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. The 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 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.

1.1 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.

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

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

[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.

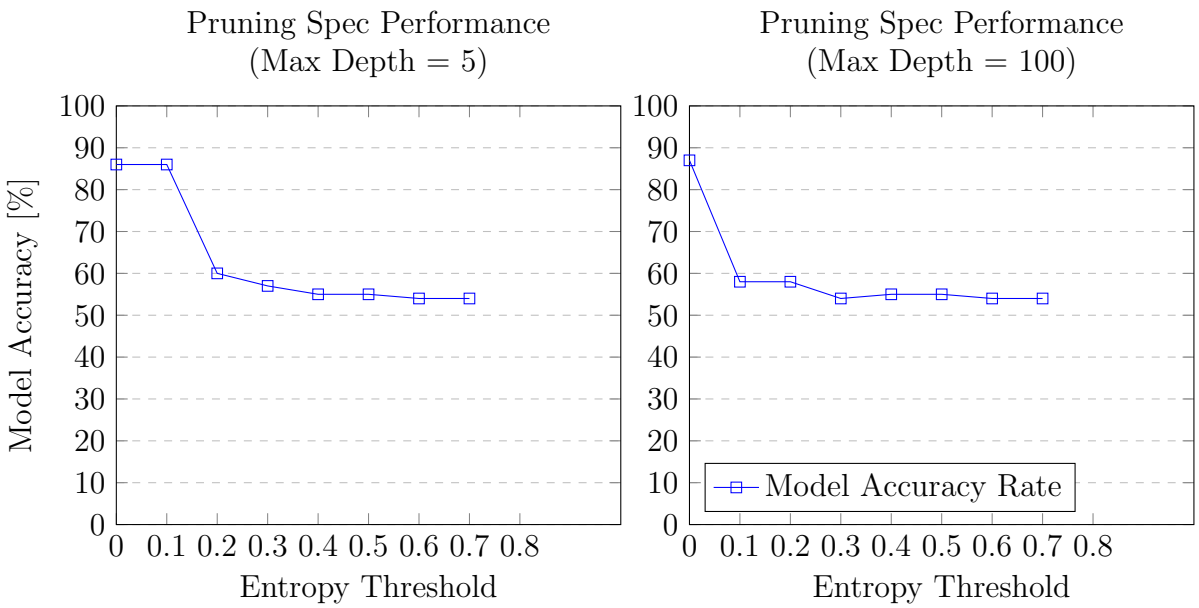


Figure 1: Learning Curves

no better than a coin flip. Thus, setting a pre-pruning specification is quite important 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.

1.2 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.

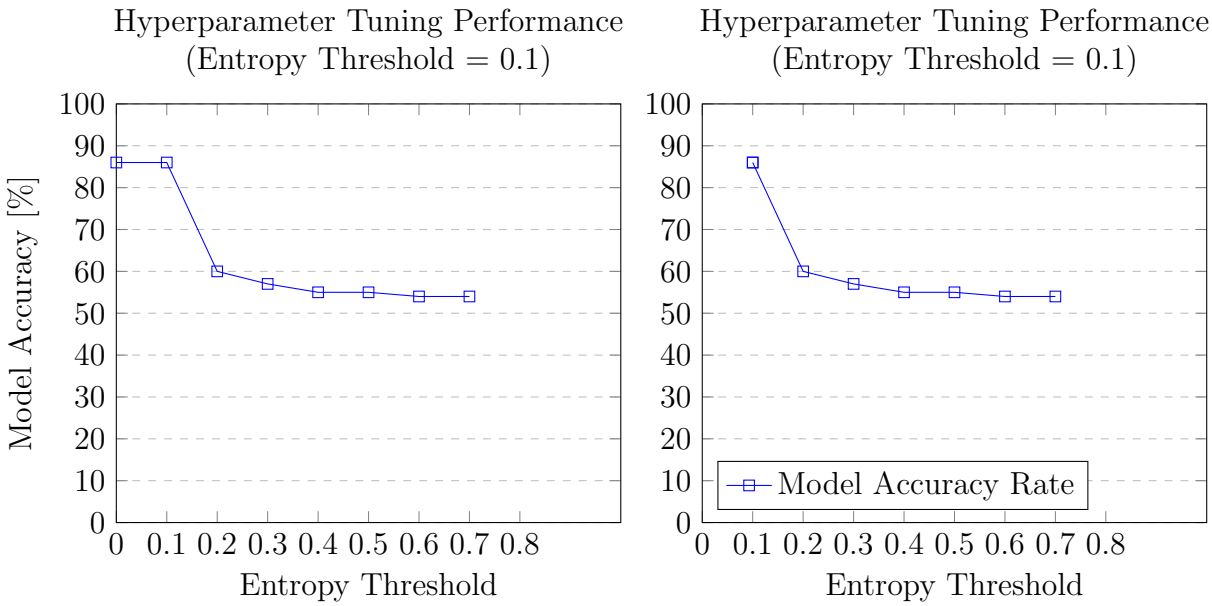


Figure 2: Learning Curves

2 Artificial Neural Networks

2.1 Learning Curves Analysis

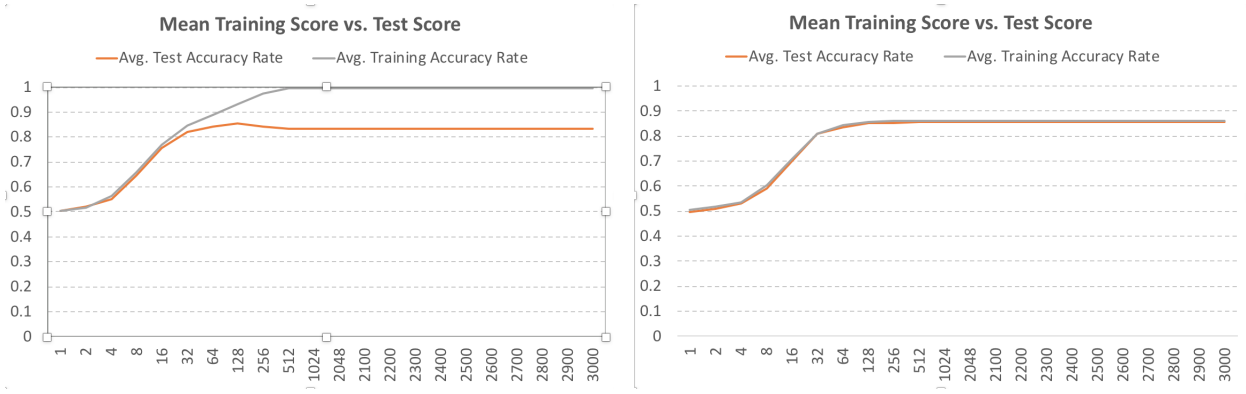
To get a basic understanding of how our model performed, we first start off with a "basic" neural net with the following specifications: (1) 1 Hidden Layer of size 30. (2) Activation Function = 'relu' (rectified linear unit function, returns f(x) = max(0,x)) (3) L2 Regularization Alpha = 0. The figure in Figure 3 shows the results with the aforementioned specifications, but letting our L2 Regularization/Penalty term to vary from 0 & 10.

As you can see, specifying a larger L2 Penalty term reduces the accuracy of our both our Training and Test data. While L2=0 produces a training set that is close to 100% accuracy, the L2=10 rendition produces a training that is 86% at it's best.

2.2 Model Complexity Analysis

We can further optimize our analysis by letting other parameters in our Neural Network to vary. For one, we can add complexity by increasing both the depth and size of our hidden layers.

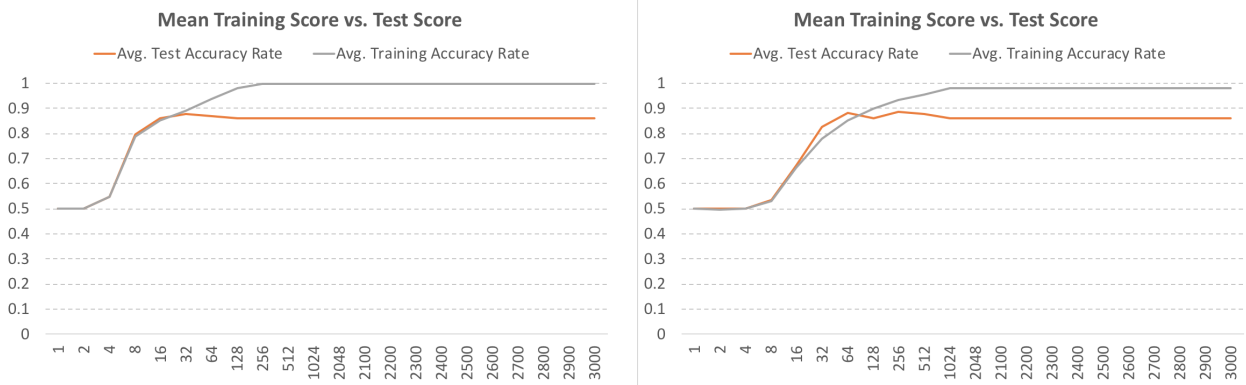
The first figure in 4 produces very promising results. With 30 nodes in each of the 3 Hidden Layers, we achieve 100% fit in the training set, and 86% in the test set with only 32 weight-adjustment iterations (backpropogation). However, one should be weary of these results, as it's highly suggestive of overfitting. The speed in which the training accuracy improves is highly suggestive of this. The right graph uses the *basicResults.best_params_* method made available in scikitlearn to choose the best parameter for our ANN. The best



(a) L2 Regularization = 0

(b) L2 Regularization = 10

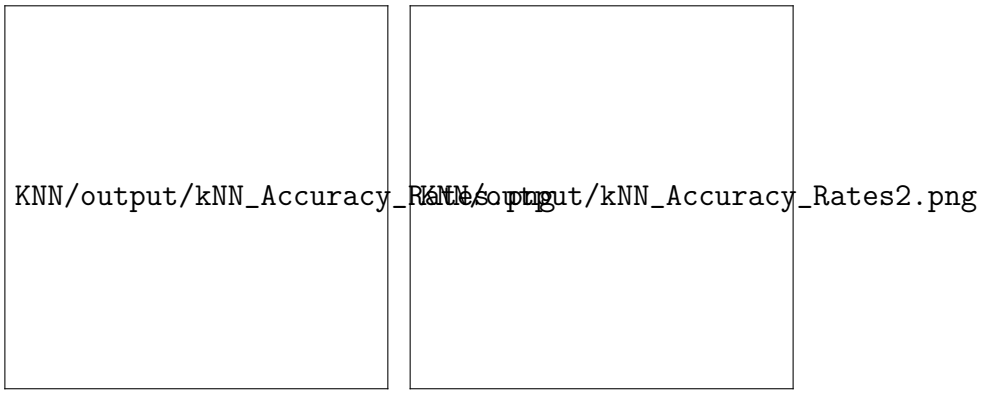
Figure 3: Learning Curves: Testing vs. Training Sets w/ Varying Alpha (L2-Norm)



(a) 3 Hidden Layers, 30 Nodes Each

(b) 1 Hidden Layer, 28 Nodes

Figure 4: Complexity Curves: Testing vs. Training Sets w/ Varying Depth and Number of Hidden Layer Nodes



(a) k = 5 (b) L2 Regularization = 10

Figure 5: Learning Curves: Testing vs. Training Sets w/ Varying k

parameters are: (1) L2 Penalty: 0.00316 (2) Hidden Layer Depth: 1 (3) Hidden Layer Size: 28 (4) Activation Function: Logistic. Note that the figure on the right suggests we can reach a generalizable and accurate (88% test accuracy rate with 99% training accuracy rate) model with anywhere between 512-1024 weight-adjustment iterations.

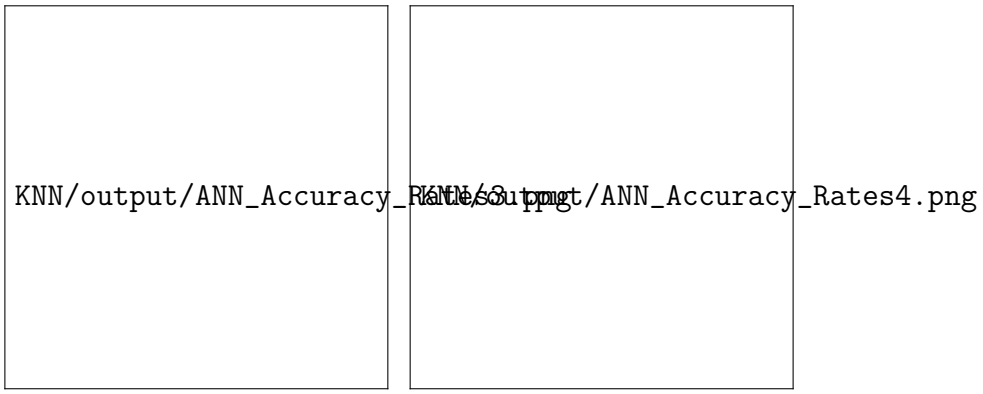
3 k-Nearest Neighbors

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).

3.1 Learning Curves Analysis

To get a basic understanding of how our model performed, we first start off with a "basic" implementation of kNN with the following specifications: (1) Default k of 5. (2) Distance = Euclidian (3) Weighting distance of training data uniformly. The figure in Figure 3 shows the results with the aforementioned specifications, but letting our k vary from 5 to the optimum k outputted via $GridSearchCV.best\_params\_$.

As you can see, specifying a larger L2 Penalty term reduces the accuracy of our both our Training and Test data. While L2=0 produces a training set that is close to 100% accuracy, the L2=10 rendition produces a training that is 86% at it's best.



(a) kNN w/ Manhattan, Uniform Distance

(b) kNN w/ Manhattan, Weighted Distance

Figure 6: Complexity Curves: Testing vs. Training Sets w/ Varying Depth and Number of Hidden Layer Nodes

3.2 Model Complexity Analysis

We can further optimize our analysis by letting hyper-parameters in our kNN model to vary. For one, 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.

The first figure in **??** produces very promising results. With 30 nodes in each of the 3 Hidden Layers, we achieve 100% fit in the training set, and 86% in the test set with only 32 weight-adjustment iterations (backpropogation). However, one should be weary of these results, as it's highly suggestive of overfitting. The speed in which the training accuracy improves is highly suggestive of this. 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: 28 (4) Activation Function: Logistic. Note that the figure on the right suggests we can reach a generalizable and accurate (88% test accuracy rate with 99% training accuracy rate) model with anywhere between 512-1024 weight-adjustment iterations.