Algorithms for speech and natural language processing: TD #1

Due on February 25, 2019

ACHER Clément

Part 1

The goal of this part is to build a classifier that given a voice command with a fixed length can recognize the spoken command. To achieve the best results, quite a few tricks can be used at various levels.

Feature extraction and feature engineering

The first thing I have done is to remove the cap on the number of samples used. Note that for the validation and test sets, this is almost necessary : the proposed implementation will only extract samples from the first couple classes. They then can't be used to validate a model.

Tuning the speech features

It is pretty rare in ASR to feed a classifier with the raw signals. Features like MFCCs and Mel Filter Banks are more suited for such tasks. They however both rely on the choice of a few hyperparameters. To tune this features, I have mostly rely on the resources from this website[1] and the default parameters from another feature extracting library[2] that I had used before. Table 1 shows the huge impact the choice of these parameters has : it shows the accuracy obtained using the two models initially included (no model tuning done at this point).

Accuracy on val. set	MFCC - Initial parameters	Tuned MFCC - no normalization	Tuned MFCC - w/ normalization	Mel Filter Banks - w/ normalization
Log. Reg.	3.6	14.92	**30.46**	10.42
MLP	39.4	53.68	**65.92**	52.66

Table 1: Accuracy on vanilla models using different features

Normalization

Another factor of improvement that can be done at the feature level is about normalization. The idea is to get all the 13 features of the MFCC (or Mel filter banks) within the same range. As it can be seen on the Figure 1a, the MFCC features are heavily dominated by the first coefficient. The 3 datasets are then normalized using the mean and the standard deviations of each coefficients across the whole training set. Note that there are other possibilities to normalize the datasets. Normalized MFCCs can be seen on the Figure 1b. Normalization has a positive impact on the accuracy of the classifiers as shown in table 1. The MLP classifier also converges faster (21 iterations vs. 29).
As MFCC features were giving the best results, I have decided to only use these.

Data augmentation

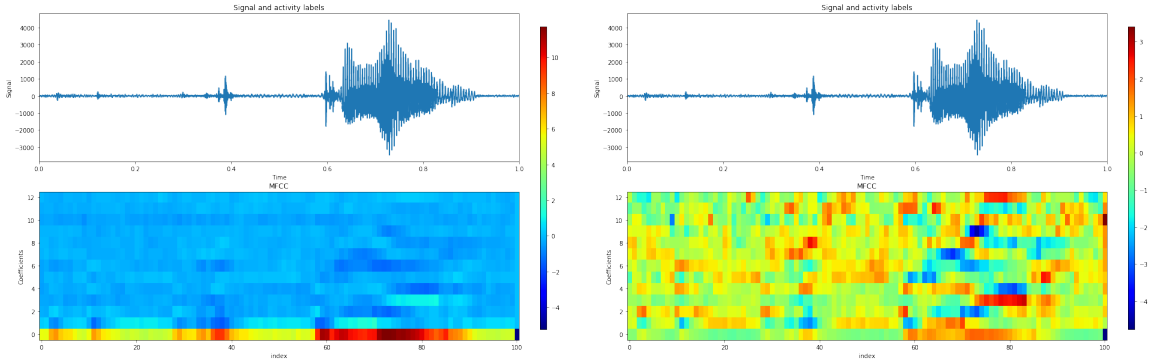
The dataset also provides various sounds that can be used to perform data augmentation. However, the voice command audio signal and the noise should not simply added : many noise are way louder than the voice command. Simply summing these signals makes the command inaudible. To weight the "quantity" of noise to add, I use the ratio of the RMS of the two signals. Augmenting the dataset does not seem to have a significant impact on the models.

Shuffling the samples

As the Python implementation to load the samples goes through the folders containing the commands one after the other, the training dataset is not shuffled. I have added an extra step to shuffle the datasets as this could potentially impact the training of the MLP (Pytorch dataloader already shuffles the samples).

[1] http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/
[2] https://python-speech-features.readthedocs.io/en/latest/

Problem 1 continued on next page...



(a) No normalization (b) With normalization

Figure 1: Raw signal (top) and MFCC feature (bottom) with and without normalization.

Model tuning

CNN-based models seem quite relevant for ASR in this setting. Even though we have samples with fixed length, the part with voice may be either at the beginning or at the end of the sample. CNNs are great at finding patterns even though they may not be at the same position every time, contrary to a simple dense network. This is the main reason why I have not spend much time on trying to fine tune the hyperparameters of the proposed models to rather focus on implementing a CNN using Pytorch.
Another advantage of CNNs is that we don't need to flatten the MFCC features before giving them to the model. We actually keep the structure.

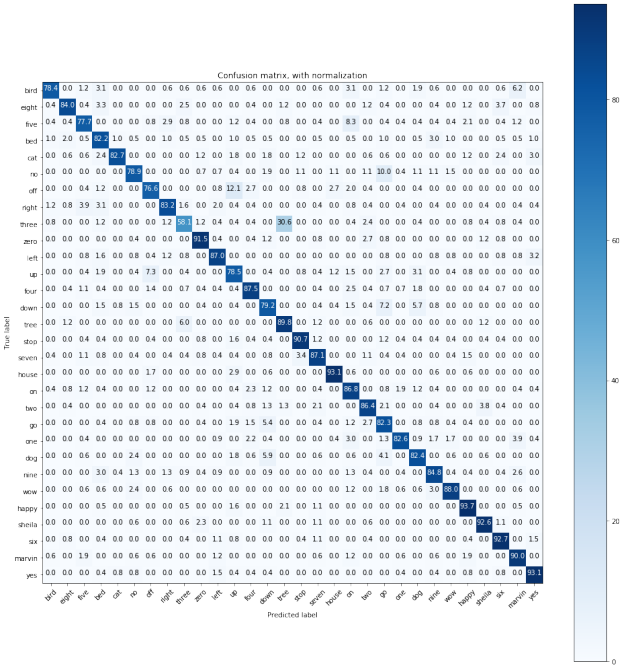


Figure 2: Confusion matrix for the CNN-based model.

This final model achieves an **accuracy of 83%** on the test set.

Figure 2 shows the confusion matrix on the test set of this model. We can for instance see that the word

`three` is missclassified about 30% of the time as `tree`. `go` and `no` are also swapped which can be easily understood, but there are some other errors that are harder to grasp : `off` gets missclassified as `up` 12% of the time while the corresponding sounds seem quite different from each other.

Part 2

Question 2.1
The numerator is a sum of positive integers, so the WER can't be negative. It can be greater than 100, for instance if the reference is 'I like apple' and hypothesis "She really loves dogs and cats", the WER is 200.

Question 2.2
`train_labels.count(label) < nb_ex_per_class` ensures that the dataset used for training is balanced and then that the prior probability of each word is equal.

Question 2.3
We have the following output:

```
True sentence:  go marvin one right stop
Predicted sentence with greedy search:  go marvin on five stop
```

There are 2 substitutions out of the 5 words. The WER is then $\frac{2}{5} = 0.4$.

Question 2.4
In the bigram model, we have:

$$P(W) = \prod_{k=1}^{T} P(W_k|W_{k-1})$$

Question 2.5
The bigrams are stored in matrix $(P(w_k = j|w_{k-1} = i))_{i,j}$. To compute these probabilities, we go through all the sequences of the training set, and increment a the count of the encountered bigrams. The probabilities are then obtained by normalizing by the sum of the rows.
On top of that, there are two extra tricks : Laplace smoothing (i.e. add 1 to every bigram counts) is used to avoid 0-counts. We also add an extra "starting word" at the beginning of each sequence so that there is a bigram for the first word of the sequence. We can notice this way that `go` is often the first word of the sequence.

Question 2.6
Increasing N leads to a more accurate language model. However the number of entries in the transition matrix is M^N where M is the number of different possible words (31 here). The space complexity increases exponentially, and as the training dataset is rather small, it makes the matrix very sparse.

Question 2.7
We denote L the length of a sequence ($L = 5$), B the beam width and M the number of different words. The time complexity is then $\Theta(LBM(1 + \log(BM)))$ considering that the sorting algorithm has a complexity of $\Theta(n \log(n))$.

Question 2.8
Let's denote $V_{k,j}$ the probability of the most probable sequence $P(w_1, ..., w_k, x_1, ..., x_k)$ that has j as its final state, A the transition matrix and $\boldsymbol{W}$ the set of possible words. We then have:

$$V_{k,j} = \max_{j' \in \boldsymbol{W}} (P(x_k|j) \cdot A_{j',j} \cdot V_{k-1,j'})$$

From this, we can simply derive the complexity of the algorithm : using the previous notations $O(LM^2)$.

Comparing the different decoders

	Training set	Test set
Greedy	0.181	0.190
Beam search	0.050	0.046
Viterbi	0.068	0.064

Table 2: WER for the different decoders

Table 2 contains the WER corresponding to the different decoders combined with the CNN-based model introduced before. As expected, beam search and Viterbi decoders perform much better than a simple greedy decoder. However it is a bit surprising to see that beam search gives slightly better results than the Viterbi decoder : Viterbi is an exact method while beam search is not. The difference is pretty small, the difference may not be significant.

Question 2.10
Using plain N-grams only based on counts can raise issues due to sparsity. If we use the maximum likelihood estimator to compute a bigram model, if a certain bigram was not founds in the training set, it won't ever be predicted at inference time. To tackle this, smoothing techniques can be applied, such as the "Laplace" smoothing that I have used which simply consists in adding one to all bigram counts.
Another technique consists in computing a weighted average of the bigram and the unigram $p_{interp}(w_i|w_{i-1}) = \lambda p_{ML}(w_i|w_{i-1}) + (1-\lambda)p_{ML}(w_i)$. It requires to tune the lambda parameter and it does not solve the issue of sparsity if the word does not appear in the training set, which is the case for some commands here. I have tested this method and it doesn't give any better results.

Question 2.11
To optimize jointly an acoustic model and a language model, we could use a different set of algorithms to perform end-to-end speech recognition. For instance, RNN-based models to perform sequence-to-sequence like in [1].

References

[1] Rohit Prabhavalkar, Kanishka Rao, Tara Sainath, Bo Li, Leif Johnson, and Navdeep Jaitly. A comparison of sequence-to-sequence models for speech recognition. 2017.