Data Mining: Decision Trees (C4.5) and Naïve Bayes

Table of Contents

[Introduction 3](#_Toc45587240)

[Description 3](#_Toc45587241)

[Propose 4](#_Toc45587242)

[Assumption 6](#_Toc45587243)

[Requirement 6](#_Toc45587244)

[Steps to Run 6](#_Toc45587245)

[Source Code 7](#_Toc45587246)

[Output 17](#_Toc45587247)

[Conclusion 18](#_Toc45587248)

# Introduction

This paper will discuss Decision Tree (C4.5) and Naïve Bayes algorithms. These are supervised data mining classification (Option 1) and it will be using a classification dataset. We will implement both algorithms from scratch and execute our dataset. We will run k-fold cross validation and compare our result and identify which algorithm is better. The source code is written in Python using Jupyter notebook.

# Description

The Decision Trees are supervised learning algorithms used for both, classification and regression tasks. The main idea of decision trees is to find those descriptive features which contains the most information regarding to the target feature and then resulting split sub dataset. It takes the training set as input and builds a tree that resembles to an orientation diagram where each end node (leaf) is a decision (a class).

In this project, we will be implementing the **C4.5** algorithm. It is an improvement on the common algorithm called ID3. ID3 will calculate the entropy and retrieve the highest information gain of each attribute in our dataset. The formulas are shown below.

*Entropy(class) = ∑ – p(I) . log2p(I)*

*Gain(class|attribute) = Entropy(class) - ∑ – p(I) . log2p(I)*

The C4.5 algorithm will use the maximum gain ratio among our attributes. We will need to calculate the split info to retrieve the gain ratio. This can be done with the equation below.

*SplitInfo(attribute) = -∑ | attribute j|/| attribute | x log2| attribute j|/|attribute|*

*GainRatio(class|attribute) = Gain(class|attribute) / SplitInfo(attribute)*

The Naïve Bayes is simple classifier and power algorithms which is used for binary and multi-class classification problems. It is based on Bayes Theorem, as shown below, to find the probability of class happening with given data sets. The higher probability will be selected as the label on that data sets. The *P(class|data)* is the posterior probability of class given data, P(class) is the class prior probability and P(data) is the prior probability or predicator known as evidence.

*P(data|class) \* P(class)*

*P(class|data) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*P(data)*

To start, we would convert the data set into a frequency table and create a likelihood table on attributes on each class. We will then use the Bayes Theorem to calculate the posterior probability for each class. The class with the highest probability is the result of the prediction.

The k-fold Cross-validation will be used to evaluate each our algorithm. This technique will divide the set into k sub-samples. In each run, use one distinct sub-sample as testing set and the remaining k-1 sub-samples as training set. Evaluate the algorithm using the average of all the k runs. This method will reduce the randomness of training set/testing set.

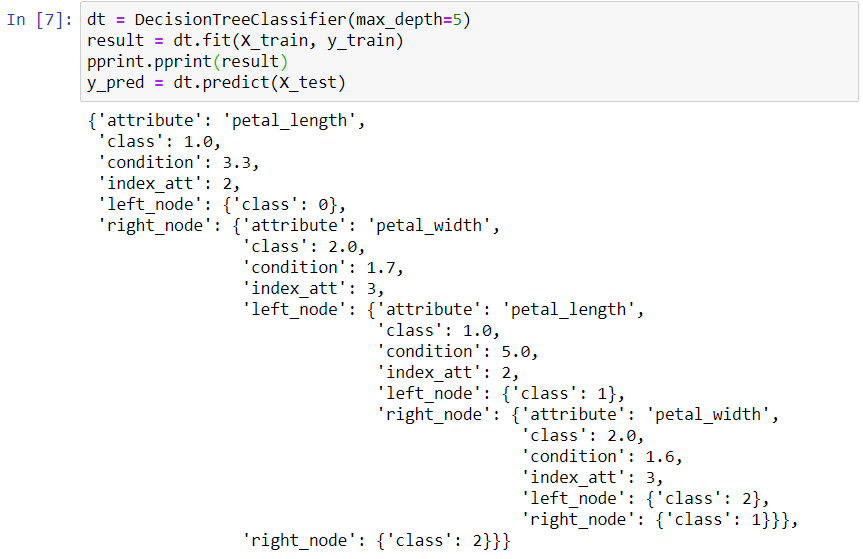
# Propose

For this exercise, we will be using the IRIS dataset from the link below. We will read our dataset with the help of Pandas library and split into training and testing set. We will be splitting based on the percent towards the testing set, but we will be using 30% on our testing set and the rest will be training set. The splitting will be randomized for to eliminate bias.

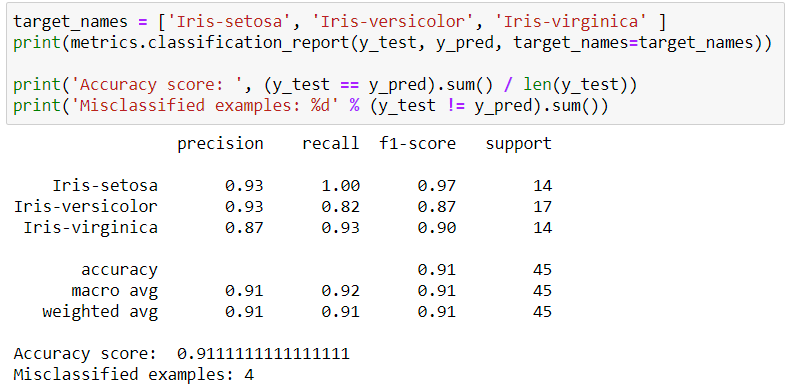
<https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data>

We will implement Decision Tree Classification first and train our model with our training set. We will find which attribute will be use as our root node. We will iterate through each attribute to find the best split between each attribute. We will select the best attributes by calculating Entropy, Information Gain and Gain Ratio. First, we will calculate and select the smallest entropy and high information gain. We will find the split information and use the information gain to calculate the gain ratio. We will select the largest gain ratio with the best splitting method/condition our dataset.

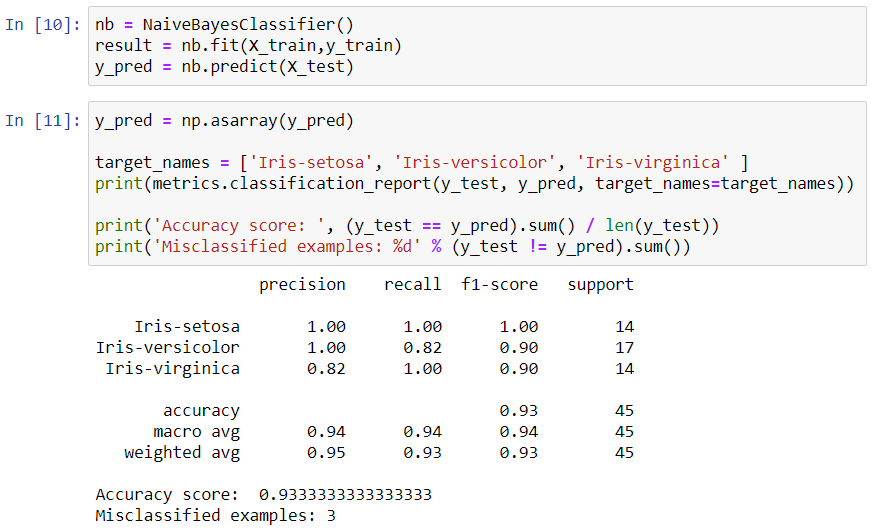
With our training set and the *pprint* library, we build and visualize our tree. In each node, we will have a leaf node with the class/label decision. It also contained the attribute used and condition where the data is split. We will use this tree, to predict the labels on each testing set.



We will use the model tree to predict the labels on each testing set. In predict function, we will traverse down the tree and meeting the conditions, it will output the predicted the labels. In our model, the testing set has the accuracy score of 91 % and 4 misclassified dataset.



Next, we will implement Naïve Bayes Classification and execute our training phase with our training set. We will split our classes and loop on each attribute to calculate the mean and standard deviation known as evidence P(X). We will use this information to predict our testing set. This algorithm will be using Bayes theorem. We will calculate the posterior probability P(Y|X) and find the highest probability on each testing set to identify the label. We ran our testing set and resulted to 93% accuracy score with 3 misclassified datasets.



Finally, we will create and run our 10-fold cross validation method for both algorithms to measure the classification accuracy. In this module, we split our data randomly into 10 partition/group and select one for testing set and the rest are training set. We will iterate this process on each group and collect our accuracy score. As the end, we will take our average accuracy score to measure the algorithm. We will compare both of our algorithms using graphs.

# Assumption

We will be using Iris classification dataset in Decision Tree and Naïve Bayes classifier. We will be pulling the dataset using URL above and start predicting a discrete class label output.

# Requirement

* Jupyter Notebbok
  + Install Anaconda – This is used to open Jupyter Notebook. It is a Python distribution that simplifies package management. It installs with many data science libraries and Jupyter.
  + pip command is an alternative to install Jupyter. First, make sure the pip is in latest version.

|  |
| --- |
| pip3 install –upgrade pip |

Second, install the Jupyter Notebook using the command below.

|  |
| --- |
| pip3 install jupyter |

* It will need NumPy and Pandas packages.

# Steps to Run

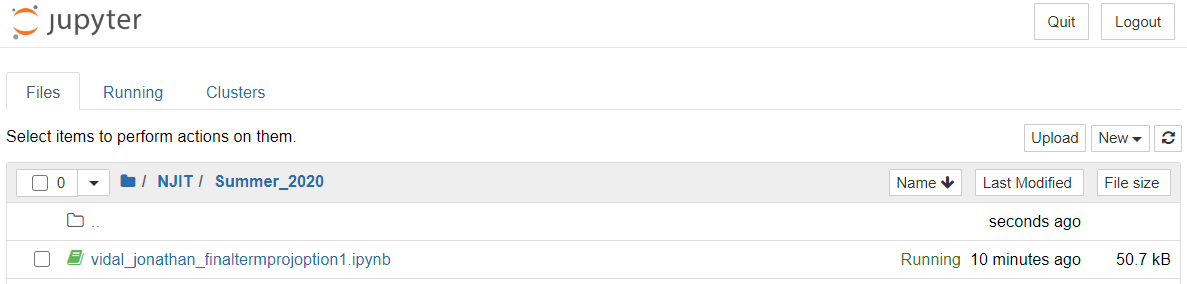
This project will be using Python 3 and Jupyter Notebook.

1. From the command line/terminal, we can start the notebook by running the command below.

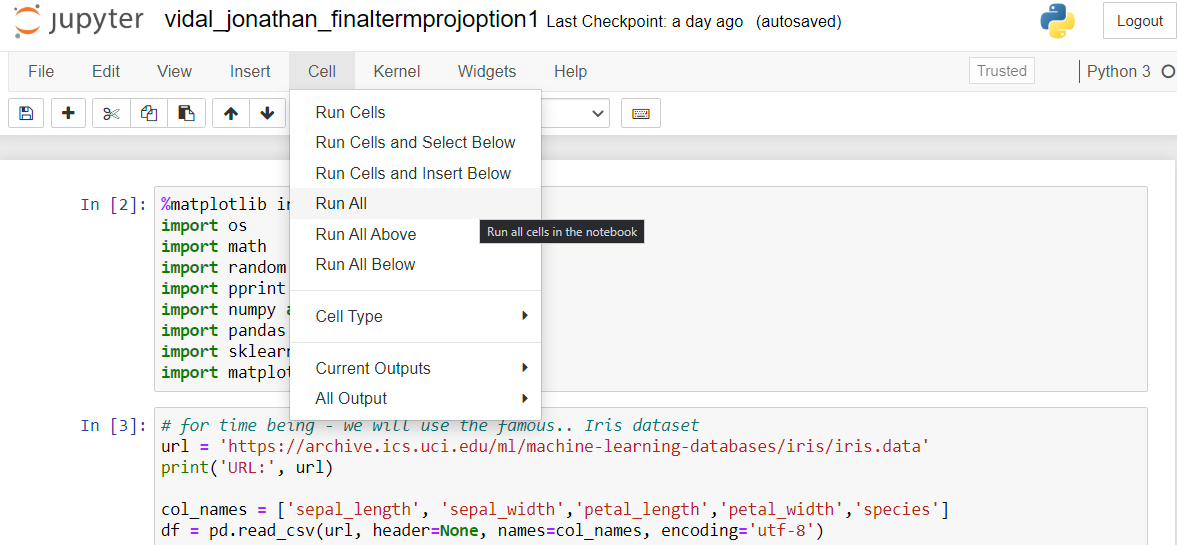
|  |
| --- |
| jupyter notebook |

1. It will open the tab from a web browser and select the python file. Since it is using Jupyter notebook, the file extension will be *ipynb*.

In my machine, the file is located in /NJIT/Summer\_2020/



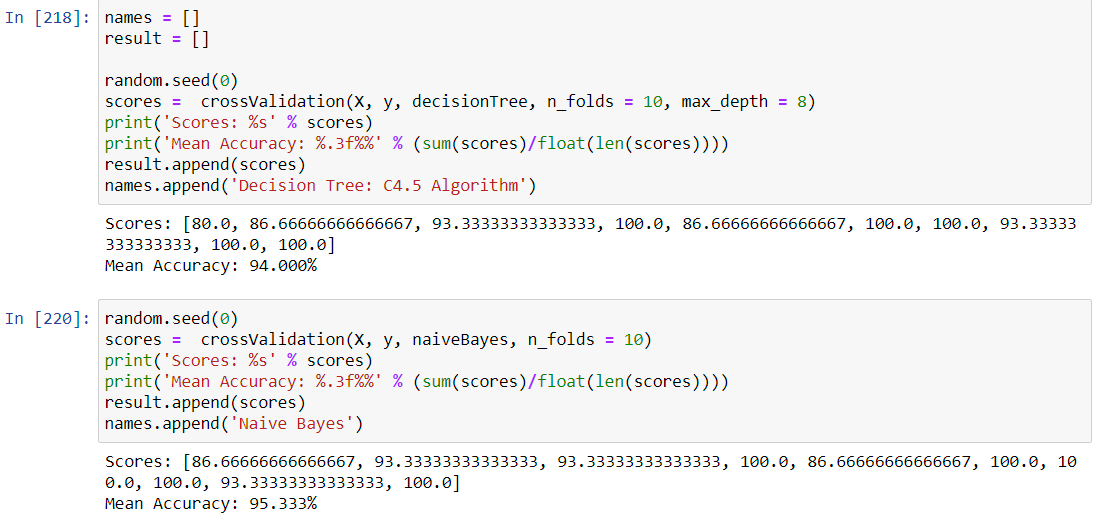
1. Once the file is open, you can click “Cell” and select “Run All”.



# Source Code

|  |
| --- |
| *#!/usr/bin/env python # coding: utf-8  # In[2]:* get\_ipython().run\_line\_magic(**'matplotlib'**, **'inline'**) import os import math import random import pprint import numpy as np import pandas as pd import sklearn.metrics as metrics  import matplotlib.pyplot as plt   *# In[3]:   # for time being - we will use the famous.. Iris dataset* url = **'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'** print(**'URL:'**, url)  col\_names = [**'sepal\_length'**, **'sepal\_width'**,**'petal\_length'**,**'petal\_width'**,**'species'**] df = pd.read\_csv(url, header=None, names=col\_names, encoding=**'utf-8'**)  *# we have 'Iris-setosa', 'Iris-versicolor', and 'Iris-virginica'  # species; we will convert it to numeric as labels* print(**'Species: '**, df[**'species'**].unique())  df.loc[(df[**'species'**] == **'Iris-setosa'**, **'species'**)] = 0 df.loc[(df[**'species'**] == **'Iris-versicolor'**, **'species'**)] = 1 df.loc[(df[**'species'**] == **'Iris-virginica'**, **'species'**)] = 2  df = df.rename(columns={**"species"**: **"label"**}) *#df.head()   # In[4]:   # split our dataset to train and test randomly* def train\_test\_split(X, y, test\_size = 0.3):  if isinstance(test\_size, float):  test\_size = round(test\_size \* len(X))   indices = [\*range(len(X))]  test\_indices = random.sample(population=indices, k=test\_size)    X\_test = X[test\_indices].astype(**'float'**)  y\_test = y[test\_indices].astype(**'int'**)    X\_train = np.delete(X, test\_indices, axis=0).astype(**'float'**)  y\_train = np.delete(y, test\_indices, axis = None).astype(**'int'**)    return X\_train, X\_test, y\_train, y\_test   *# In[5]:* X = df.iloc[:,:4].values y = df.iloc[:,4].values  random.seed(0) *# random seed to reproduce our results* X\_train, X\_test, y\_train, y\_test= train\_test\_split(X, y, test\_size = 0.3)   *# In[6]:* class DecisionTreeClassifier(object):  def \_\_init\_\_(self, max\_depth = 5):  self.depth = 0  self.max\_depth = max\_depth *# define the node levels* self.infogain = dict()    def fit(self, x, y, root\_node={}, depth=0):   *# checking if all our labels is the same otherwise we are done here* all\_same = all(i == y[0] for i in y)    if all\_same:  return {**'class'**:y[0]}  elif len(y) == 0:  return None  elif root\_node is None:   return None  elif depth >= self.max\_depth:  print(**'Fucking Bitch'**)  return None    if len(self.infogain) == 0:  *# compute our information gain* self.infogain = self.getInfoGain(y)    *# split by the lowest entropy* att, condition, entropy, infogain, gainratio = self.splitAttribute(x, y)   *#print(entropy)* root\_node = {**'attribute'**: col\_names[att], **'index\_att'**:att,  **'condition'**:condition,**'class'**: np.round(np.mean(y))} *# for multiclass, take the average   # using recursion, we wil run the fit function on our left and right node..* y\_left = y[x[:, att] < condition]  x\_left = x[x[:, att] < condition]   y\_right = y[x[:, att] >= condition]  x\_right = x[x[:, att] >= condition]   root\_node[**'left\_node'**] = self.fit(x\_left, y\_left, {}, depth+1)  root\_node[**'right\_node'**] = self.fit(x\_right, y\_right, {}, depth+1)   self.depth += 1 *# increase the depth of our tree* self.root\_node = root\_node *# store our tree* return root\_node      def getInfoGain(self, y):  num = len(y)  infogain = dict()  for label in set(y):  positive = sum(y==label) *# true* negative = sum(y!=label) *# false* ent = self.entropyCalc(positive, negative)   infogain[label] = ent  return infogain    def splitAttribute(self, x, y):  att = None   condition = None  infogain = self.infogain  min\_entropy = 1  max\_gain = 0   max\_infogain= 0  for i, c in enumerate(x.T):  *# loop per attribute to find the best feature to split* ent, condition\_test = self.getMinimumEntropy(c, y)  if ent == 0:*# return if entropy is zero* return i, condition\_test, ent, max\_infogain, max\_gain      *# find the minimum\_entropy* if ent <= min\_entropy:  overall\_infogain = []  for label in set(y):  overall\_infogain.append(infogain[label] - ent)  max\_infogain = max(overall\_infogain)  *# retrieve maximum information gain* split\_info = self.getSplitInfo(c, y, i, condition\_test)   gain = self.getGainRatio(max\_infogain, split\_info)  if gain > max\_gain:  att = i  min\_entropy = ent  max\_infogain = max\_infogain  max\_gain = gain  condition = condition\_test      return att, condition, min\_entropy, max\_infogain, max\_gain    def getSplitInfo(self, x, y, att, condition): *#C4.5* if len(y) == 0:  return None    y\_num = len(y)   y\_left = len(y[x < condition])  y\_right = len(y[x >= condition])   if y\_left == 0:  left\_split = 0  else:   left\_split = -(y\_left/y\_num)\* math.log(y\_left/y\_num, 2)  if y\_left == 0:  right\_split = 0  else:   right\_split = -(y\_right/y\_num)\* math.log(y\_right/y\_num, 2)    return left\_split + right\_split      def getGainRatio(self, max\_infogain, split\_info):  if split\_info == 0:   return 0  return max\_infogain/split\_info      def getMinimumEntropy(self, att, y):  n = len(y)  minimum\_entr = 1  *# loop trought unique value and find the lowest entrophy* for value in set(att):  y\_pred = att < value  *# retrieve our entropy on each attributes* entr = self.getEntropy(y\_pred, y)  if entr <= minimum\_entr:  condition = value  minimum\_entr = entr  return minimum\_entr, condition    def getEntropy(self, y\_pred, y):   *# this will get me the entropy on the possible split* num = len(y)  *# calculate the entropy left\_node and right\_node entropy* entrophy\_positive, num\_positive = self.entropyNode(y[y\_pred])  entrophy, num\_negative = self.entropyNode(y[~y\_pred])   *# overall entropy* overall\_entropy = num\_positive/num \* entrophy\_positive + num\_negative/num \* entrophy    return overall\_entropy    def entropyNode(self, labels):   total\_entropy = 0  num = len(labels)  classes = set(labels)  *#compute each entrophy; example from the slides:  # E(P\_Usage) = 6/15 I(1,5) + 6/15 I(5,1) + 3/15 I(3,0)* for label in classes:   positive = sum(labels==label) *# true* negative = sum(labels!=label) *# false* ent = positive/num \* self.entropyCalc(positive, negative)   total\_entropy += ent  return total\_entropy, num    def entropyCalc(self, positive, negative):  *# if one contains all the same classes, entropy will return 0* if positive== 0 or negative == 0:   return 0  num = positive + negative   *# calculate the info gain with entropy* positive\_side = -(positive/num) \* math.log(positive/num, 2)  negative\_side = -(negative/num) \* math.log(negative/num, 2)    return positive\_side + negative\_side    def predict(self, x):  tree = self.root\_node    result = np.array([0]\*len(x))  for i, c in enumerate(x):  result[i] = self.parseTree(c)  return result    def parseTree(self, row):  tree = self.root\_node  *#pprint.pprint(tree)  # simply traverse down the tree on each attribute* while tree.get(**'condition'**, {}):  if row[tree[**'index\_att'**]] < tree[**'condition'**]:  tree = tree[**'left\_node'**]  else:  tree = tree[**'right\_node'**]  else:  return tree.get(**'class'**)   *# In[7]:* dt = DecisionTreeClassifier(max\_depth=5) result = dt.fit(X\_train, y\_train) pprint.pprint(result) y\_pred = dt.predict(X\_test)   *# In[8]:* target\_names = [**'Iris-setosa'**, **'Iris-versicolor'**, **'Iris-virginica'** ] print(metrics.classification\_report(y\_test, y\_pred, target\_names=target\_names))  print(**'Accuracy score: '**, (y\_test == y\_pred).sum() / len(y\_test)) print(**'Misclassified examples: %d'** % (y\_test != y\_pred).sum())   *# In[9]:* class NaiveBayesClassifier(object):  def \_\_init\_\_(self,unique\_classes = None):  self.classes=unique\_classes    def combineData(self, X, y):  *# combmine our attributes and labels* dataset = []  for i in range(len(X)):  dataset.append(X[i].tolist()+[y[i]])  return dataset    def fit(self, X, y):  if len(X)==0 or len(y) == 0:  return None  elif X is None or y is None:  return 0  else:  datasets = self.combineData(X, y)    *# training phase which returns the prior probability on each labels* self.model = self.infoPerClass(datasets)  return self.model    def infoCalc(self, datasets):  info = []  *# calculate the mean, standard deviation and count for each dataset* for col in zip(\*datasets):  info.append([np.mean(col), np.std(col), len(col)])  info.remove(info[-1])  return info   def infoPerClass(self,datasets):  *# collect all dataset on each labels* label\_sets = self.datasetPerClass(datasets)    info = dict() *# info of each class* for label, data in label\_sets.items():  info[label] = self.infoCalc(data)  return info    def datasetPerClass(self, datasets):  *# collect all dataset on each labels* label\_sets = dict() *# used dictionary to return* for i, c in enumerate(datasets):  label = c[-1] *# last column is our labels* if (label not in label\_sets):  label\_sets[label] = list() *# create a list in the dictionary* label\_sets[label].append(c)  return label\_sets   def calcProb(self, x, mean, stdev):  exponent = math.exp(-((x-mean)\*\*2 / (2 \* stdev\*\*2 )))  return (1 / (math.sqrt(2 \* math.pi) \* stdev)) \* exponent   def calcClassProb(self, row):  summaries = self.model    totals = []  for label in summaries:  totals.append(summaries[label][0][2])  total\_rows = sum(totals)    probabilities = dict()  for class\_value, class\_summaries in summaries.items():  probabilities[class\_value] = summaries[class\_value][0][2]/float(total\_rows)    *# calculate the probabilities on each testing set* for i in range(len(class\_summaries)):  mean, stdev, \_ = class\_summaries[i]  probabilities[class\_value] \*= self.calcProb(row[i], mean, stdev)  return probabilities   def predict(self, X\_test):  row = X\_test.tolist()  label=[]  *# iterate our testing set and get the likehood class* for i, c in enumerate(row):  probabilities = self.calcClassProb(c)  best\_label, best\_prob = None, -1  for class\_value, probability in probabilities.items():  *# list all probabilities each class and select the highest one* if best\_label is None or probability > best\_prob:  best\_prob = probability  best\_label = class\_value  label.append(best\_label)  return label   *# In[10]:* nb = NaiveBayesClassifier() result = nb.fit(X\_train,y\_train) y\_pred = nb.predict(X\_test)   *# In[11]:* y\_pred = np.asarray(y\_pred)  target\_names = [**'Iris-setosa'**, **'Iris-versicolor'**, **'Iris-virginica'** ] print(metrics.classification\_report(y\_test, y\_pred, target\_names=target\_names))  print(**'Accuracy score: '**, (y\_test == y\_pred).sum() / len(y\_test)) print(**'Misclassified examples: %d'** % (y\_test != y\_pred).sum())   *# In[12]:* def combineData(X, y):  dataset = []  for i in range(len(X)):  dataset.append(X[i].tolist()+[y[i]])  return dataset  def crossValidationFold(dataset, folds=3):  dataset\_split = list()  dataset\_copy = list(dataset)  fold\_size = int(len(dataset) / folds)  for i in range(folds):  fold = list()  while len(fold) < fold\_size:  index = random.randrange(len(dataset\_copy))  *#index = range(len(dataset\_copy))* fold.append(dataset\_copy.pop(index))  dataset\_split.append(fold)  return dataset\_split  *# Calculate accuracy percentage* def accuracyMetric(actual, predicted):  correct = 0  for i in range(len(actual)):  if actual[i] == predicted[i]:  correct += 1  return correct / float(len(actual)) \* 100.0   *# Evaluate an algorithm using a cross validation split* def crossValidation(X, y, algorithm, n\_folds, \*\*kwargs):  dataset = combineData(X,y)  folds = crossValidationFold(dataset, n\_folds)  scores = list()  for fold in folds:  train\_set = list(folds)  train\_set.remove(fold)  train\_set = sum(train\_set, [])  test\_set = list()  for row in fold:  row\_copy = list(row)  test\_set.append(row\_copy)  row\_copy.remove(row\_copy[-1])    y\_pred = algorithm(train\_set, test\_set, \*\*kwargs)  y\_test = [row[-1] for row in fold]  accuracy = accuracyMetric(y\_test, y\_pred)  scores.append(accuracy)  return scores   def naiveBayes(train\_set, X\_test):  y\_train = [row[-1] for row in train\_set]  X\_train = [row[:-1] for row in train\_set]  X\_train = np.array(X\_train)  y\_train = np.array(y\_train)  X\_test = np.array(X\_test)    nb = NaiveBayesClassifier()  result = nb.fit(X\_train,y\_train)  y\_pred = nb.predict(X\_test)   return y\_pred  def decisionTree(train\_set, X\_test, max\_depth=5):  y\_train = [row[-1] for row in train\_set]  X\_train = [row[:-1] for row in train\_set]  X\_train = np.array(X\_train)  y\_train = np.array(y\_train)  X\_test = np.array(X\_test)    dt = DecisionTreeClassifier(max\_depth)  result = dt.fit(X\_train, y\_train)  y\_pred = dt.predict(X\_test)   return y\_pred   *# In[13]:* names = [] result = []  random.seed(0) dt\_scores = crossValidation(X, y, decisionTree, n\_folds = 10, max\_depth = 8) print(**'Scores: %s'** % dt\_scores) print(**'Mean Accuracy: %.3f%%'** % (sum(dt\_scores)/float(len(dt\_scores)))) result.append(dt\_scores) names.append(**'Decision Tree: C4.5 Algorithm'**)   *# In[14]:* random.seed(0) nb\_scores = crossValidation(X, y, naiveBayes, n\_folds = 10) print(**'Scores: %s'** % nb\_scores) print(**'Mean Accuracy: %.3f%%'** % (sum(nb\_scores)/float(len(nb\_scores)))) result.append(nb\_scores) names.append(**'Naive Bayes'**)   *# In[15]:   # Boxplot Graph* fig = plt.figure() fig.suptitle(**'Algorithm Comparison'**) ax = fig.add\_subplot(111) plt.boxplot(result, showmeans=True) ax.set\_xticklabels(names) plt.show()   *# In[16]:   # Bar Graph* n\_groups = 10  *# create plot* fig, ax = plt.subplots() index = np.arange(n\_groups) bar\_width = 0.35 opacity = 0.8  rects1 = plt.bar(index, dt\_scores, bar\_width,  alpha=opacity,color=**'b'**,label=**'Decision Tree (C4.5)'**)  rects2 = plt.bar(index + bar\_width, nb\_scores, bar\_width,  alpha=opacity,color=**'g'**,label=**'Naive Bayes'**)  plt.xlabel(**'Training #'**) plt.ylabel(**'Accuracy Score'**) plt.title(**'Algorithm Comparison'**) plt.xticks(index + bar\_width, (**'1'**, **'2'**, **'3'**, **'4'**, **'5'**,  **'6'**, **'7'**, **'8'**, **'9'**, **'10'**)) plt.legend()  plt.tight\_layout() plt.show() |

# Output



# Conclusion

In the boxplot graph below, it shows the performance comparison between Naïve Bayes and Decision Tree classifier using the 10-fold Cross Validation. Also, the bar graph shows the accuracy score of each k fold. We determined that the Naïve Bayes classifier is slightly performed better, with an average accuracy score of 95%, than the Decision Tree classifier with 94% average accuracy score.

