MVA 2020 Speech-NLP Homework

Dimitri Meunier

Preliminary Remarks. In the first part, we modified the code that creates the train, validation and test sets to contain the same number of elements per classes. The way it was originally implemented only put 5 classes (out of 30) in the test and validation sets leading to unreliable validation scores. For question 1.1 to 1.4 we evaluated the scores with the MLP by default and a learning rate of 0.001.

Question 1.1 Figure 1 displays the grid search we performed over lower and upper frequency values. For each frequency range, we generated new MFCC features for the train and validation set, trained the Perceptron on the train set and evaluated its accuracy on the test set. The heatmap clearly shows that the higher the frequency range, the higher the accuracy. This does not concord with human speech characteristics as we are only used to listen to lower frequencies and do not recognize well high frequencies. However, humans have other contextual information to understand speech which may explain the observed difference. The range we selected is [20,8000], for the next questions the frequency range was thus set to this value. Since the dataset is sampled at 16kHz, we cannot go above half of the framerate (8000).

Question 1.2 Figure 2 displays the grid search we performed over the parameter nfilt of the mel-log filterbanks. The optimal number of filters is 7 and going above 10 strongly reduces the accuracy. Since MFCC also uses the nfilt parameter we also run a grid search and the optimal number of filters is 20. For ncepstral of the MFCC the optimal number of cepstral coefficients is 8 (figure 2). In theory, the decomposition in cepstral coefficients allows to separate the non informative glottal source from the vocal tract. The glottal source appears in more distant coefficients, that is why we only retain 8 cepstral coefficients.

Question 1.3 The delta and delta_delta hyper-parameters represent the first and second derivatives of the cepstral coefficients. Our evaluation shows that only the first derivative seems to be a relevant feature. This can be explained by the fact that the signal obtained after the cepstral decomposition seems to be very flat, so the second derivative may take very small values and bring only additional noise.

Question 1.4 For the next questions, we fix the hyperparameters we chose for the generations of the MFCC features and we experiment with different pre-processing techniques. We tried to standardize each features individually. We also tried to add centered Gaussian noise with different standard deviations. Both approach lead to a decrease in accuracy on the test set, we therefore did not use them for the next questions.

Question 1.5 We know play with the hyperparameters of the Perceptron. Reducing the learning rate from 0.01 to 0.001 had a real positive effect. By default the optimiser is ADAM, we change it to SGD with different learning rate schedulers and different learning rates. The problem with SGD is that it is very slow to lead to convergence, it therefore performed worse than ADAM. Keeping ADAM, we then experimented with different activation functions: ReLU, tanh, logistic and identity, the best score was attributed to ReLU. We then changed the internal architecture of the Multi Layer Perceptron going from a single layer with 100 neurons to a three layer Perceptron with 250 neurons each improved the accuracy by almost 8%. Going above three layers did not improved accuracy as it started to overfit. The main flaw of the scikit-learn implementation of the MLP is that is does not support more advanced regularization techniques such as dropout. We therefore compare our last model to a Keras fully connected classifier with droupout. We did not managed to further improve the accuracy with Keras and therefore stayed with the scikit-learn architecture. Let's remark that using ADAM with ReLU activation is not surprising as it is often what works best in practice.

Question 1.6 The final set of hyperparameters we selected for the MFCC features is *nfilt=20*, *ncep=8*, *lowerf=20*, *upperf=8000*, including first derivative and no second derivative. We did not include noise and normalization as it seems ineffective. The Neural Network we selected is a Multi-Layer Perceptron with three hidden layers of 250 neurons each, trained with ADAM and a learning rate of 0.001. The final scores we reached is 74.1% on the validation set and 64.7% on the test set. The confusion matrix is displayed in figure 3. The biggest confusions being between *three* and *tree* and between *go* and *no* which is not surprising given the similarities between the words.

Question 2.1 In the greedy approach we want to maximise $P(X|W)$ in W where X is a spoken word and W is a written word from the vocabulary. However, the greedy search decoder function uses an argmax over the output of the Neural Network (posteriors neural mfcc) which are the $P(W|X)$.

```
predicted_sequence = [np.argmax(s) for s in data]
```

We therefore assume that $P(W|X) = P(X|W)c$ where c does not depend on W. But since $P(W|X) = \frac{P(X|W)P(W)}{P(X)}$ it means that all the $P(W)$ are equal i.e. we assume a uniform distribution over the W_i.

Question 2.2 All the components of the WER (S,D,I,N) are non-negative values, it is therefore non-negative. Since the hypothesis and reference sequences does not necessarily have the same size, the WER can grow arbitrarily large. For example, if the reference is (dog,cat) and the hypothesis is (dog,cat,bird,bird,bird) the WER is 150.

Question 2.3 The reference is "go marvin one right stop" and the hypothesis is "bed marvin one nine stop". They have the same length therefore there is no deletion and insertion and there is two wrong words, thus, the WER is $100 * 2/5 = 40$.

Question 2.4 For any sequence of words $w_1^n = (w_1, \cdots, w_n)$. The chain formula gives $P(w_1^n) = \prod_{i=1}^n P(w_i|w_1^{i-1})$ where $P(w_1|w_1^{1-1}) = P(w_1)$. The Bigram approximation is $P(w_n|w_1^{n-1}) = P(w_n|w_{n-1})$ (Markov dependence of order one). The formula is therefore $P(w_1^n) = \prod_{i=1}^n P(w_i|w_{i-1})$. With Ngram, we assume a Markov dependency of order N-1: $P(w_n|w_1^{n-1}) = P(w_n|w_{n-N+1}^{n-1})$ and the approximation is $P(w_1^n) = \prod_{i=1}^n P(w_i|w_{i-N+1}^{i-1})$.

Question 2.5 The immediate way to compute Bigram transitions is to instantiate a matrix T of sized $V * V$ where V is the size of the vocabulary. We iterate over the train set and every time a transition between words i and j is seen, we add 1 to the entry T_{ij}. At the end if we normalize each row, it gives for each word a transition probability towards the other words in the vocabulary. This approach is simple to implement but highly memory inefficient as the matrix is extremely sparse. Another drawback is that we have to know in advance the vocabulary. We suggest another implementation: we build a **directed graph** where each node is a word and every (directed) edge between words counts the number of times the transition has been seen. The resulting graph is a Python class that we called **Automaton** (it is basically a dictionary of dictionaries with facilities to access information). It is memory efficient and does not imply to know the vocabulary in advance. It also scales better to Ngram as we just have to tell the graph to change the nodes to tuple of N-1 words. On the other hand, the scaling to Ngram for transition matrices would imply to use a tensor of dimension N almost uniquely filled with 0, this would quickly fill the whole memory of the computer. Unfortunately, using such a directed graph would imply an heavy use of loops when we implement the Viterbi algorithm or the Beam search algorithm. This would be fine with a low level language but it is dramatic in Python. We therefore also implemented a method to convert the graph to an array, sacrificing memory efficiency to improve computational speed with Numpy efficiency. There is also a possibility to take advantage of both Numpy and a memory friendly implementation using the sparse matrix class of Scipy but we did not have time to try it.
Let's remark that it is extremely important to use **log** and additions instead of probability multiplications to avoid underflow in the next algorithms.

Question 2.6 Increasing N (i.e. increasing the context) leads to more coherent sentences. However, a larger N can also creates corpus dependencies that will not generalize well. For example a language model trained with $N = 10$ on Shakespeare's texts would only generates sentences that are exactly like in the corpus. It also increases the memory consumption to store the Ngrams transitions.

Question 2.7 If K is the beam size, V is the vocabulary size and T is the length of the sequence of observations, the time complexity is $O(TKV)$ (for each iteration and for each K current best candidates we compute V scores); the memory complexity is $O(KT)$ (if we ignore the fact that we also have the language model in memory).

Question 2.8 We denote by $v_k(j)$ the probability of being in state j at step k, we have $v_k(j) = v_{k-1}(j')a_{j'j}b_j$ where $a_{j'j}$ is the probability of transition from state j' to state j (computed from the language model) and b_j is the state observation likelihood of the observation at time k given the state j (computed with the Neural Network). The Viterbi algorithm has a time complexity of $O(TV^2)$ and a memory complexity of $O(TV)$.

Question 2.9 Results are presented in tables 1 and 2. In theory, Viterbi is the best model but we see that with a Bigram language model it only wins with a thin margin over BeamSearch. Increasing the beam size for the Beam Search is beneficial. We also see that a Trigram language model (and higher) gives better results, the BeamSearch with only a memory of 3 beats the Viterbi from the Bigram model. However 5gram is too much and we see a decrease in accuracy on the test set. We should look at an implementation of the Viterbi algorithm for Ngram with N>2 as it will likely improve performances.

Question 2.10 If the test set includes Bigram or single words that are not included in the train set for the language model, it will induce systematic errors as their will never be selected by Viterbi or BeamSearch. Fortunately, we checked that every Bigram and words from the test set are also in the train set. Therefore to detect systematic errors du to the language model, we focus on errors made by the Viterbi but not by the Greedy. For exmaple for ['happy', 'tree', 'on', 'happy', 'house', 'wow'] (sentence 319), at the end the Greedy predicts the good words but the Viterbi predicts "house","on". The Neural Network is fairly confident that the last word is "wow" but since the Bigram probability of ("house","wow") is 0.19 and the one of ("house","on") is 0.44, the langage model creates an error.

Question 2.11 We can mitigate the rare word phenomenon using Laplace smoothing. We could also use linear interpolation or back-off strategies but they require a validation set to tune their hyperparameters. For OOV words, we can also assign to them a small probability. We run again the experiments with a Laplace smoohing (also called add-1 smoothing) of the Bigram model and add-k smoothing for different values of k. Results are presented in table 3. It badly decreases the performance. In fact add-k smoothing is not a really smart smoothing approach and it is preferable to use Katz-backoff or Kneser-Ney smoothing.

Question 2.12 It is possible to learn directly an acoustic model and a language model using a RNN network like a Bi-LSTM. The state-of-the-art models for this task are not RNN anymore but networks with Attention.

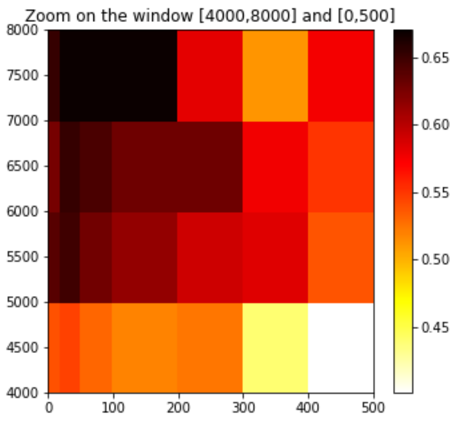

Figure 1: HeatMap of the grid search performed over the frequency range for the MFCC. Each cell colour is proportional to its accuracy on the validation set (from white to black). The best accuracy was obtained with the range [20,8000].

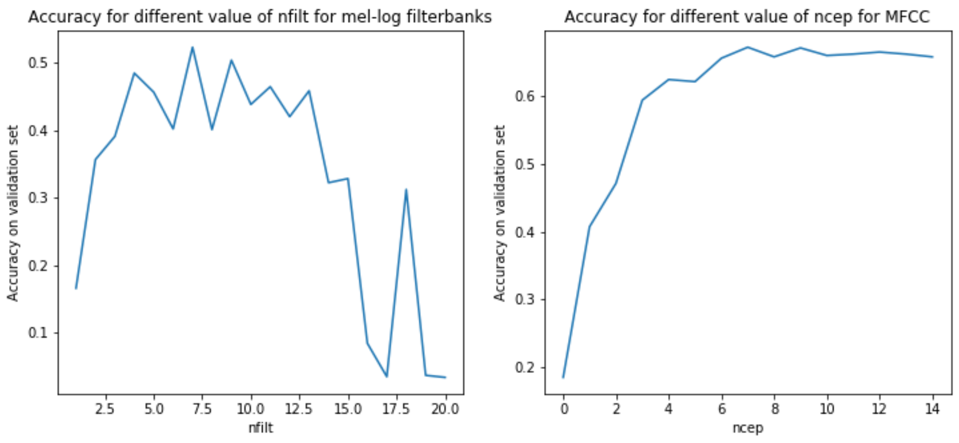

Figure 2: Grid search for nfilt of meg-log filterbanks (left) and ncep of MFCC (right).

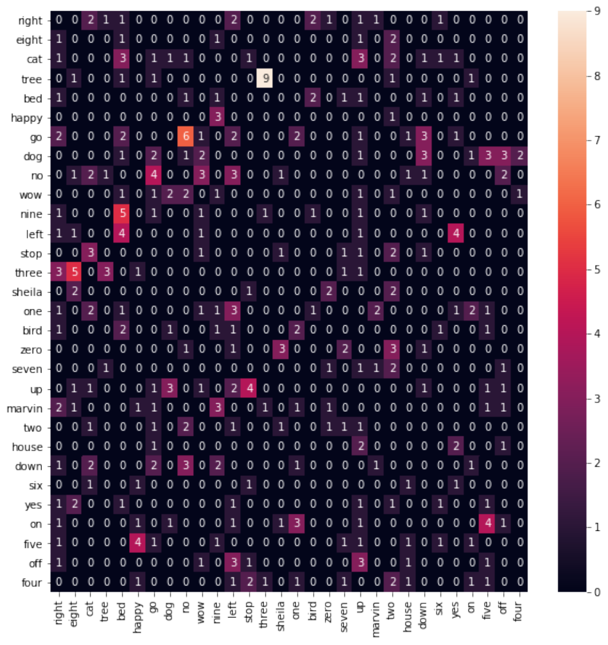

Figure 3: Confusion Matrix for the training of the Neural Network with optimal hyperparameters (the diagonal was intentionally filled with zeros).

Algorithm / LM	Bigram	Trigram	4-gram	5-gram
Greedy	39.9	***	***	***
BeamSearch (k=3)	13	11.7	11.7	11.4
BeamSearch (k=4)	12.9	12.1	11.5	11.3
BeamSearch (k=5)	12.5	**11.7**	**11.5**	**11.1**
Viterbi	**12.2**	***	***	***

Table 1: WER scores (in %) on a subset of the train set (200 sentences).

Algorithm / LM	Bigram	Trigram	4-gram	5-gram
Greedy	37	***	***	***
BeamSearch (k=3)	15.7	13.9	13.7	15.5
BeamSearch (k=4)	12.6	12.1	11.8	13.3
BeamSearch (k=5)	12.6	**11.9**	**11.5**	**13.2**
Viterbi	**12.1**	***	***	***

Table 2: WER scores (in %) on the test set.

Algorithm / Smoothing	Add-1	Add-0.5	Add-0.1	Add-0.01
BeamSearch (k=5)	22.8	21.7	19.9	19.4
Viterbi	22.8	21.6	20.9	18.5

Table 3: WER scores (in %) on the test set with different smoothing