HW1

Gabriel ROMON

I Classification of single voice commands

Because of the constraints `len(valid_labels) < 1000` and `len(test_labels) < 1000` the original validation and test set only contained 4 classes out of the 30 present in the dataset. For this reason accuracy on those sets is not representative of the performance of the model as a whole. I decided to include all the validation and test data listed in the files `validation_list.txt` and `testing_list.txt`. Moreover I increased the cap on training samples per classes to 1000 instead of 300. My training, validation and test sets have size respectively 30.000, 6798 and 6835 with histograms shown in Figure 1. The data is fairly balanced in each dataset.

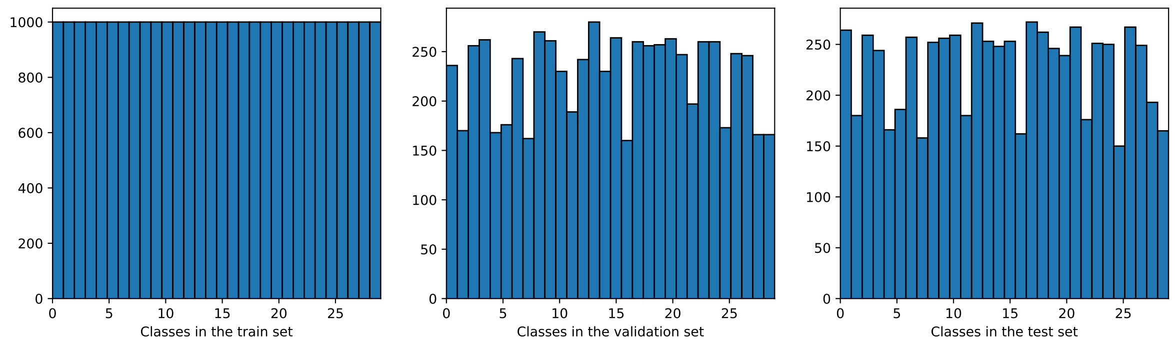
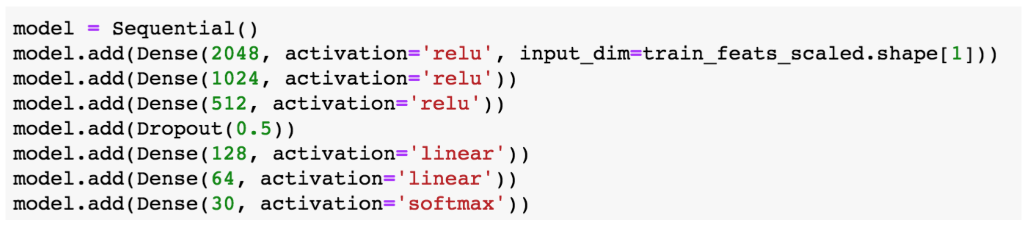


Figure 1: Histograms of classes

I chose the hyperparameters of the Mel-filterbanks and MFCC functions according to the default values suggested by the author of the package `python_speech_features` in the blog post [1]. I experimented with two models: logistic regression and multilayer perceptron. They were trained on a Google Cloud Instance with 8 cores, 25 GB RAM and a K80 GPU.

Because of the dimension of the training set I was not able to use `scikit-learn`'s `LogisticRegression`, so I used the SGD version instead. L1 regularization was added because the dimension of the features is quite high compared to the number of samples. I used 3-fold cross-validation to find a good value of the regularization parameter. Besides, I enforced early stopping to mitigate overfitting.

I used Keras to implement the multilayer perceptron architecture shown in Figure 2. I did not spend much time fine-tuning it but it yields satisfactory results. It was trained using categorical cross-entropy as loss and Adam as optimizer. Batch size was set to 1024.



```
model = Sequential()
model.add(Dense(2048, activation='relu', input_dim=train_feats_scaled.shape[1]))
model.add(Dense(1024, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='linear'))
model.add(Dense(64, activation='linear'))
model.add(Dense(30, activation='softmax'))
```

Figure 2: MLP architecture

Results are reported in Table 1. Logistic regression does not achieve accuracy higher than 33% on the validation set, no matter the variant. Note that scaling the features significantly improves performance in each case. Besides, it is fairly time-consuming to train because of cross-validation.
On the other hand, the neural network is much faster to train and yields much better accuracy ($\sim$ 75%). Scaling features provides an accuracy boost with mel features, but no significant improvement for MFCC. Besides, the MLP overfits much more when the features are scaled.

Method	Feature dimension	Model	Training accuracy	Validation accuracy	Time to train
Mel without deltas	2626	Logistic SGD	30.73	27.27	327 s
		Logistic SGD scaled	38.37	32.67	373 s
		MLP	74.19	64.23	63 s
		MLP scaled	86.52	75.64	34 s
Mel with deltas	7878	Logistic SGD	30.58	25.59	937 s
		Logistic SGD scaled	38.37	27.99	1072 s
		MLP	76.12	73.29	70 s
		MLP scaled	90.61	75.90	42 s
MFCC without deltas	1313	Logistic SGD	26.30	24.84	214 s
		Logistic SGD scaled	37.50	28.39	217 s
		MLP	82.54	75.03	53 s
		MLP scaled	94.03	72.16	50 s
MFCC with deltas	3939	Logistic SGD	28.12	25.75	554 s
		Logistic SGD scaled	39.73	28.89	590 s
		MLP	87.90	**76.49**	63 s
		MLP scaled	94.99	76.08	44 s

Table 1: Results of the experiments

The best model (MFCC with deltas and unscaled features) achieves 76.00% accuracy on the test set.

II. Classification of segmented voice commands

Q2.1 Since S, D, I, N are non-negative quantities, WER is non-negative. Since the ASR may produce an arbitrary number of insertions, WER may be arbitrarily large.

Q2.2 Because of the line `elif train_labels.count(label) < nb_ex_per_class` the training set of the discriminator was constrained to be perfectly balanced, thus yielding a uniform prior on the words.

Q2.3 In the example from the original notebook, the true sentence is `go marvin one right stop` and the predicted one is `happy five one two two`. Except for the word `one`, there are 4 incorrect substitutions, hence a WER of $\frac{4}{5} = 0.8$.

Q2.4 The Bigram approximation of the language model writes as

$$\mathbb{P}(w_i \mid w_{i-1}, ..., w_2, w_1) = \mathbb{P}(w_i \mid w_{i-1})$$

Q2.5 I constrained my implementation to the case $N = 2$. The transition matrix `T` is built such that `T[i,j]` approximates $\mathbb{P}(j|i)$. A simple estimate of this probability is given by counts: $\frac{\#(i,j)}{\#i}$. However this is not well-defined when the bigram (i, j) does not appear in the training samples. Consequently I used 1-Laplace smoothing, which adds 1 to every bigram count. The estimate becomes

$$\frac{\#(i,j) + 1}{\#i + |\text{vocabulary}|}$$

Q2.6 Higher values of N yield more realistic models. However they lead to an exponential increase in memory usage, which may be mitigated by using sparse data structures.

Q2.7 Let b denote the beam size, L denote the length of the sequence of commands and V the size of the vocabulary. The time complexity of the Beam-Search algorithm I have implemented is $O(bL\,V \log V)$. However it could easily be reduced to $O(bLV)$, see the comment in my code. The memory footprint is at most $O(bV)$.

Q2.8 Let $v_t(j)$ denote the probability of being in state j at step t. Then

$$v_t(j) = \max_{1 \leq i \leq N} v_{t-1}(i)\mathbb{P}(j|i)\mathbb{P}(X_t|j)$$

Note that $\mathbb{P}(j|i)$ is given by the transition matrix and $\mathbb{P}(X_t|j)$ is produced by the discriminator.
The Viterbi algorithm has time complexity $O(LV^2)$ and memory complexity $O(LV)$.

Q2.9 Both beam-search and Viterbi algorithms will be biased by words or bigrams that do not occur in the training samples, thus resulting in a systematic error when decoding.

Q2.10 The influence of rare words can be mitigated by using Laplace smoothing.

Results
I used the best model from the previous part as a discriminator. Training and test WER are shown in Table 2.

Method	Parameter	Train WER	Test WER	Time to train
Beam Search	1	0.4162	0.4067	56 s
	2	0.4145	0.4025	57 s
	3	0.4024	0.3899	56 s
	4	0.3814	0.3676	56 s
	5	0.3351	0.3409	56 s
Viterbi		0.2779	0.0904	56 s

Table 2: Results of the experiments

References

[1] James Lyons. MFCC tutorial. `http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/`.