Data Exploration

In this project, we are looking at the data file *breast-cancer-wisconsin.data*. This file contains information on 699 separate diagnosis of breast cancer. Some of these attributes include clump thickness, uniformity of cell size and shape, marginal adhesion, and number of bare nuclei. This data file was supplied in *.data* format. However, to help streamline the process of reading data, I converted the file to a *.cvs* file. Furthermore, to increase clarity when reading data, I added a row to the beginning of the file with labels for each column of attribute variables. In total, there are eleven attributes for each breast cancer diagnosis. The supplied pdf for project 3 contains a complete index that describes the names and types of each attribute. The available attributes are as follows:

Attributes	Value	Type
0. Sample Code Number	ID Number	Identifier
1. Clump Thickness	1-10	numerical
2. Uniformity of Cell Size	1-10	numerical
3. Uniformity of Cell Shape	1-10	numerical
4. Marginal Adhesion	1-10	numerical
5. Single Epithelial Cell Size	1-10	numerical
6. Bare Nuclei	1-10	numerical
7. Bland Chromatin:	1-10	numerical
8. Normal Nucleoli	1-10	numerical
9. Mitosis	1-10	numerical
10. Class	2: Benign or 4: Malignant	classification

As you can see, the numerical attributes all have integer values with a range between 1 and 10. The last attribute, “Class”, is a classification variable that will either have the value 2 or 4, denoting whether the cancerous cells are benign or malignant. Since this attribute only has two values, it can easily read as a binary variable. Looking through the supplied data, I found that the attribute “Bare Nuclei” has 16 missing variables. Instead of a numerical value, the attribute value is a “?”. Outside of these, I was unable to find any other missing values in the data. Furthermore, there are no apparently incorrect or illegal values within this set of data.

Data Preparation

The next step of this project is to properly examine and analyze the given data. The computed statistics for each attribute can be found in the chart on the next page. These calculations can help us with future analysis processes such as standardization of variables.

Attribute #	Mean	SD	Min	Max	# of values
1	4.418	2.816	1	10	699
2	3.134	3.051	1	10	699
3	3.207	2.972	1	10	699
4	2.807	2.855	1	10	699
5	3.216	2.214	1	10	683
6	3.525	3.628	1	10	699
7	3.438	2.438	1	10	699
8	2.867	3.054	1	10	699
9	1.589	1.715	1	10	699
10	2.690	0.951	2	4	699

Looking at the chart above, you can see that the means and standard deviations for these attributes are relatively small and close to each other. Furthermore, the min and maxes all fall within expected values, confirming that there are no illegal values within this data set. Most importantly, the number of values for the 5th attribute, "Bare Nuclei" is 16 less than the expected number of 699. This reiterates the fact that there is a good amount of data missing for that attribute. To fix this issue of missing values, we must decide on an imputation strategy to replace them.

The imputation strategy I opted to use is the cold deck imputation method. With this strategy, you would systematically choose a value from another individual cancer diagnosis who has similar values for other variables. I chose this method because it effectively allows me to set constraints to only possible values. Usually, I opt to use the mean imputation method. However, these are integer-based numbers referring to specific statistics about cells. As a result, replacing the missing values with the attribute's mean would only serve to further decrease the accuracy of the data and my learning algorithm.

Dimension Reduction & Data Separation

In regards to reducing dimensions, I decided to remove the first attribute, "Sample Code Number", because its purpose is purely for identification of results. This attribute has no informational value that could help my algorithm make a more accurate prediction. For the purposes of this project, I am separating my data into three sections: training, evaluation, and testing data sets. The percentages and exact number of responses are as follows:

Data Set	Percentage	Count
Training	70%	489
Evaluation	15%	105
Testing	15%	105
Total	**100%**	**699**

Implementation of *k*-Nearest-Neighbors Algorithm

For this project, I used python to implement a k-nearest-neighbors algorithm with the purpose of predicting whether the data from a cancer diagnoses indicates a benign or malignant tumor. The implementation can be found in the included file *kNN.py*. The training data set I used for training this algorithm can be found in the included file *breast-cancer-wisconsin-te.csv.* For this implementation, to increase clarity of the code, I broke the algorithm down into separate helper functions. The first function, *loadData*, reads the supplied .csv file and randomly splits the file into two separate data sets. The next helper function I created for use in my implementation was *calcDist*. *CalcDist* takes two data points and calculates the Euclidean distance between the two. This distance will be used as the calculated similarity between data instances. Since all of my values (with the exception of the result variable) are numerical and integer-based, this was the only type of distance calculation I needed to implement for this algorithm.

The next two functions, *findNeighbor*s and *predict*, handle the bulk of the processing for this algorithm. The first, *neighbors*, is relativity simple. It iterates through each data instance finding *k* most similar, complimentary instances. In other words, it finds the instances with the shortest total distance between each other. These instances, or neighbors, are then stored in a list and returned. Next, the function, *predict*, creates a predicted response based on the *k* neighbors returned from the previous function.

Finally, for the purpose of analysis, I implemented the function, *analyze*. This function iterates through the list of my algorithm's predictions and checks them against the true, expected values given in the data. For each comparison, the function decides if it was a true positive (TP), true negative (TN), false positive (FP), or a false negative (FN). After iterating through all of the predicted answers, *analyze*, then calculates the overall accuracy of the algorithm. In addition, the function calculates the true positive rate (TPR), positive predictive value (PPV), true negative rate (TNR), and the F_1 Score of the algorithm using the counted TN, TP, FN, and FP values. The functions used for these calculations are below.

I can use these calculated values as performance metrics to accurately determine the optimal *k* for use in my *k*-nearest-neighbor algorithm. Below, are the confusion matrices created using the data produced by my function, *analyze*. These figures are from processing the testing and evaluation sets of data for the *k* values 2 to 8, 17, and 33.

		Predicted Class	
		Benign	**Malignant**
True Class	**Benign**	**TN**	**FP**
	Malignant	**FN**	**TP**

Statistic	**Formula**
Accuracy	(TN + TP) / (TN + TP + FN +FP)
TPR	TP / (TP + FN)
PPV	TP / (TP + FP)
TNR	TN / (TN + FP)
F_1 Score	(2 x PPV x TPR) / (PPV + TPR)

K = 2		Predicted Class	
		Benign	Malignant
True Class	Benign	122	7
	Malignant	3	69

K = 3		Predicted Class	
		Benign	Malignant
True Class	Benign	136	1
	Malignant	8	49

K = 4		Predicted Class	
		Benign	Malignant
True Class	Benign	145	0
	Malignant	5	58

K = 5		Predicted Class	
		Benign	Malignant
True Class	Benign	138	4
	Malignant	2	58

K = 6		Predicted Class	
		Benign	Malignant
True Class	Benign	125	2
	Malignant	6	60

K = 7		Predicted Class	
		Benign	Malignant
True Class	Benign	129	10
	Malignant	2	62

K = 8		Predicted Class	
		Benign	Malignant
True Class	Benign	123	4
	Malignant	5	66

K = 17		Predicted Class	
		Benign	Malignant
True Class	Benign	133	2
	Malignant	2	57

K = 33		Predicted Class	
		Benign	Malignant
True Class	Benign	126	6
	Malignant	3	54

k	Accuracy	TPR	PPV	TNR	F1 Score
2	95.02%	0.958	0.908	0.946	0.932
3	95.36%	0.860	0.980	0.993	0.916
4	97.60%	0.921	1.000	1.000	0.959
5	97.03%	0.967	0.935	0.972	0.951
6	95.85%	0.909	0.968	0.984	0.937
7	94.09%	0.969	0.861	0.928	0.912
8	95.45%	0.930	0.943	0.969	0.936
17	97.94%	0.966	0.966	0.985	0.966
33	95.24%	0.947	0.900	0.955	0.923

From the confusion matrices and chart above, 17 appears to be the optimum for *k*. Overall, the accuracy of the algorithm was most accurate when k = 17. In addition, that run of the algorithm produced the highest F_1 Score. As a result, I will perform further testing of my k-nearest-neighbors implementation with k = 17. The analytic results of running my algorithm on the testing data set, *breast-cancer-wisconsin-test.csv,* is shown below.

***K* = 17**		**Predicted Class**	
		Benign	**Malignant**
True Class	**Benign**	**20**	**1**
	Malignant	**0**	**13**

	k = 17
Accuracy	97.06%
TPR	1.000
PPV	0.929
TNR	0.952
F1 Score	0.963

While the overall accuracy of the algorithm slightly declined by 0.88%, the F_1 Score is extremely close to that of the training and evaluation run for k = 17. This very slight decrease can be explained by the smaller amount of overall data instances to train and infer from.

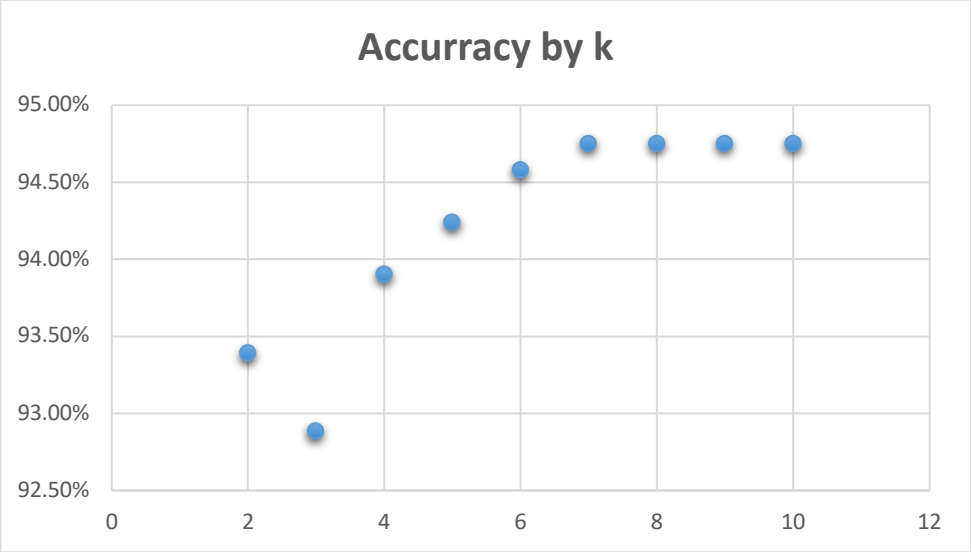
Implementation of Decision Tree Classifier

For this project I used python for my implementation for a decision tree classifier. Like my previous implementation, I built this one using separate helper functions. First, I implemented functions to set up and format the given data. Once my build_tree function is called, the program iterates through each attribute and its values and determines the best split as it runs. Next, my program calculates the root node using the given data. After the root node is determined, my program recursively finds the next best split to effectively build the decision

tree. Once there are no more splits to be made, or one of the constraints are reached, the program exits and prints the tree.

The next section of my implementation surrounds applying a constructed decision tree to a data set and returning an array of predicted classifications. The function I wrote to handle this task recursively navigates down the given decision tree and checks if each node visited is a terminal classification value and should be returned as a prediction. To test the accuracy of both my created decision trees and the resultant predictions, I tested my algorithm on the training and evaluation data sets and calculated the accuracy of the each run with varying maximum depths. The results of k = 2-10 runs are below.

k	Accuracy
2	93.39%
3	92.88%
4	93.90%
5	94.24%
6	94.58%
7	94.75%
8	94.75%
9	94.75%
10	94.75%



Looking at the graph and chart above, overall, the algorithm became more accurate as k increased past 3. Furthermore, after k = 7, the algorithm's accuracy began to level out at 94.75%. As a result, I will pick k = 7 to conduct my final run analysis using the testing data set. The resultant accuracy of the test run with k = 7 is 94.29%. This accuracy is surprisingly close to the training run as my algorithm had few data instances to work with.