Predicting the Presence of Seizing from a Single LFCC Feature Vector

Elliott Krome
Department of Electrical and Computer Engineering, Temple University
krome@temple.edu

Introduction: This is a binary classification problem whose goal is to map a single feature vector to an outcome: {seizure, no seizure}. The LFCC (linear frequency cepstral coefficients) feature vector is extracted from a single window of time from a single channel of an electroencephalogram, or EEG. An EEG is a multi-channel signal which describes the electrical activity in the brain via voltages measured in a variety of locations on the scalp.

Seizures do not occur very often - this is good from an empathetic perspective, but presents a significant problem from an applications perspective. However, here we greatly simplify this problem in training and validation by using data sets where the binary priors are approximately equal – the LFCC feature vectors extracted from moments of seizing are present in relatively very high proportion compared to those extracted from more typical EEG behavior.

This simplifies the problem insofar as the training feature vectors are (hopefully) more likely to be around an ideal decision surface. However, generalizability of training using this overrepresentation of seizure data is questionable, especially given the lack of temporal or spatial (other channels) context. This is a naïve approach. Because these are isolated feature vectors that are not related temporally or spatially, there is no need to model sequential structure, and simpler models such as random forest and multilayer perceptron are sufficient. Two learning algorithms were used: a multilayer perceptron (MLP), and a random forest.

Random Forest: Random forests use a set, or ensemble, of decision tree learners to overcome the problem of high variance / overfitting present in the use of a single decision tree. Two different constraints are placed on the training of an individual tree to encourage overall robustness: (1) an individual tree only sees a subset of the training examples, and (2) each individual split in the tree is only made based on a subset of the features. This first constraint, known as bootstrap aggregating (bagging), resamples the data into many smaller subsets, allowing for each sub-model to get its own "view" of the data before model-averaging – this serves to reduce the high-variance problem of individual decision trees. The second constraint, known as the random subspace method, encourages independence from tree to tree – each branch is made by using only a randomly draw subset of features. These two constraints are similar in that each limits the individual tree's access to data to encourage the aggregated learned representations to be more diverse, and hence more generalizable. A (perhaps too poetic) sidenote: the metaphor of a forest is doubly apt: where each tree in a physical forest only has a limited view of the input to capture (the sun), each tree in a random forest only has a limited view of the source of discriminative power (the training set), and in both situations the diversity of the aggregate forest serves to maximize capture of input "power".

The implementation used comes from Python's sklearn library. It constructs 100 individual decision tree classifiers based on the training data, and outputs an overall classification based on the averaging of these individual trees. The size of the dataset was enough to support this quantity of trees, and training took minimal computation time. this The maximum number of features looked at when determining a single split for a single tree was 5, which is approximately the square root of the feature vector size 26. It should be noted that this number of features was expanded if a suitable split was not found within the initial set. The minimum "leaf size" after each split was 9 – the larger this value, the shallower the trees. Unlike in the usual decision tree paradigm where increased depth of the tree is associated with overfitting, in random forests the overfitting of the individual trees is, in aggregate, an asset. The primary cost of increasing tree depth is increased computation time, but that was not a significant problem.

Multilayer Perceptron: This network, which uses Python's Keras library, had three hidden layers. The size of these hidden layers tapered at each step: 26 input nodes → 22 hidden nodes → 26 hidden nodes → 10 hidden nodes → 2 output nodes. Each of these hidden layers had a rectified linear unit (ReLU) activation function as input. The final output layer had a softmax activation function as input. It was trained using a binary cross entropy loss function, with the Adam optimizer.

Each of the hidden layers was trained using dropout, which randomly nullifies certain nodes for each batch. For every batch, for every layer, 20 % of the units were nullified. This prevents the network from relying on specific information paths – it encourages learning multiple representations, and in this way is similar to the random subspace method used in the random forest algorithm. It is often argued that this dropout technique is most beneficial with complicated networks – this would make sense as it allows larger number of pathways of information and multiple representations of the crucial information. However, in this case, the tapered structure proved sufficient. Following Occam's Razor, if more information does not provide much benefit, the simpler structure should be chosen. To be explicit: a network with larger hidden layers could have been employed, but this was experimentally found to not be particularly useful.

To decide the number of epochs (loops over the training data) to use when training this algorithm I examined the error rate trends over the first 125 epochs for both open loop and closed loop training. This is shown in figure 1. Figure 1 shows that the trend of the error rate in open loop training stops decreasing around epoch 25, while the closed loop continuously decreases as the number of training epochs increases. This gives an indication that the algorithm begins to overfit the training data somewhere around epoch 25. Because of this, I limited the number of training epochs to 23.

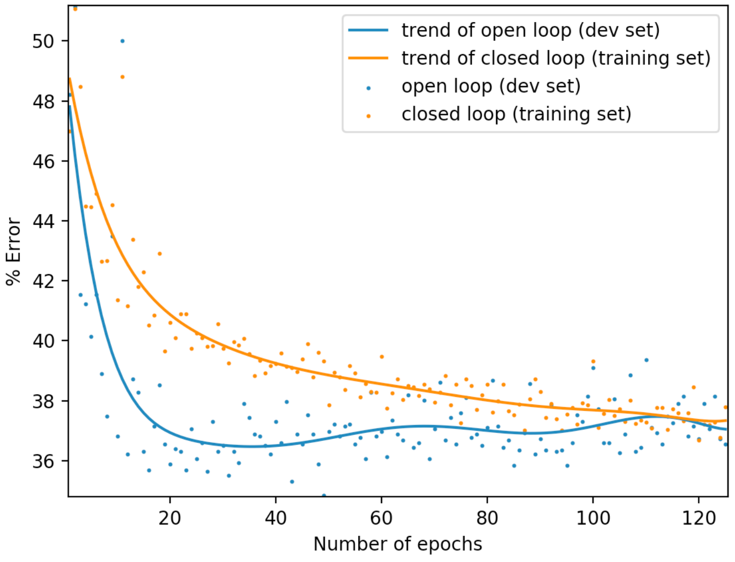

Fig. 1 Error rate in open- and closed-loop vs. number of epochs

Results:

Table 1 shows the performance of each of the algorithms on the various datasets. Performance on the eval set was poor for both algorithms. It is possible that the evaluation set was significantly different than the training or dev sets. Perhaps the prior probabilities are closer to the real-world probabilities, as discussed in the introduction. While the multilayer perceptron performed significantly better than the random forest on train and dev sets, the random forest proved to generalize slightly better to the eval set.

	Data Set		
Algorithm	**Train**	**Dev Test**	**Eval**
Random Forest	38.18%	36.50%	45.91%
Multilayer Perceptron	39.40%	36.26%	46.68%

Table 1. Performance of the algorithms on each dataset.

Conclusions: The performance of both algorithms is fairly comparable in this experiment – neither did much better than chance. It is not clear how to improve the training of either the random forest, outside of chasing the diminishing returns of adding more trees and increasing tree depth, or the training of the perceptron, outside of providing more and larger layers, although there are some interesting tricks that we can play with the training data, such as augmentation via additive Gaussian noise. This difficulty suggests that if we are to classify this kind of data it would be advantageous to make use of sequential structure.