the probability option for speech evaluation. Also for only speech data we used rather only percentage to avoid problems with generalization when the border is moved away from zero.