Task 2 – Decision Tree

2802 – INTELLIGENT SYSTEMS

JESSY BARBER

31/05/2021

Software Design

Similarly, to Task 1, the votes.csv file is initially read and stored in a variable called dataset using the pandas module. The list of party members is then stored in Party, and the Republicans and Democrats are stored in the same fashion as the species of Iris in task 1. The Attributes variable is then initialized with the list of attributes across the first row of the data and acts as the list of voting features. The driver function will be explained in more depth later, but for now the program begins by asking the user to input the value of the training data split in the form 0.X, the value of k for KNN, and the learning curve step size.

Def split_data(percentage):

The basic operation of the split_data function is to split the training and testing data as per the percentage parameter. Firstly, the function extracts all of the data columns and stores this in a variable called extract_data. The numpy random.permutation function is now called to randomly shuffle the order of the data. This is stored in a variable called random_data which is then re-assigned to the extract_data variable. The training_size variables is initialized with the shape of the extract_data multiplied by the parameter percentage. This percentage of data is now stored in the training_df variable which represents the training data. The rest of the data is stored in the test_data variable which represents the testing data. Variables train_data and test_data are now initialized with the values of training_df and testing_df. Y_true and x_true are then initialized with the last column of the train_data and test_data which stores the corresponding party type for each data point.

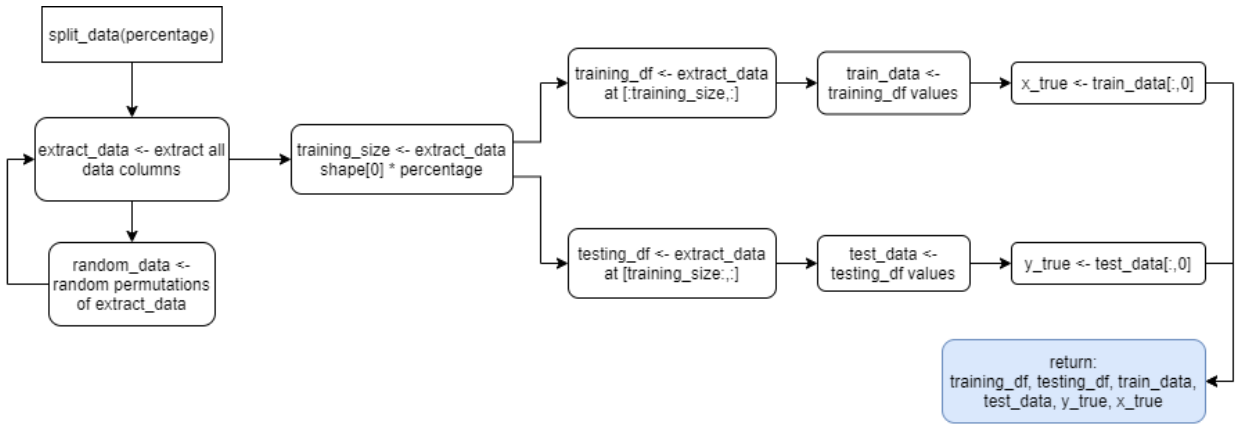

Figure 1: split_data Software Design

Def information_gain(dataframe, attribute):

The information gain function is responsible for finding the information gain for each attribute that is passed as a parameter. It does this by subtracting the call of two functions, party_entropy and attribute_entropy. This will be explained next but the information_gain function subtracts the return value of these two functions, stores the result in a variable called IG and returns IG.

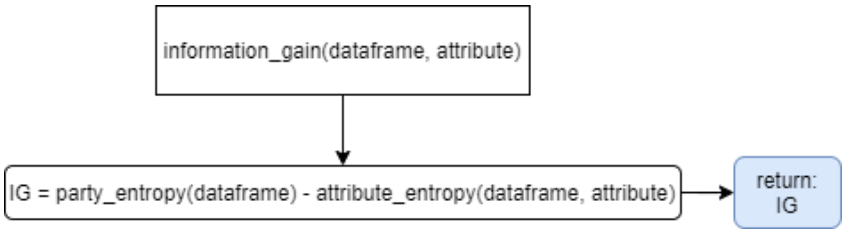

Figure 2: information_gain Software Design

Def party_entropy(dataframe):

The party_entropy function represents the Node Entropy for decision trees. It assigns a variable called party_keys to the first column containing the list of party members, and party_names is created to represent the unique elements of this list. As a result, party_names will contain 'democrat' and 'republican'. For each party a weighted total will be created from the count of that party within dataframe divided by the length of the number of people voting. For each party the total entropy is then the summation of the weighted totals multiplied by $\log_2$ of the weighted total. This is more clearly expressed in the following mathematical formula (T, 2019):

$$E(party) = \sum_{i=1}^{c} p_i \log_2 p_i$$

The total entropy E(party) is now returned by the party_entropy function.

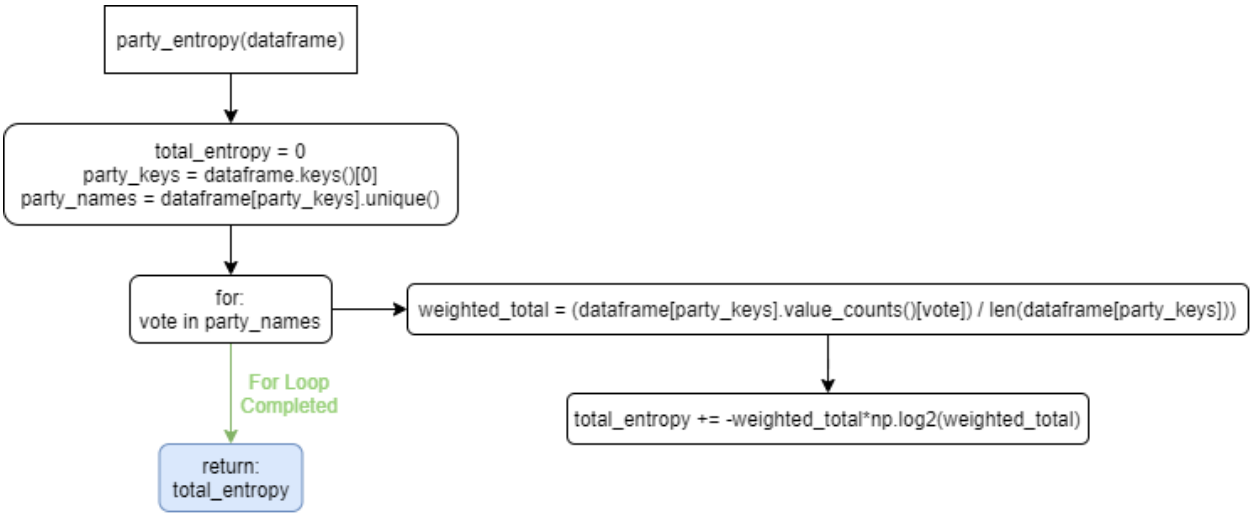

Figure 3: party_entropy Software Design

Def attribute_entropy(dataframe, attribute):

Unlike the party_entropy, the attribute entropy finds the entropy of the party for each target attribute as a weighted proportion of the members of the party and voters of the attribute. The party_keys and party_names are defined the same as in the party_entropy function, but attributes is now initialized as a list of the unique attributes within the dataframe. For each member of the party and for each attribute the weighted_total will be found for the attribute and party and

summated into a total_entropy function the same as in party_entropy. An eps value is added to the weighted total calculating to avoid any potential divisions by 0. A variable called population_fraction is now initialized which is assigned the value of the length of the dataframe for that attribute containing a vote divided by the length of the dataframe itself. The total_attribute_entropy is now defined as the summation of the negative value of population_fraction multiplied by the total_entropy. The absolute value of total_attribute_entropy is now returned. This entropy formulation can be expressed clearer using the following formula (T, 2019):

$$E(Party|Attribute) = -\sum_{j=1}^{n} \frac{N_{attributej}}{N_{dataframe}} \sum_{i=1}^{c} p_i \log_2 p_i$$

Where, n = party, $N_{attributej}$ = length of dataframe for attribute for party vote, $N_{dataframe}$ = dataframe length.

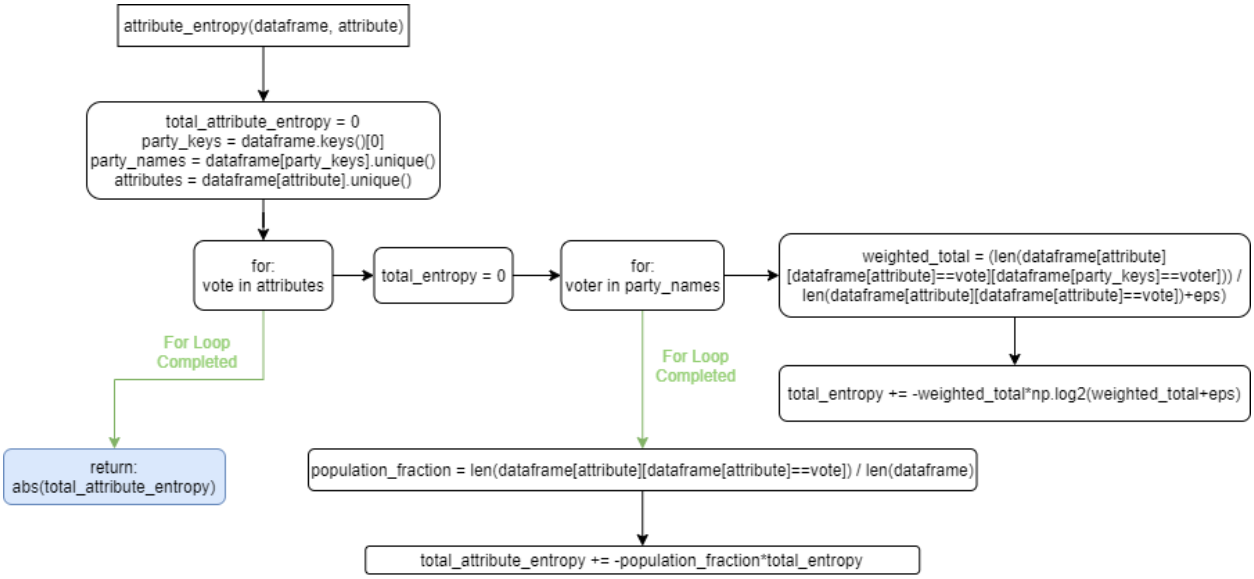

Figure 4: attribute_entropy Software Design

Def decision_tree(current_data, original_data, attributes, party_names="party", parent_node=None):

The decision_tree function is the heart of building the decision tree network. There are 4 conditional statements here that are called depending on the status of the current node and tree that is being created and will stop node generation if any of these first three if conditionals are evaluated as true. The first if conditional evaluates whether the node is a pure subset in which it only contains one type of party. The second evaluates whether the end of the dataset has been reached and the third evaluates whether all of the attributes have been explored. If not of these conditions have been met, the function will recursively generate child nodes. It does this by iterating through the attributes and finding the attribute with the highest information gain. This will determine the best opportunity to split the tree and create a new node based on this attribute. The tree is created in the form of a dictionary, and a new attribute list is created that include the attributes that are not the attribute with the current max information gain value. For each value

that is unique inside of the current_data indexed by the highest attribute, a child_data variable will be created which excludes the columns that include the highest attribute. The decision_tree function is now called recursively with child_data as the new current_data and current_data as the original_data. The attributes parameter is now called with the new attributes list and the new parent_node is used an argument. After the recursion is complete the tree is returned by the function.

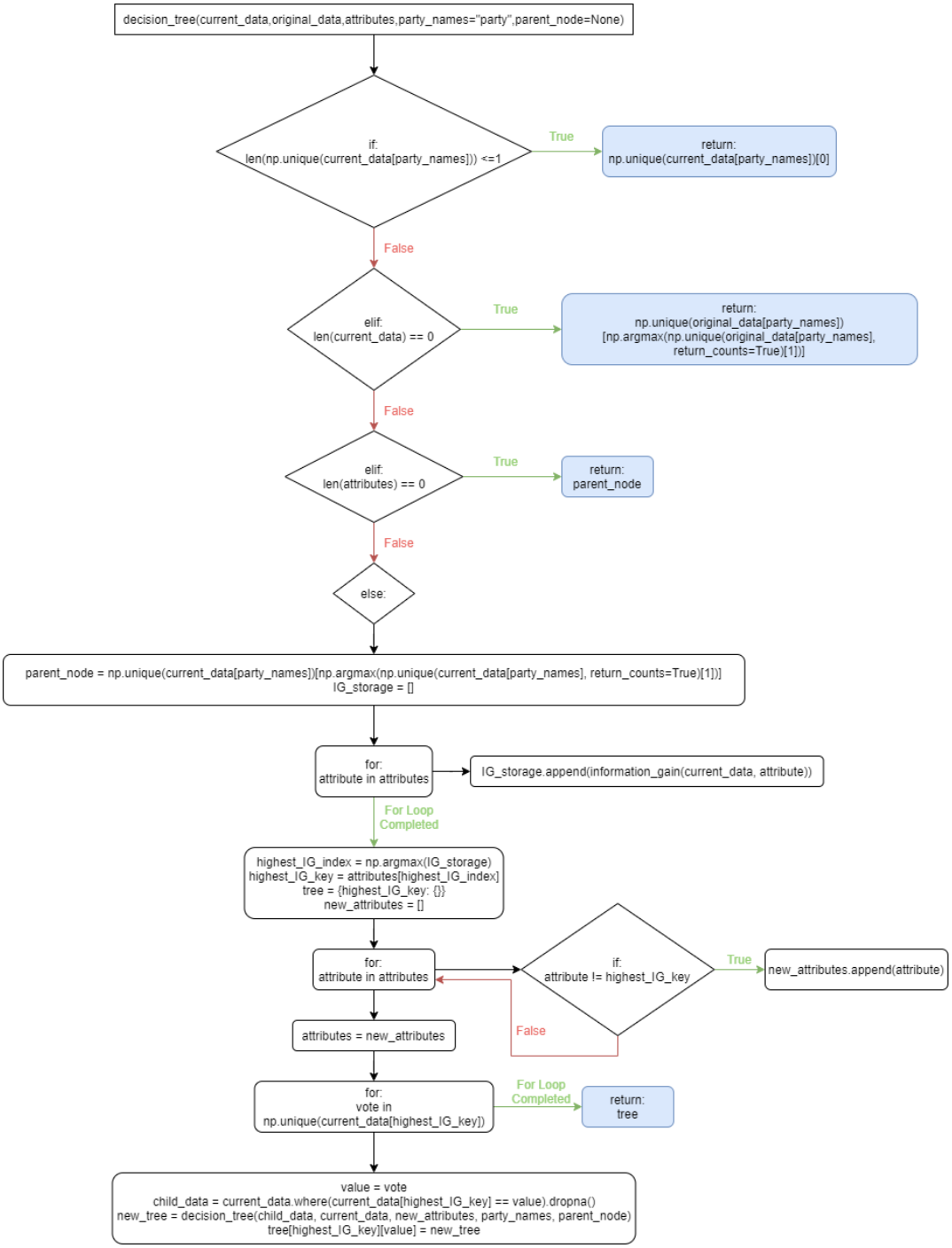

Figure 5: decision_tree Software Design

Def accuracy(current_data, current_tree):

This accuracy function is specifically designed for finding the accuracy for data in the decision tree. Current_data and current_tree are passed as parameters which represent testing_df or training_df and the developed tree arguments in the driver code. A variable called tree_query is created to store a conversion of the current_data in a dictionary format. A for loop now iterates over the length of current_data and the classify function is called with its return value stored in a list called predicted. The accuracy is now the np.sum of the elements where predicted is equal to the party columns in current_data divided by the length of current_data. This is multiplied by 100 and returned along with the predicted list.

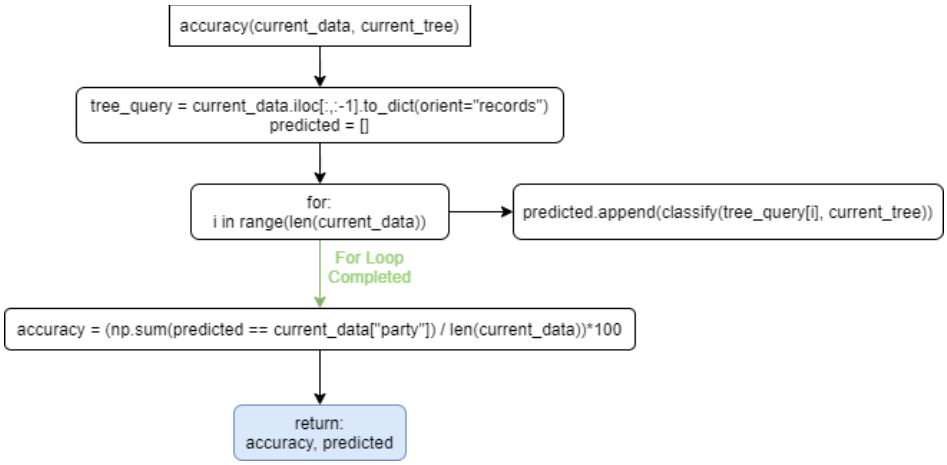

Figure 6: accuracy Software Design

Def classify(tree_query, current _tree, default = 1):

The classify function operates with a for loop that iterates over the list of tree_query keys from the tree_query dictionary that is passed as a parameter. Fir the keys in this list, a try and except block is called to test the assignment of current_tree indexed by key and tree_query[key] to the variable result. If this fails, the except block is called which will return the default parameter of 1. This handles exceptions for where there is an incorrect value at this index. If the try statement was successful, another if conditional statement will be called with isinstance to check whether result is of the type dict. If this is true, then the classify function is called recursively with the tree_query and result instead of the current_tree. If the if conditional statement evaluates to false, result is returned.

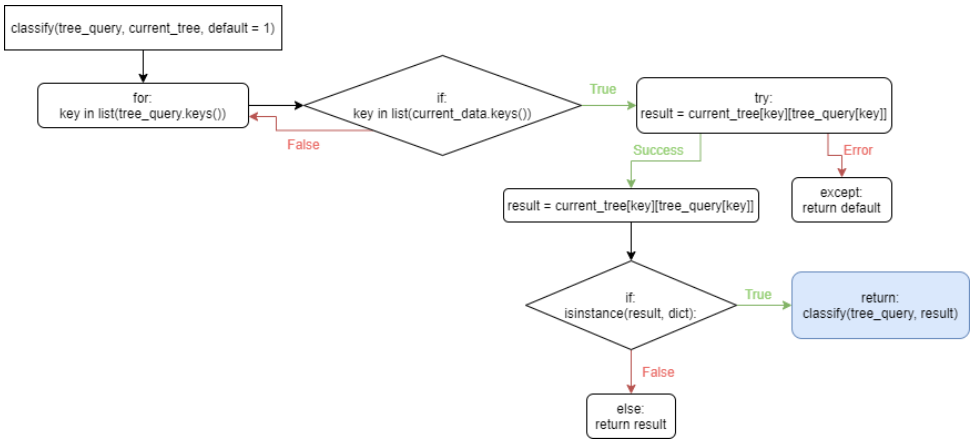

Figure 7: classify Software Design

Def euclidean_distance(x_test, x_train):

Def knn(x_test, x_train, k):

Def knn_classify(x_test, x_train, k):

The above functions were taken from task 1 are exactly the same except for some minor details. Because of this, the software design diagrams for these functions will not be shown. The only major difference to these functions is in the knn_classify function which no longer takes euc as a Boolean parameter. This is because the only distance metric being used for knn is the Euclidean distance.

Def knn_accuracy(true, prediction):

The knn_accuracy function is important because it allows for the knn classifier to attain its TP, TN, FP, and FN values to calculate accuracy, precision, recall and f1. True and prediction are passed as parameters, true being a list of the expected values and prediction being a list of the predicted knn values. These are all initialized to 0 including a correct_count variable. A for loop is executed which iterates over the length of the true list. Inside are four if conditional statement which determine the counters to be incremented. The first evaluates whether the element in true and prediction are both democrat. If this is true, the correct_count and TN are incremented. The next evaluates whether the element in true and prediction are both republican. If this is true, the correct_count and TP are incremented. The third evaluates whether the element in true is republican and the element in prediction is democrat. This will increment the FN counter. The last conditional statement is called which represents the case in which true is democrat and prediction is republican. This will increment the FP counter. After the for loop has finished execution, an accuracy variable will be assigned the value of the correct_count counter divided by the length of the true list. This is multiplied by 100 to represent the percentage accuracy. TN, TP, FP, FN, and accuracy are now returned.

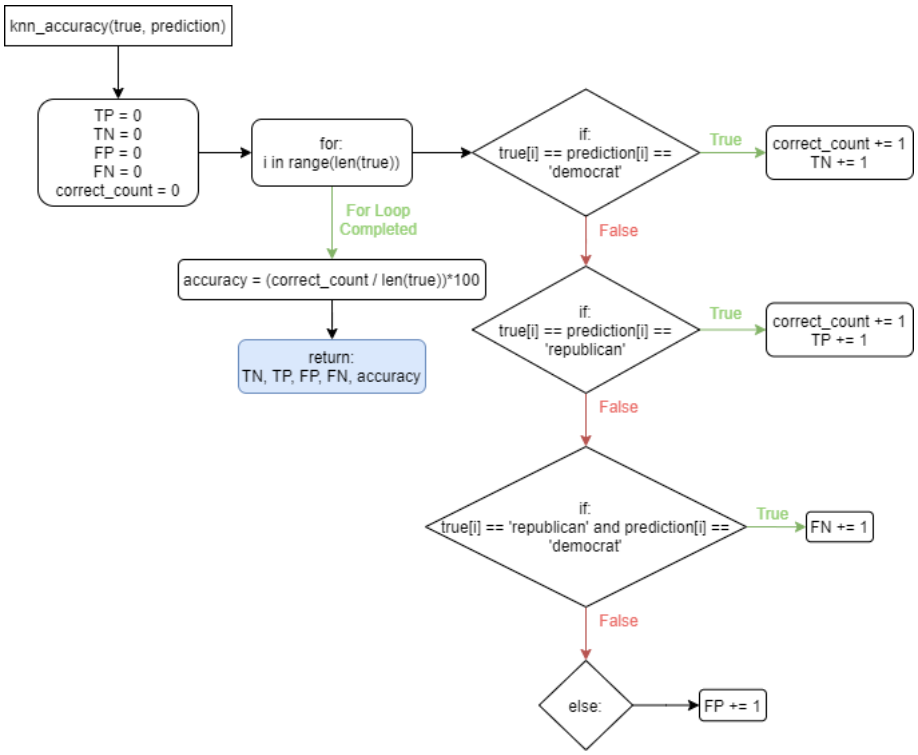

Figure 8: knn_accuracy Software Design

Def learning_accuracy(true, prediction):

The learning_accuracy function is a specialized version of the knn_accuracy function is designed such that the decision tree elements can be passed through and only the accuracy is returned. This means that only the first two if conditional statements within the knn_accuracy function are used since the accuracy only needs the correct_count counter. The accuracy is calculated and returned in the same fashion as the knn_accuracy function.

Driver code:

All of the functions have now been covered, the driver code is now used to call these functions and execute the output for the program. As mentioned before the driver code starts by asking the user for the percentage, k, and learning_curve_step inputs. These are then converted to their respective variable types. The split_data function is now called with the user input percentage as its argument. A variable called train_data_tree is now assigned the return tree of calling the decision_tree function. The accuracy function is now called with testing_df and the newly formulated tree, returning into variables test_accuracy and test_prediction. This function is called again but with the training_df, returning into train_accuracy and train_prediction. A confusion matrix is now made for the testing_df and training_df and these are converted to data frames using pd.DataFrame. test_confusion_matrix and train_confusion_matrix are now assigned the pd.crosstab of the elements of the confusion matrix data frames to correctly tabulate the TN, TP, FN, and FP elements. The testing and training TN, TP FN and FP variables are created in a similar fashion to the knn_accuracy function but instead extracts this information from their corresponding confusion matrix. The test_DT_accuracy and train_DT_accuracy can now be found as an alternative to using the accuracy function and can be used to compare these two accuracies. The accuracy, recall, precision and f1 can now be found as follows (Wikipedia, 2021):

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$f1 = 2 * \frac{precision * recall}{precision + recall}$$

These values are printed to the screen for the decision tree testing and training data, and the knn testing and training data. Now, the learning curve for the decision tree and knn is called. The learning curve operates by iterating from the length of the testing_df to the length of the dataset with steps of learning_curve_step specified by user input. Just before the decision tree for loop, two lists are initialized called test_DT_accuracy_var, to store the accuracy values over different data sizes, and test_DT_size, to store the size of the training data as it grows over time. Inside the for loop, each iteration i is appended to test_DT_size and train_DT_set is initialized with the

assigned value of dataset.loc[:i]. Another list is initialized called test_DT_predictions and test_DT_true which stores the testing_df party column values. A growing_DT variable is created to store the current decision_tree return value which is called with train_DT_set, train_DT_set, and train_DT_set.columns[1:]. The regular accuracy function is now called with testing_df and growing_DT which returns into discard_accuracy and DT_prediction. Discard_accuracy acts as a throwaway variable since the learning_accuracy function will give a more accurate accuracy value. The DT_prediction is used to call the learning_accuracy function which takes y_true and the DT_prediction variable as arguments, returning its result into DT_accuracy. This DT_accuracy is finally appended to the test_DT_accuracy_var list.

The learning curve for KNN is similar to the decision tree learning curve. Like the decision tree, before the for loop two lists are initialized called KNN_accuracy_var and KNN_size. The for loop now iterates over the same stop start and step arguments as the decision tree learning curve. Inside the for loop, the current iteration i is appended to the KNN_size list. The KNN_train_set variable is now initialized with the assignment of dataset.loc[:i] and the KNN_train_values is initialized to the KNN_train_set values. Two new lists called KNN_predictions and KNN_true are now initialized which is followed by a for loop iterating over test_data. For each iteration j of this for loop, the return of the function knn with calling arguments j, KNN_train_values and k are appended to the KNN_predictions list. After this for loop, the return value of learning_accuracy called with arguments y_true and KNN_predictions is stored in the KNN_accuracy variable. The variable is then appended to the KNN_accuracy_var list.

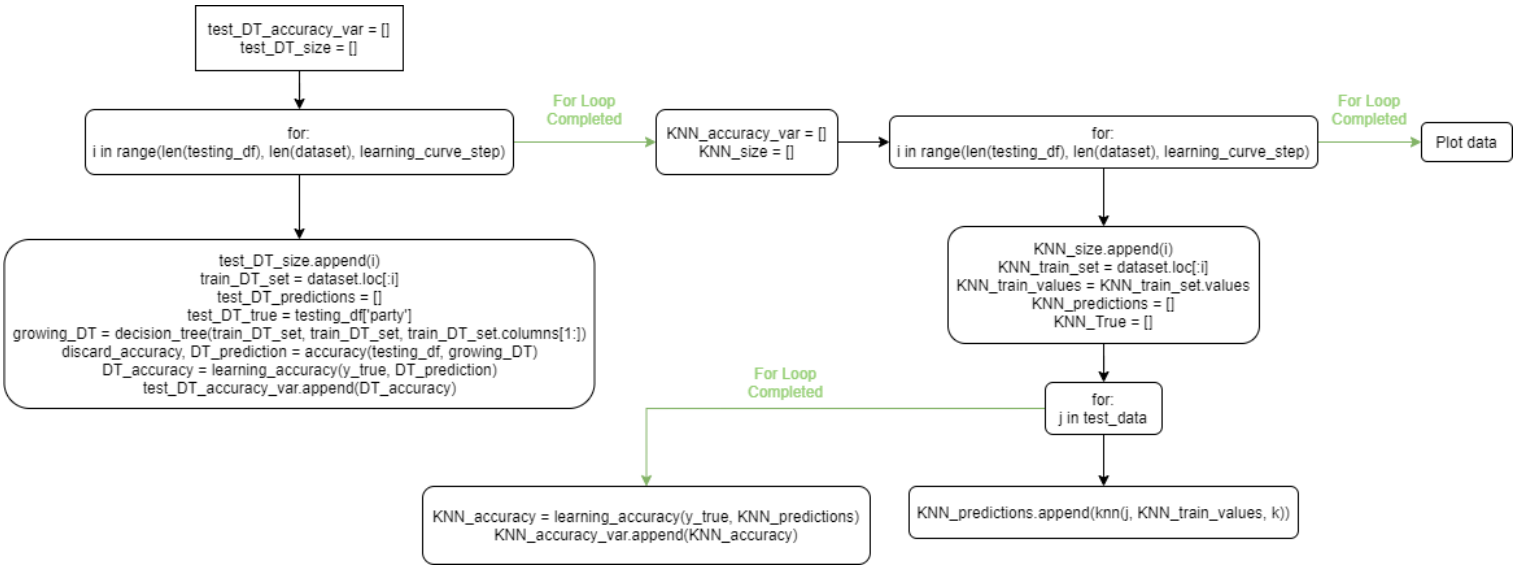

Figure 9: Learning Curve Software Design

Results

For testing purposes, 70% of the data was split for training and 30% for testing. The value of k was determined by finding the square root of the test data. Since the size of the data is 435 congress members, 30% of this is 130, and the square root of this value is approximately 11. After testing, the best learning curve step size was found to be 7.

```
Length of dataset:  435

Enter percentage of training data (0.X):0.7

Enter k value for KNN:11

Enter the learning curve step size:7
--------------------------------Decision Tree Testing Data--------------------------------
Test data accuracy:  93.89312977099237 %
Test data cm accuracy: 95.34883720930233 %
Test data precision: 0.9642857142857143
Test data recall: 0.9310344827586207
Test data f1: 0.9473684210526316
--------------------------------Decision Tree Training Data--------------------------------
Train data accuracy:  98.35526315789474 %
Train data cm accuracy: 100.0 %
Train data precision: 1.0
Train data recall: 1.0
Train data f1: 1.0
------------------------------KNN Testing Data-----------------------------------------
Test data accuracy: 94.65648854961832 %
Test data precision: 0.9193548387096774
Test data recall: 0.9661016949152542
Test data f1: 0.9421487603305785
------------------------------KNN Training Data-----------------------------------------
Train data accuracy: 35.85526315789473 %
Train data precision: 0.35855263157894735
Train data recall: 1.0
Train data f1: 0.5230018098404814
```

Figure 10: DT & KNN Testing and Training Results

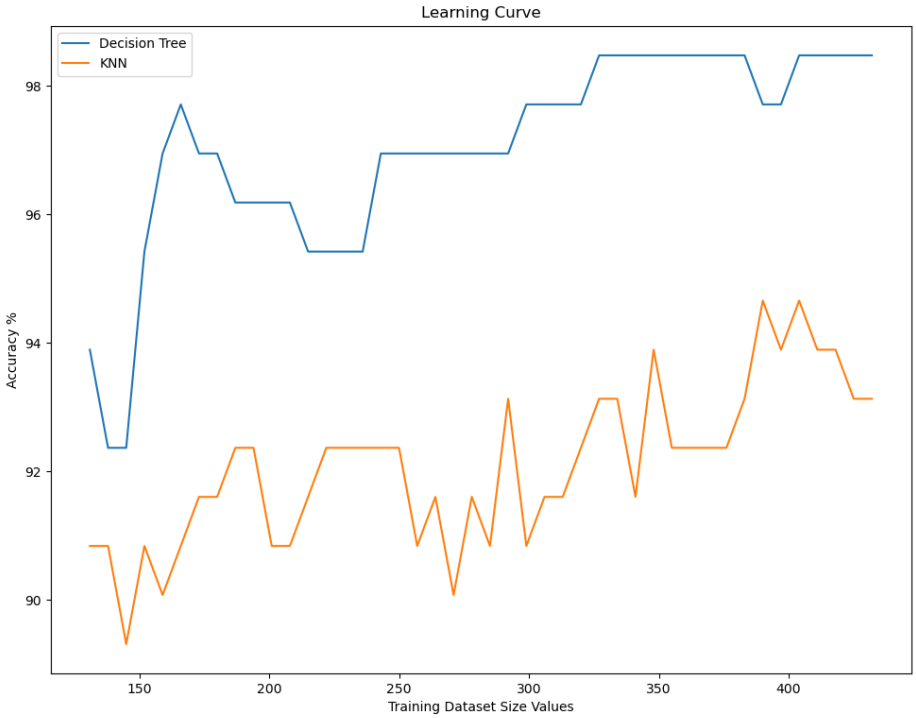

Figure 11: DT and KNN Learning Curve

Conclusion

Figure 10 shows the results of the decision tree and KNN testing and training data with a 70% split of training data and k value of 11. There is a deviance between the accuracy results of the testing and training data for the accuracy value returned by the accuracy function, and the accuracy achieved with the confusion matrix. This is a small 1% - 2% difference but gives indication that the confusion matrix returns a more expected accuracy result. According to these results and additional testing, the decision tree test data returns an accuracy of 95-98%, and the training data returns an accuracy of 100%. This is expected of the training data since the training values are being compared to themselves. As for the KNN, there is a testing accuracy between around 90% to 95% and a training accuracy between 30% to 60%. The training data is less reliable for the knn model, so for the sake of this conclusion only the testing models will be compared. It can clearly be seen that the decision tree testing data is able to achieve a much higher overall accuracy when compared to the knn method. In direct comparison to task 1, this method of learning is much more efficient and is capable of greatly increasing its accuracy as the size of the training data increases. These observations indicate that the knn model produces a more underfit model of the data. Although the decision tree is a better model for learning, there seems to a be a small amount of overfitting. This can be seen close to the training dataset size of 400, and indicates that there is an aspect of noise or random fluctuation that the training data is picking up and learned as concepts by the model (Brownlee, 2016). After this fluctuation the accuracy stabilizes at over 98% and is overall present the decision tree method as a good statistical fit and learning method for the data.

Bibliography

Brownlee, J. (2016, August 12). *Overfitting and Underfitting With Machine Learning Algorithms*. Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/

T, S. (2019, January 11). *Entropy: How Decision Trees Make Decisions*. Retrieved from towards data science: https://towardsdatascience.com/entropy-how-decision-trees-make-decisions-2946b9c18c8

Wikipedia. (2021, May 13). *Confusion Matrix*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Confusion_matrix