J McLeod Assignment 4

October 14, 2019

Jericho McLeodCSI-777Assignment 4Chapter 6 Review

Chapter 6 Review: This is the material that I find exciting. The first sentence of the chapter (decision trees… fast and produce intelligible results … often surprisingly accurate) is something that I have repeated often. Pruning is not a new concept to me, but I was taught originally that it was simply snipping a tree and adding the information gain that was trimmed to the error rate (conceptually if not mathematically). Raising a subtree is an interesting concept, but I do not see how this would result in a better model given that models are created on information gain. I think I will need to implement this myself to get a better understanding of how impactful it is, because even in looking it up I found limited outside resources and explanations.

The section on the complexity of trees is also new to me. As someone coming from a background outside of the sciences, O notation is a newer concept to me, and has taken some time for me to become comfortable with. That said, one of the things my manual implementation of decisions handled poorly was the sorting of attributes. The textbook points out that sorted sets can be stored for subsequent use, while I was destroying them. However, it also claims sorting is O(n log n), which is generally an effective sort time, but Python uses Radix sort, which is O(n), or actual time n*k/d. So, while I made some errors that slowed down my implementation, I also saved some time in other places.

I also found the discussion section the history of decision trees interesting. It was nice to see that all the various techniques did not come from a single super-genius, but from a collected body of work over time. More specifically, though, the statement regarding the ease with which people understood complex systems based on phrasing. In my mind, there is no difference between having a set of rules that end with classifications. However, it was shown in an experiment that in at least the context studied, exceptions are a better way to communicate rules because they better align with the perspective of people familiar with the subject matter. In other words, medical workers think of cases in terms of 'if symptom A, diagnosis X, unless symptom/test B, in which case diagnosis Y.' It is noteworthy that matching a set of rules to the existing method of making classifications has value in sharing knowledge, especially as someone who intends to work in a field where this has value.

Implementing C4.5 First, I need to import libraries.

```
In [1]: from sklearn import tree
        import csv,random
        import matplotlib.pyplot as plt
```

Then, the dataset must be imported. I am using the traditional dataset rather than the dataset with truncated features.

```
In [2]: data = []
        with open('iris.csv') as csvfile:
            csv_r = csv.reader(csvfile,delimiter=',')
            for row in csv_r:
                data.append(row)
```

Next I need to create datastructures for the features vs. the classifications.

```
In [3]: x = []
        labels = data[0]
        y = []
        for i in range(len(data)):
            if i > 0:
                x.append(data[i][:4])
                y.append(data[i][4])

        print(data[0])
        print(data[1])
        print(data[2])
```

```
['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']
['5.1', '3.5', '1.4', '0.2', 'setosa']
['4.9', '3', '1.4', '0.2', 'setosa']
```

The next two lines of code show how ridiculously easy it is to implement a C4.5 tree in python. Technically, this is a CART tree that can handle numerical values without any additional coding, but otherwise the alogirthm is C4.5 per scikit-learn documentation available here: https://scikit-learn.org/stable/modules/tree.html

```
In [4]: clf = tree.DecisionTreeClassifier()
        clf = clf.fit(x,y) # each sample row is together
```

Verifying that I did not introduce any faults thus far by classifying observations from the training data:

```
In [5]: print(clf.predict([x[0]])[0],y[0])
        print(clf.predict([x[50]])[0],y[50])
        print(clf.predict([x[100]])[0],y[100])
```

```
setosa setosa
versicolor versicolor
virginica virginica
```

Great; the decision tree can classify observations within the training data. This doesn't tell us anything about its efficacy at classifying new instances, but it does tell us that the implementation functions as expected; it is not broken.

Below is a manual implentation of K-fold validation. It is a flawed implementation, but it is close enough to not matter much for the purposes of exploring the system in a classroom setting. It outputs an array of the accuracies across the K tests.

The flaw is that it uses random numbers between 0 and 1, and then sorts out by slicing that space by K. A better implementation would be to use random numbers between 0 and K, and as classes fill up, reduce K and distribute among the remaining classes.

```
In [6]: k = 5 # This will give us ~30 observations in the validation set

        def k_validation(k,x,y):
            rand_array = []
            for i in range(len(x)):
                rand_array.append(random.random())
                # this function will virtually never return exact values that match anything;

            if len(rand_array) != len(x):
                print("Error in random array generation")

            accuracy = []
            for n in range(k):
                x_train,x_test,y_train,y_test = [],[],[],[]
                for i in range(len(x)):
                    if rand_array[i] > (1/k)*n and rand_array[i] < (1/k)*(n+1):
                        x_test.append(x[i])
                        y_test.append(y[i])
                    else:
                        x_train.append(x[i])
                        y_train.append(y[i])
                clf = tree.DecisionTreeClassifier()
                clf = clf.fit(x_train,y_train)
                correct = 0
                for i in range(len(x_test)):
                    result = clf.predict([x_test[i]])
                    if result[0] == y_test[i]:
                        correct+=1
                try:
                    accuracy.append(correct/len(y_test))
                except:
                    pass
            return(accuracy)

        accuracy = k_validation(k,x,y)

        print(accuracy)
```

```
[0.9722222222222222, 0.967741935483871, 0.9354838709677419, 0.9583333333333334, 0.928571428571
```

It looks like using 5-fold cross validation gives us results between 88% and 96% accurate across

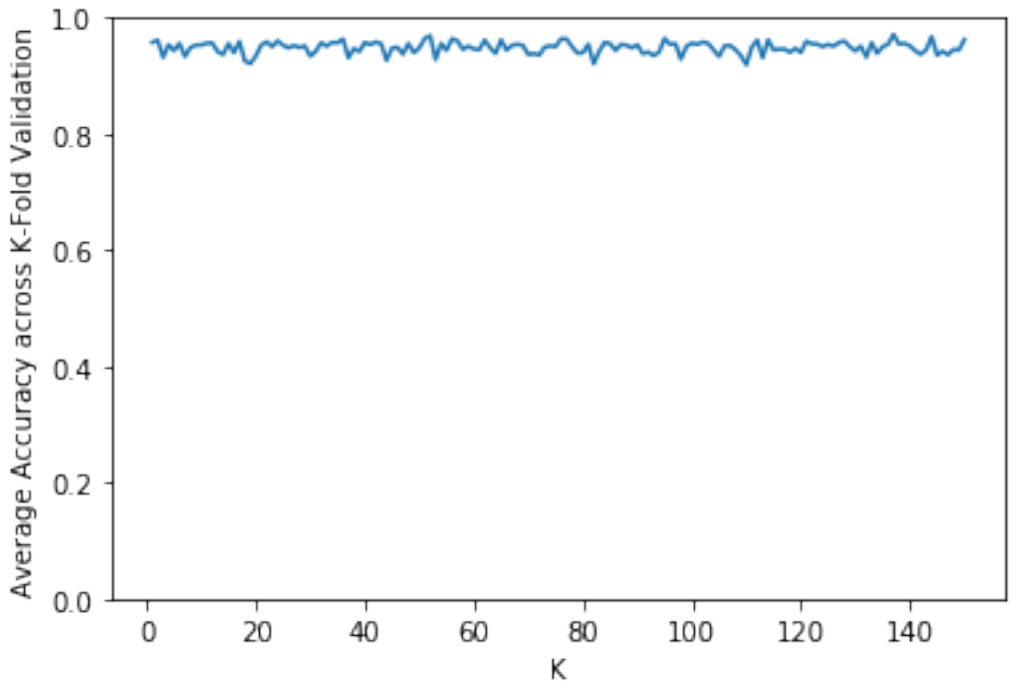
one possible run. Let's look at running from K=2 to K = length of the dataset to see how it compares.

```
In [7]: indexes = []
        accuracies = []
        for i in range(len(x)):
            indexes.append(i+1)
            accuracy = k_validation(i+2,x,y)
            total = 0
            for i in accuracy:
                total += i
            total = total/len(accuracy)
            accuracies.append(total)
```

And plotting the outcomes:

```
In [8]: plt.plot(indexes,accuracies)
        plt.ylim(0,1)
        plt.xlabel('K')
        plt.ylabel('Average Accuracy across K-Fold Validation')
```

```
Out[8]: Text(0, 0.5, 'Average Accuracy across K-Fold Validation')
```

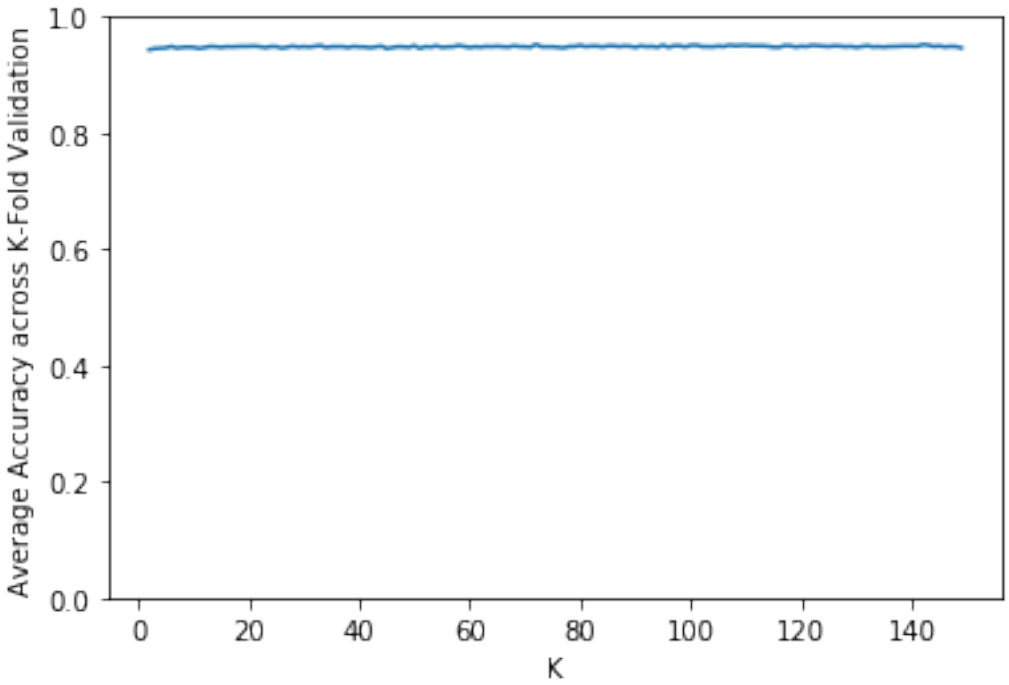


I am not convinced that K matters much. This seems like a good time to run some monte carlo simulations and get smoother outputs.

```
In [9]: accuracies = []
        for q in range(len(x)-2):
            indexes.append(q+1)
            high_acc =[]
            for j in range(100):
                accuracy = k_validation(q+2,x,y)
                total = 0
                for k in accuracy:
                    total += k
                total = total/len(accuracy)
                high_acc.append(total)
            total = 0
            for j in high_acc:
                total+=j
            total = total/len(high_acc)
            accuracies.append(total)
```

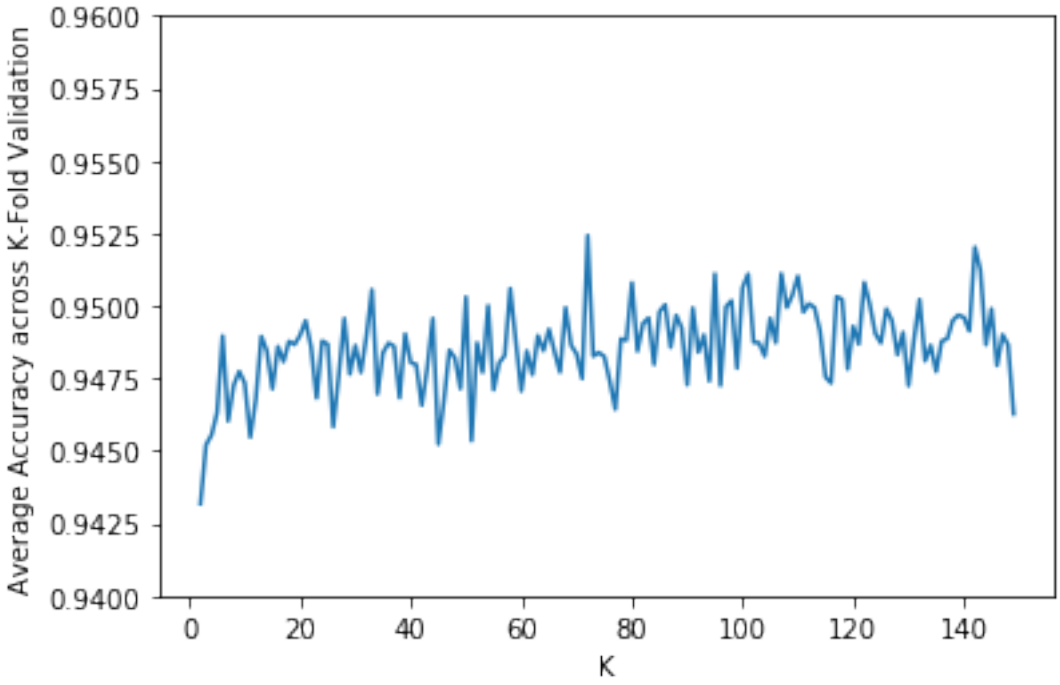
```
In [10]: indexes = []
         for i in range(len(accuracies)):
             indexes.append(i+2)

         plt.plot(indexes,accuracies)
         plt.ylim(0,1)
         plt.xlabel('K')
         plt.ylabel('Average Accuracy across K-Fold Validation')
         plt.show()
```



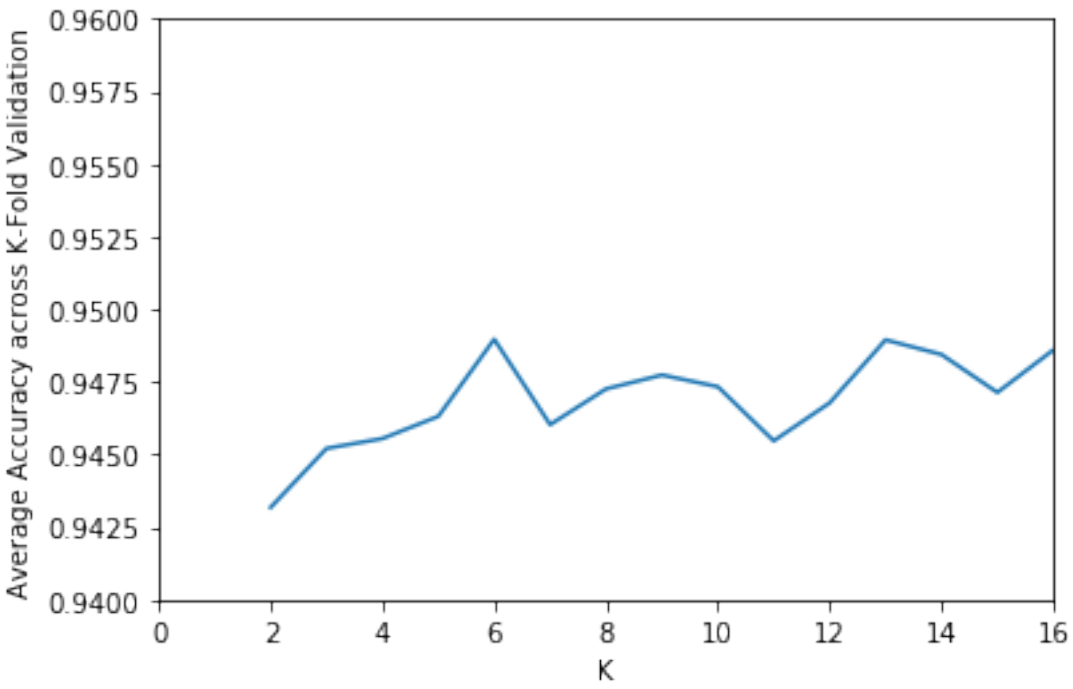
This is basically a smooth line. That tells me the results are similar, but not much else. The next step is to zoom relative to the Y axis.

```
In [11]: plt.plot(indexes,accuracies)
         plt.ylim(0.94,0.96)
         plt.xlabel('K')
         plt.ylabel('Average Accuracy across K-Fold Validation')
         plt.show()
```



This chart shows 2% of the Y-axis of the prior visualization. This amplifies the noise from test to test, but also shows the smallest values of K are outliers. The next step is to zero in on this range on the X axis.

```
In [12]: plt.plot(indexes,accuracies)
         plt.ylim(0.94,0.96)
         plt.xlim(0,16)
         plt.xlabel('K')
         plt.ylabel('Average Accuracy across K-Fold Validation')
         plt.show()
```



This shows only the first 10% of the tests to further identify which K values were insufficient. It looks like K<4 in this case.

Updating my prior opinion: K matters a little. When it is too small you may lose some fidelity in your models. When it is too high (as in, k>instances), there will be empty test sets. With Iris data, at least, K can be virtually anything from 4 to the length of the dataset.

Computationally, K is the number of times a model must be trained and evaluated. For an expensive model, minimizing K would decrease validation times while, apparently, not sacrificing much in terms of the quality of the validation. For less expensive models other methods, like 'leave one out' (K = n) work fine, but I see no readily apparent advantage to that method from this particular example.

As a footnote, I ran this several times. Typically, there isn't a valley on the right-hand side of the middle chart, but the one on the left is very consistent. However, I am currently also training an artificial neural network, and it is hogging systems resources to the point that I did not want to re-run this test again. Feel free to re-run the jupyter notebook to check this out yourself, but as a warning; don't train an ANN at the same time!