Gender and Age Classification from Sound

Orya Spiegel and Roi Birger

orya.sp@gmail.com

rbirger123@gmail.com

This article was written as a final project for the 'Deep Learning methods for Language analysis and Sound' course in B.Sc. Computer Science and Mathematics in Ariel University.
The course is run by Dr. Or Anidjar.

Link to our code: `https://github.com/Orya-s/DeepLearningSound`

Abstract

Automatic sound analysis is becoming more relevant in the modern world. Extracting information from the speaker can serve many purposes for varied kinds of companies. For example, marketing companies can fit the right sort of products for a customer, after recognizing the age and the gender of the speaker. In addition, digital assistants like Apple's Siri are programmed to respond only to the owner of the device. This kind of technologies can improve user experience significantly and save a lot of money and resources for companies.

Keywords

Deep Learning, CNN, Classification from Sound, Sound Classification, Machine Learning, age, gender.

Introduction

Age and gender play important roles in social interactions. Languages reserve different grammar rules for men or women, and very often different vocabularies are used when addressing adults compared to young people. Despite the basic roles these attributes play in our day_to_day lives, the ability to automatically estimate them accurately and reliably from sound, using artificial intelligence models, is still lacking.

In our project, we used a Deep Learning model to classify age and gender from sound. Deep learning is a type of machine learning and AI that imitates the way humans gain certain types of knowledge. To use sound files for Deep Learning we used the Wav2Vec model which converts Wav files to torch vectors. After achieving the vectors, we used a Neural Network to make the predictions. A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. The network we used is Convolutional Neural Network (CNN), which we will elaborate on further when explaining the model structure.

Datasets

We started by using the Common Voice dataset (Common Voice (mozilla.org)). Common Voice is a publicly available voice dataset, powered by the voices of volunteer contributors around the world. People who want to build voice applications can use the dataset to train machine learning models.

Common Voice contains the next features: client id, path, audio, sentence, age, gender, accent, and language. We used the audio as our feature and age and gender as our labels. After preprocessing the data, we got 31,134 different attributes.

While training our models we came to the conclusion that adding more data can significantly improve our results, so we added a big part of the vox_celeb dataset.

VoxCeleb is an audio-visual dataset consisting of short clips of human speech, extracted from interview videos uploaded to YouTube. There are two VoxCeleb datasets, we used VoxCeleb1. VoxCeleb contains a file called Metadata, which contains the features for each speaker – ID, name, gender, nationality and whether the speaker is in the train (dev) set or the test set.

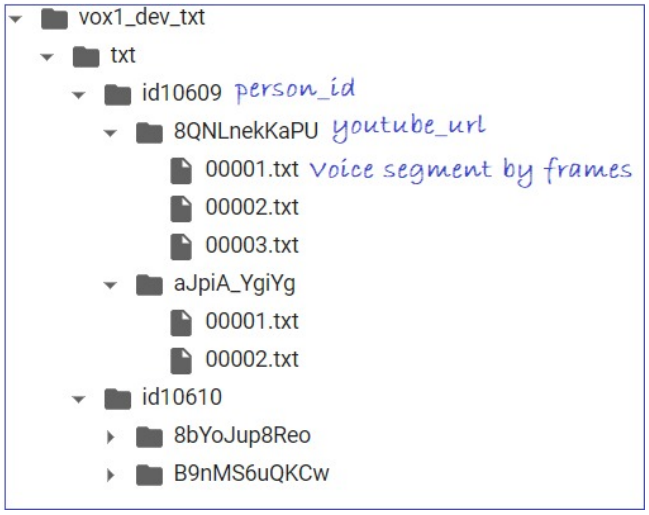
Structure of metadata-

	A	B	C	D	E
1	VoxCeleb1 ID	VGGFace1 ID	Gender	Nationality	Set
2	id10001	A.J._Buckley	m	Ireland	dev
3	id10002	A.R._Rahman	m	India	dev
4	id10003	Aamir_Khan	m	India	dev
5	id10004	Aaron_Tveit	m	USA	dev
6	id10005	Aaron_Yoo	m	USA	dev
7	id10006	Abbie_Cornish	f	Australia	dev
8	id10007	Abigail_Breslin	f	USA	dev
9	id10008	Abigail_Spencer	f	USA	dev
10	id10009	Adam_Beach	m	Canada	dev
11	id10010	Adam_Brody	m	USA	dev

The train and test sets are two separate files arranged in the same way. We will explain the structure of the train set as an example. The train set is a folder called vox1 dev txt, which contains a single folder called txt. In txt there are multiple folders, each one represents a different speaker by a unique ID, which connects the content of the folder with features in the metadata file. In each of the ID folders there are a few other folders, each one represents a different YouTube video, with the URL as the name of the folder. In each of these folders there are a couple of txt files, each one represents a segment of the video by frames in which the speaker is talking. Each second in the video is represented by 25 frames.

To get the audio segment, we took the first frame and the last frame in each txt file and used it to cut the right part of the video. When a speaker had a few separate parts in the video where he was speaking there were a few txt files in the folder. Taking the consecutive frames which represent a part of the video allowed us to get the audio we wanted from each speaker.

VoxCeleb structure –



Eventually our data contains:
Gender - [101,094 Male, 54,272 Female]
Age - [3,118 Teen, 44,434 Adult]
Total sum of data - 155,366

Preprocessing

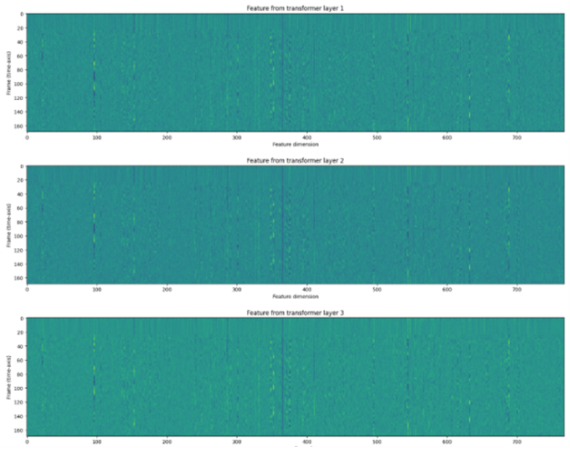
After gathering the data, we needed to represent it in the right way for it to be used as the features for our models. We started by cutting off the header of each wav audio file. The header of a wav file is 44 bytes long. To train our model we decided to use 3 seconds long wav files, so we cut each of the audio files we downloaded to 48,000 bytes, as we made sure the sampling rate of the audio is 16,000.
To prepare the wav files to be used as our features we needed to convert the wav to a different form of data, that can be used by our models to train and test, so we used Wav2Vec.

Wav2Vec

Torchaudio's Wav2Vec is a convolutional neural network (CNN) that takes raw audio as input and computes a general representation that can be input to a speech recognition system. The process Wav2Vec performs on the data looks like the following:

1. Extract the acoustic features from audio waveform. The returned features are a list of 12 tensors, each tensor is the output of a transformer layer.

   An example of the first 3 tensors in a plot -



2. Uniting the tensor layers to create the final tensor. Wav2Vec2 model provides method to perform the feature extraction

   and unite them in one step with the following command -

```
with torch.inference_mode():
    emission, _ = model(waveform)
```

   The emission is the tensor vector we will use to represent each wav file in our model.

Once we had the tensors for each wav, we wanted to combine each tensor with its matching labels for gender and age, and then unite all our data together. To do that we used the Pickle library.

Pickle

The Pickle module implements binary protocols for serializing and de-serializing a Python object structure. "Pickling" is the process where a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, where a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.
After combining each tensor with it's labels in a tuple, we created a list that unites all the tuples and used pickle to serialize it. This allowed us to gather all our data online and then to only

download the pickle file to train and test our model locally.

Load data

Once we downloaded the pickle, we wanted to arrange our data in a way that will allow us to create two separate models, one to predict the gender of the speaker and the other to predict the age of the speaker. To do that we created three lists, when loading each tuple in the pickle we added the tensor to a list which we called X_data, as it gathers the features. We added all the gender labels to a list called y_gender, and all the age labels to a list called y_age.
To make sure our model only gets numerical values we converted the labels according to these dictionaries-

```
self.genders = {"male": 0, "female": 1}
self.age = {"teen": 0, "adult": 1}
```

Then to start training and testing our models we used the train_test_split function from sklearn.model_selection library to create train, test and validation.

We split the data in the following way-
Train – 70%
Test – 20%
Validation – 10%

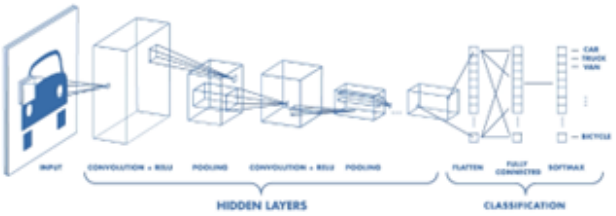
An example for splitting in the gender model-

```
X_train, X_test, y_train, y_test =
    train_test_split(np.array(X_data), np.array(y_gender), test_size=0.20)

X_train, X_val, y_train, y_val =
    train_test_split(np.array(X_train), np.array(y_train), test_size=0.15)
```

CNN Model

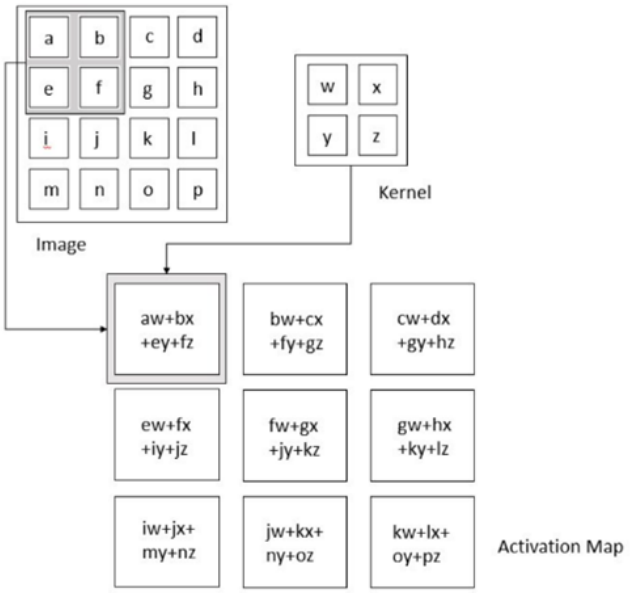
CNN is a convolutional neural network that specializes in processing data that has a grid-like topology, which is why it is commonly used for models based on images. A CNN typically contains convolutional layers, pooling layers, and a final prediction layer.

CNN example for image classification-



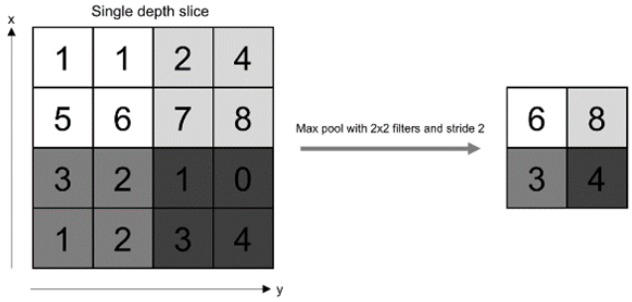
The convolution layer is the core building block of the CNN. It carries the main portion of the network's computational load. This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field (our data).

An example for how convolution works –



Using a kernel smaller than the input allows us to store fewer parameters, which not only reduces the memory, but also improves the statistical efficiency of the model, by detecting the meaningful information.
The pooling layer replaces the output of the network at certain locations by deriving a summary statistic of the nearby outputs. This too helps in reducing the spatial size of the representation, which decreases the required amount of computation and weights.
There are several pooling functions, the most popular process is max pooling, which returns the maximum output from the neighborhood.

An example of max pooling –

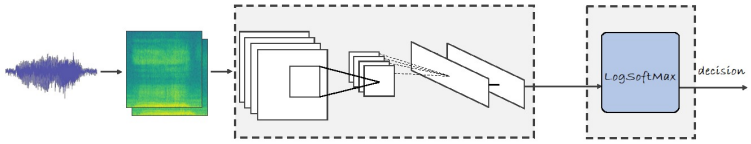


Since convolution is a linear operation and audio inputs (or images) are far from linear, non-linearity layers are often placed directly after the convolutional layer to introduce non-linearity to the activation map.
In our model we used. The Rectified Linear Unit (ReLU) has become very popular in the last few years. It computes the function f(k)=max (0, k). In other words, the activation is simply threshold at zero.

Another thing we used in our model is Dropout. The Dropout layer randomly sets input units to 0 with a frequency of rate (fraction of the input units to drop) at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by 1/(1 - rate) such that the sum over all inputs is unchanged.
The Dropout layer only applies when training is set to True such that no values are dropped during inference.

LogSoftmax

Softmax is a mathematical function which takes a vector of K real numbers as input and converts it into a probability distribution of K probabilities, proportional to the exponential of input numbers. After applying softmax, each component will be in the interval [0,1], and the components will add up to 1, so that we can interpret these values as probabilities.
Log softmax (the log of softmax function) is advantageous over softmax for improved numerical performance. logsoftmax also has the effect of heavily penalizing the model when it fails to predict a correct class.
After training the data on our models we came to the conclusion that adding two more layers to the gender model can improve the results drastically.



Training

To train our model we take the train set and use the CNN model we created to study the data in a way that will allow us to make prediction on future data that the model didn't encounter.
We started by using the class_weight function from sklearn.utils, because in both of our models the data distribution is unbalanced. The issue with that is that the algorithm will be more biased towards predicting the majority, and the algorithm will not have enough data to learn the patterns present in the minority class. The purpose of class_weight is to penalize the misclassification made by the minority class by setting a higher-class weight and at the same time reducing weight for the majority class.

```
class_weights = class_weight.compute_class_weight('balanced',
                                                  classes=np.unique(data.y), y=data.y)
class_weights = torch.tensor(class_weights, dtype=torch.float)

criterion = torch.nn.CrossEntropyLoss(weight=class_weights)
```

The loss function we used is Cross Entropy. Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label.

Each epoch

In each epoch we started with training the data using batches. Each batch is going through the network and returns an output, which is then updating the loss for that epoch, using our cross-entropy loss function. To update the network's weights, we used the Adam optimizer, Adam is a popular algorithm in deep learning because it achieves good results fast and is suited for problems with a lot of data.
Then we moved on to validation and test, in which we used batches again (to increase the speed of iterating over the data and computing the results).

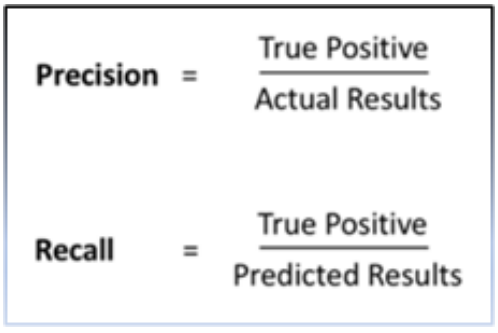
After using our network to make the prediction we updated the loss and calculated the f-score for the epoch. To be able to calculate the f-score we needed to "fix" the output from the network, as we use logsoftmax. Using a function to get the index of the maximum argument from each sample, we were able to get the prediction from the output.

```
# f1 score
f1 = f1_score(y_true, y_pred, average=None)
f1_avg = f1_score(y_true, y_pred, average='weighted')
print("f-score: ", f1, "  f-score avg: ", f1_avg, "\n")
```

The reason we calculated the f-score of each label on every epoch, is to get a better sense of the quality of the model then we could have gotten from only calculating the accuracy. The reason for this is that in both of our models the data is divided in a very unbalanced way, and f-score is a good method for this kind of challenge because it returns the average of the precision and recall.

```
F1 = 2 * (precision * recall) / (precision + recall)
```

Precision (also called positive predictive value) is the fraction of relevant instances among the retrieved instances, which means how good the model is at predicting a specific category.
Recall (also known as sensitivity) is the fraction of relevant instances that were retrieved, which means how many times the model was able to detect a specific category.



Inference

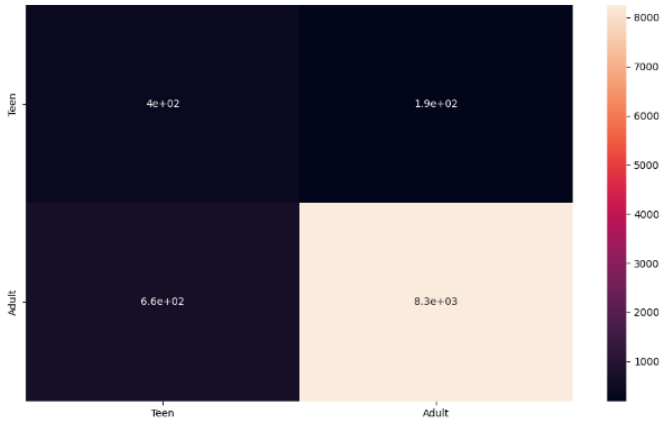
To make sure we eventually select the best models we created an inference testing. We gathered over a 100 different wav files, that the model has never seen before (not even the test set). Our goal was to make sure we choose the models that make good predictions on completely new data. For every model we tested we made sure to run over all the wav files and using the predictions we found the models that worked best.

To make sure we use our model in the right way to test the new wav files, the inference function is preprocessing the new data in the same way we preprocessed our datasets. We first load the model we want to test and make sure it's in evaluation mode (the training is set to false), then load the audio files and make sure the sampling rate is the same as it was in the training, if it isn't the same we use the resample function. Then we use Wav2Vec to convert the wav file to a tensor that represent it, and only then we can use the model to get the prediction. Here too we use the argmax function to make sure we don't return the numbers the CNN returns, but the label which has the highest probability.

```
model = torch.load("age_Binary_Model-e_11_Weights.pth")
model.eval()
waveform, sr = torchaudio.load(wav_path)

if sr != 16000:
    waveform = torchaudio.functional.resample(waveform, sr, 16000)

bundle = torchaudio.pipelines.WAV2VEC2_ASR_BASE_960H
wav_model = bundle.get_model().to(device)
embedding, _ = wav_model(waveform)

with torch.inference_mode():
    print(inf(embedding, model))
```

Results

Age − model **:**
Accuracy of the network = 91.04
f-score (weighted) average = 0.92
f-score by class = [Teen - 0.487, Adult - 0.950]



The best results for the age model were from epoch 11, with 0.0001 learning rate.

Gendermodel **:**
Accuracy of the network = 89.40
f-score (weighted) average = 0.89
f-score by class = [Male - 0.920, Female - 0.842]

The best results for the gender model were from epoch 36, with 0.0001 learning rate.

Challenges

The main challenge we had was the data. To create a good classification model, we need the data to have similar balance to the balance in the real world. We started the project with the Common Voice dataset, which caused a problem for us in both models. The number of males in this dataset is 4 times the number of females.

The other problem with this dataset was that the age label was classified to 9 groups, teens to nineties, but some of the age groups had no data in them at all and some had only a few samples in them.

We first tried to solve this problem by converting the age classification problem from a 9 classes problem to a 4 classes problem, as the first 4 classes had the most amount of audio files in them (teens to forties). But the problem was that this still left us with a very unbalanced data, and the amount of data was not big enough to make this kind of classification work well.

We also noticed that some of the models we created with these 4 classes had returned results that weren't too far from the truth, for example a person classified as 'teens' returned 'twenties' as a result, or a person classified as 'thirties' returned the result 'forties' or 'twenties'. This was better then the first version of 9 classes but still not good enough.

To deal with both of these problems we added data from a new dataset. Adding Voxceleb helped improve the result of the gender model significantly. The addition improved the balance of the data, the male were now only twice the amount of female. To try and make the results even better, we added two more layers to the gender model, as the classification was still difficult to do with this distribution of the data. Adding the layers helped increase the result of the model and got us the best results we got so far.

The problem we encountered with the new dataset was that it wasn't classified by age, so what we decided to do was to turn the four classes we had so far to a binary problem, with 'Teen' and 'Adult' classes. With this change we also needed to manually label a part of the Voxceleb dataset to increase the amount of data and add as many Teens as we could to make the classification easier. After labeling a big part of the data we had two classes for the age

classification, still very unbalanced but with more data for the model to study. Making these changes increased our results significantly and got us good model.

Conclusions

This project had a lot of challenges for us, as sound analysis is new to us and required studying a lot of new methods and information. At the end we got two new models to analyze sound, which can make a great change for a lot of business or individuals. We learned how to take a neural network base and make it something completely new that has a lot of potential.
Thank you for reading!