CS 7641 Machine Learning
Assignment 1

David Yun

September 23, 2018

Abstract

The implementation (in Python2.6 or Python3, depending on the algorithm), and basic fundamentals behind the following Machine Learning Algorithms will be discussed in detail:

1. Decision trees with some form of pruning
2. Neural Networks
3. Boosting
4. Support Vector Machines (SVM)
5. k-Nearest Neighbors (kNN)

The full code files are available on github[1] Please refer to the README.txt file for concise instructions on how to run the python files associated with each of the aforementioned algorithms. Further details behind the datasets, including where to download them, are provided in the README.

Contents

[1]David Yun's Github: `https://github.com/tree-fiddy/Assignment_1`

1 Datasets

The two datasets used are

2 Decision Trees

The implementation of Decision Trees was borrowed from my previous coursework in CSE6242 (Data Visualization) with minor tweaks. **Cross-Validation** is done without the need for a training-validation split here because of the use of Random Forests, which is a collection of decision trees. Random forests allow us to create a robust model with less variance which results in a reduction in overfitting[2]. With proper pruning (Entropy thresholds & Max Depth), one can further reduce the variance of the model and further decrease the chance of overfitting. The **dataset** used is the UCI Credit Approval Dataset[3]. Some small adjustments were made to the dataset, such as removing rows where data was missing.

2.1 Learning Curves Analysis

The accuracy of our decision tree relies on our tolerance for pre-pruning. In essence, one must specify *when* to stop growing the tree. On one extreme, we can set a condition to stop growing the tree once we reach a *pure*[4] leaf. However, this will likely make the tree longer than need be. A more depth-restrictive approach is to set an arbitrary threshold for Entropy in our Decision Tree, where we will halt further node and leaf generation if the current node produces leaves with Entropy values less than our specified threshold. To ensure the efficacy of our pre-pruning process, we also incorporate a limit to the depth of our tree to 5. This will handle extreme cases where our dataset is so large, that splitting across many features is possible. This also has the added benefit of dramatic performance improvement, without sacrificing accuracy, when compared to not setting a depth and letting the tree grow indefinitely. The performance of our decision tree, while varying threshold levels is highlighted in Table 1.

As you can see, the model's accuracy reached an asymptote of around 50% accuracy, which indicates that setting a higher threshold for Entropy renders this model's efficacy no better than a coin flip. Thus, setting a pre-pruning specification is quite important

[2]By averaging several decision trees, the result is a less likely chance of overfitting

[3]UCI Credit Approval Data Set: `http://archive.ics.uci.edu/ml/datasets/credit+approval`

[4]A leaf is said to be "pure" if the leaf contains homogenous labels. By extension, this equates to a situation where Entropy equals 0.

Table 1: Pre-Pruning Spec Performance (forest size = 50)

Entropy Threshold	Model Accuracy
0.0	0.86
0.1	0.86
0.2	0.60
0.3	0.57
0.4	0.55
0.5	0.55
0.6	0.54
0.7	0.54

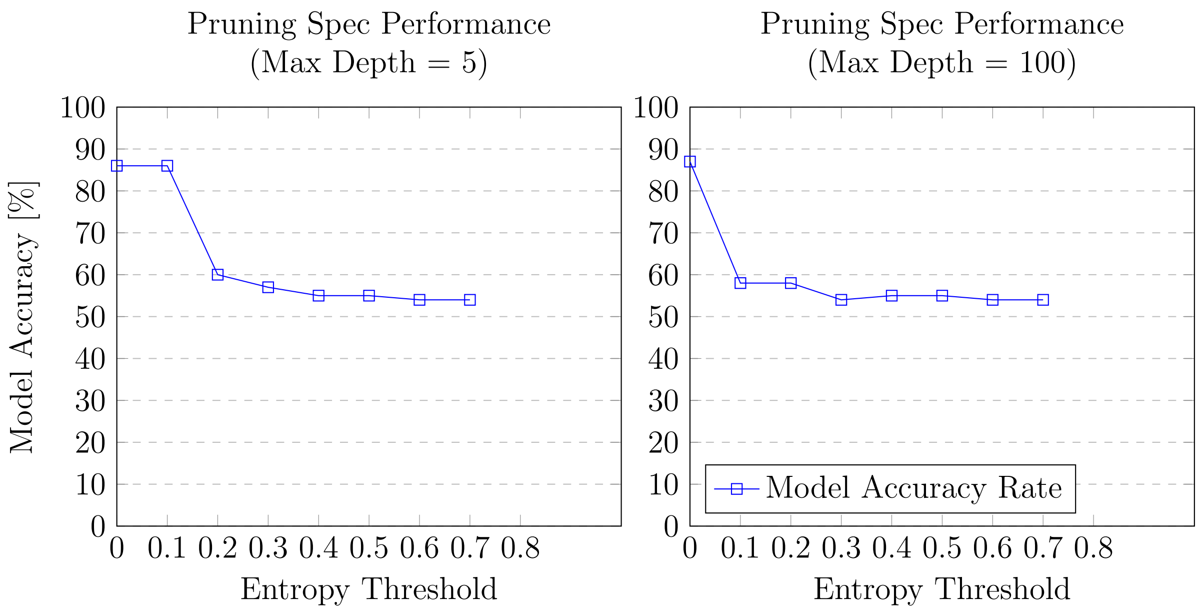


Figure 1: Learning Curves

here. More notable is the fact that using the extreme case where Entropy must equal 0 before considering the tree complete yielded the same results as a 0.1 threshold. However, computation times were slightly longer for this case, as more nodes had to be constructed until the threshold was met. While in our dataset, it wasn't a huge factor, this is an important consideration one should make when building models on much larger data. When the efficacy/accuracy of a model isn't compromised, any measures to speed up performance is a welcomed consideration.

You can also see that increasing the depth of the tree produced slightly better results **only** when we didn't set a threshold. However, the computational time increased dramatically. Moreover, setting entropy thresholds while allowing the tree to grow up to 100 levels deep caused this model to perform very poorly.

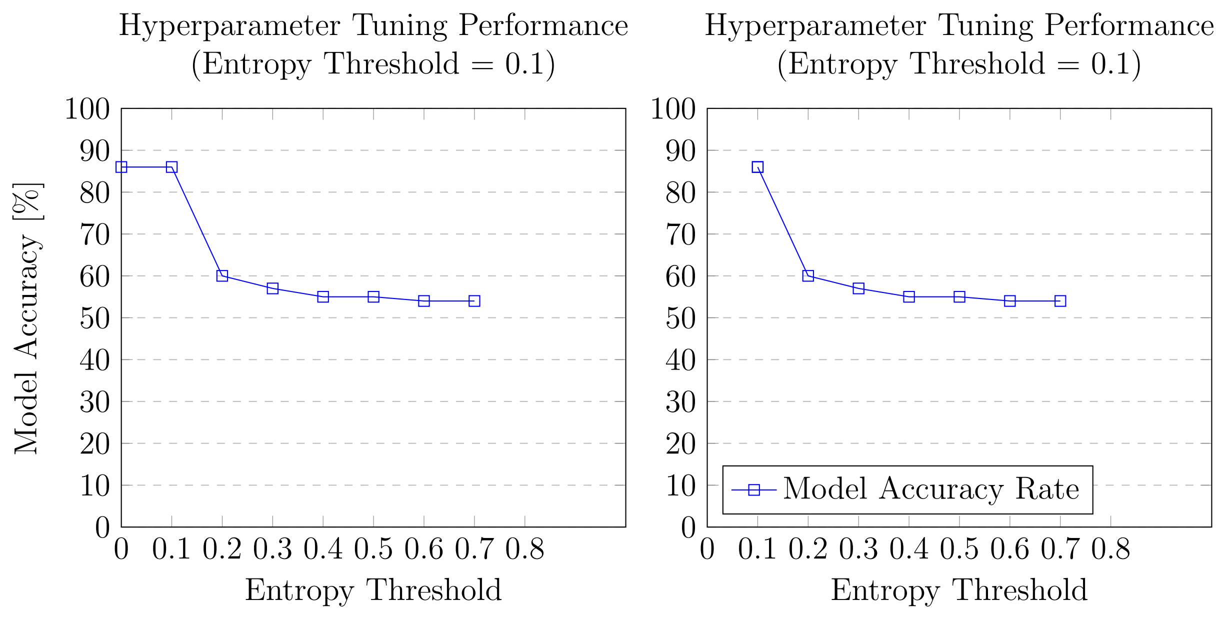


Figure 2: Learning Curves

2.2 Model Complexity Analysis

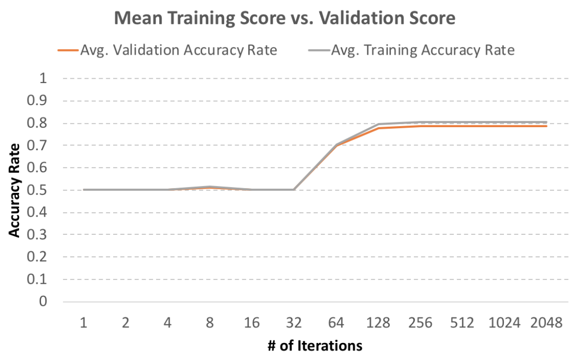
Besides varying the Entropy Threshold & Max Depth, we can also vary the **forest size** hyperparameter in our random forest. Note here that we vary the hyperparameter, while holding the Entropy Threshold to 0.1 as supported by Figure 2.

3 Artificial Neural Networks

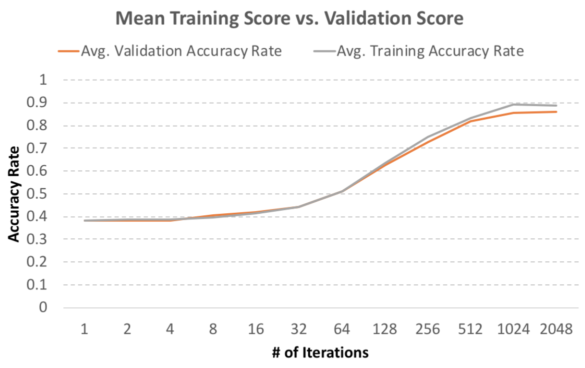
3.1 Learning Curves Analysis

We can further optimize our analysis by letting other parameters in our Neural Network to vary. For one, we can tune our parameters through increasing both the depth and size of our hidden layers. Looking at figure 3, you can see that our ANN model with 3 hidden layers, each of size 30 performs slightly better at lower iterations, but worse at higher number of iterations, as compared to specifying 1 hidden layer of size 1 node. The overall conclusion here is that the simple model with 1 hidden layer using 1 node actually performs better than one with more hidden layers at high number of iterations.

The first figure in 3 produces very promising results. With 30 nodes in each of the 3 Hidden Layers, we can only achieve 80% fit in the training set, and 80% in the test set no matter the number of weight-adjustment iterations (backpropogation). The performance is quite poor all around. The right graph uses the *basicResults.best_params_* method made available in scikitlearn to choose the best parameter for our ANN. The best parameters are: (1) L2 Penalty: 0.00316 (2) Hidden Layer Depth: 1 (3) Hidden Layer Size: 1 (4) Activation Function: RELU. Note that the figure on the right suggests we can reach a generalizable

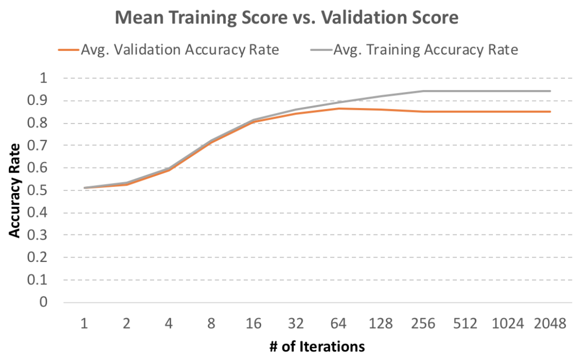


(a) 3 Hidden Layers, 30 Nodes Each

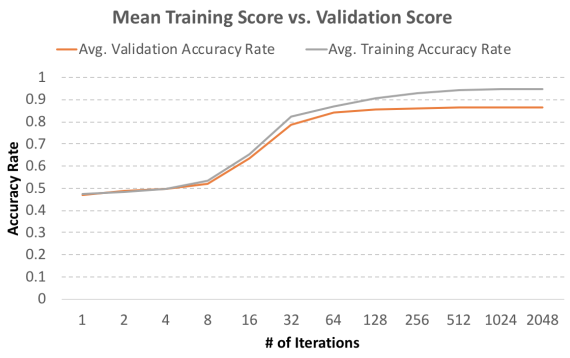


(b) 1 Hidden Layer, 1 Node

Figure 3: Complexity Curves: Testing vs. Validation Sets w/ Varying Depth and Number of Hidden Layer Nodes



(a) L2 Regularization = 0



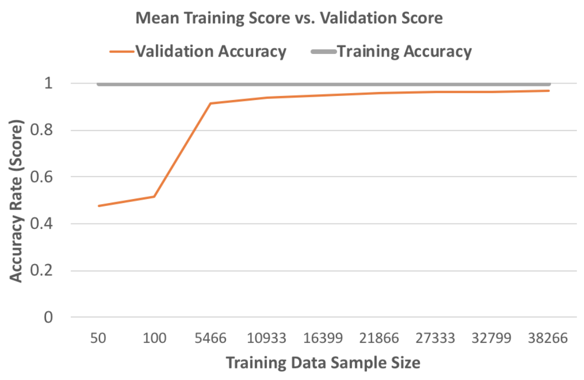
(b) L2 Regularization = 10

Figure 4: Learning Curves: Training vs. Validation Sets w/ Varying Alpha (L2-Norm)

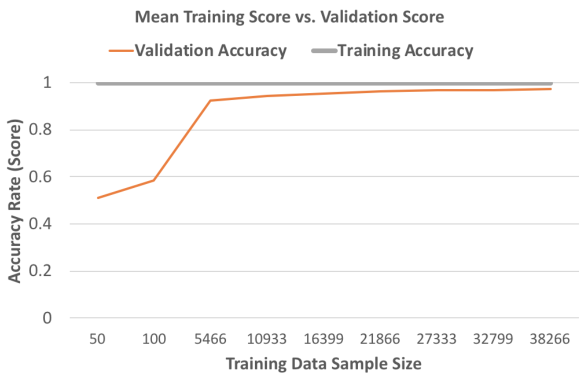
and accurate (88% test accuracy rate with 90% training accuracy rate) model with anywhere between 1,024 to 2048 weight-adjustment iterations. Figure 3(b) also highlights the importance of back-propogation. Giving our model time to make iterative adjustments to the weights drastically improves the model.

3.2 Model Complexity Analysis

To get a basic understanding of how our model performed, we first start off with a "basic" neural net with the following specifications: (1) 1 Hidden Layer of size 30. (2) Activation Function = 'relu' (rectified linear unit function, returns f(x) = max(0,x)) (3) L2 Regularization Alpha = 0. The figure in Figure 4 shows the results with the aforementioned specifications, but letting our L2 Regularization/Penalty term to vary between 0 & 10. Recall that the L2 Norm is essentially a squared error specification, and by it's nature, penalizes weights in a manner to reduce the likelihood of fitting the noise of the training data.



(a) k = 10



(b) k = 1

Figure 5: Learning Curves: Training vs. Validation Accuracy w/ Varying Training Set Size

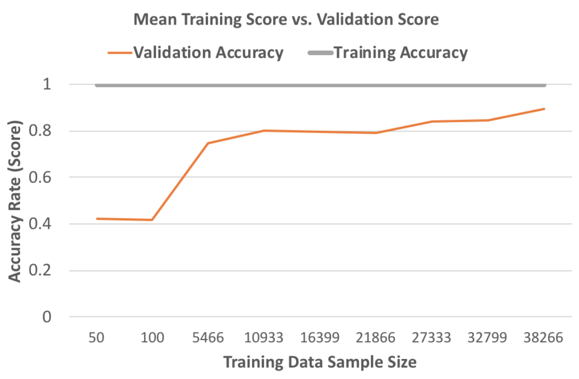
4 k-Nearest Neighbors

Classifying hand gestures based on relative position across 3-dimensional coordinate plane is a perfect candidate for kNN. By measuring the coordinates of markers attached to each finger as a hand gesture (fist, point, grab ...etc.), one can reasonably identify general characteristics related to finger positioning associated with gestures across different users (or, hands).

4.1 Learning Curves Analysis

To get a sense of how effectively our kNN model learns across different sizes of our training set, we can compare the performance of our model as we increase the training set. We will compare the outcomes of our model under 2 scenarios, where each scenario uses a different value for our hyper-parameter k. We first start off with a "basic" implementation of kNN with the following specifications: (1) Default k of 10. (2) Distance = Manhattan (3) Weighting distance of training data uniformly. We then compare our results with the "best" implementation, as chosen by *GridSearchCV.best_params_*. It turns out, $k = 1$, using the Manhattan distance weighted uniformly performs the best. The figure in Figure 5 shows the results with the aforementioned specifications, but letting our k vary from 10 to the optimum $k = 1$ outputted via *GridSearchCV.best_params_*. As you can see, setting $k = 1$ generated perfect training sets (100% training accuracy across all sample sizes). In other words, our 1-NN perfectly classified our training data set. This is largely expected, as 1-NN has low bias, and high variance, and generally produces results suggesting overfitting.

As you can see in figure 5, our validation accuracy experiences a drastic improvement in accuracy as our training set size reaches 6000. Furthermore, we see overfitting occurring at both levels of $k = 1$ and $k = 10$, evidenced by the near perfect training scores. However, the average validation accuracy, which was generated using 5-fold cross validation, suggests our results to be very robust for both values of k, as evidenced in the very similar results. Lastly, the steep jump in performance when reserving only 10% of our data suggests we can learn



(a) kNN w/ Manhattan, Weighted Distance

Figure 6: Complexity Curves: Training vs. Validation Accuracy w/ Varying k

the underlying behavior of our data with relatively little data. It is worth noting here that using 70% of our training data generated a 98% accuracy rate for $k = 1$, and 96% accuracy for $k = 10$ - a small difference between our two ks.

4.2 Model Complexity Analysis

We can add complexity by altering the distance measure (between Euclidean & Manhattan) as well as adjusting our treatment of distance weights of training data. By "discounting" training data points that are farther away from the query data point, we are effectively prioritizing the classification of the query point based on proximity, rather than prioritizing solely based on aggregate average classification of **all** training points (which is the case in the "uniform" distance weighting scenario).

Using the figure 5(a) as a reference (which uses Uniform Distancing), you can see a subtle, yet important change in the accuracy of our validation set when using Weighted Distance as a hyper-parameter. Namely, figure 6(a) shows decreased performance for almost every training set sample size, reaching a maximum validation accuracy rate of 88%. This suggests that Uniform distancing provides better performance, as opposed to discounting data points further away from a test point's nearest neighbor. Logically, this makes sense- a single instance of a hand gesture is subject to measurement bias. By discounting all the other finger marker measurements, which also was subjected to measurement bias, we are not letting the model take this type of error into account. Instead, the distance weighted model is simply placing a higher emphasis on the closest points to make an assessment of hand posture- a misplaced emphasis which most likely took outlier data points from other hand gesture readings, and misclassified our query point as such.