Project Name: C4.5 Decision Tree

Your Name(s): Aditya B., Gaurav G., Hitesh B.

University of Washington: CSE 312 Summer 2020

1 Application

Computer-based models have become incredibly powerful to the field of meteorology for weather forecasting tools, both in the short and long terms. In simple terms, forecasting decisions are made based on collections of past and present quantitative data. This process has existed for centuries, yet continues to evolve in parallel with our scientific boundaries.

Here, we can apply decision tree algorithms quite effectively: we can develop models that map the given atmospheric data to each of their weather outcomes. Therefore, when we obtain future instances of data, we can make decisions on what outcomes they map to. What's neat about decision trees, aside from their intuitive shape, is their versatility: they can be applied to all shapes of data and often require little amounts of pre-processing of the features. In a very particular use case, given atmospheric data in an area corresponding to instances of tornadoes vs strong winds vs no wind, we can make determinations on whether a tornado is occurring/forming from data collected in real-time. The models' determinations will have a huge influence on disaster preparedness and other forms of preparation given a specific wind event.

2 Dataset

link to data set: https://www.kaggle.com/grubenm/austin-weather

Our particular Decision Tree algorithm relies on categorical features because it allows the model to train on fewer categories, which we hypothesize will improve accuracy in categorization. This makes the model easier to classify because each attribute does not have an infinite range. Therefore, when we picked our data set, we were looking for one that classified on categorical features as opposed to one that incorporated ranges of numbers. We quickly found out that most data sets on Kaggle utilize numbers, so we decided to set 3 groups for each attribute. We decided on a data set that contained the weather in Austin because it had 18 different attributes of information that ranged over 3 years (a little over 1300 days). This data set contained all continuous numbers, so we set 3 groups for each category and classified them based on which group they fell into. The features that were provided included High Temperature, Average Temperature, Low Temperature, High Dew Point, Average Dew

Point, Low Dew Point, High Humidity Percentage, Average Humidity Percentage, Low Humidity Percentage, High Sea Level Pressure, Average Sea Level Pressure, Low Sea Level Pressure, High Visibility, Average Visibility, Low Visibility, High Wind Speed, and Average Wind speed. All these categories had continuous features that we changed to categorical features for the specific purpose of the model.

3 Formal Definition

Formal Definition:

Algorithm 1 C4.5 Decision Tree

INPUT: D where D = set of classified instances
OUTPUT: Decision Tree
Split data D into training set (D_r) and test set (D_t), randomly
tree_head $\leftarrow$ node
while D_r contains multiple distinct classes **do**
  for attribute a in D_r **do**
    Calculate Information Gain g for a based on set D_r
  Split D_r into sets, $D_1, D_2, \ldots, D_n$ based on the attribute a associated to the largest value of g where n is the number of distinct values of the attribute
  Add a node to tree_head for each unique value of n where the node contains data on its attribute
  Repeat while with $D_1, D_2, \ldots, D_n$

C4.5 Decision Tree Training Algorithm for Weather Prediction:

```
1: function CALC_BEST_ATT()
2:     data ← the given data frame
3:     classes ← column of decisions from data
4:     attributes ← columns of respective features from data
5:     training_set ← random portion of data according to training percentage
6:     total_decision_entropy ← 0
7:     for decision in classes do
8:         total_decision_entropy ← total_decision_entropy − (prob(decision) · log2 decision)
9:     best attribute ← the given data frame
10:    best information gain ← 0
11:    for att in attributes do
12:        att_values ← unique values of given attribute
13:        att_gain ← dec_entropy
14:        for att_val in att_values do
15:            val_count ← att_val[1]
16:            data_new ← data which matches particular attribute value
17:            decision_values ← data.groupBy(decision)
18:            val_prob ← val_count / total length
```

```
            decision_total ← decision_values.sum()
            dec_entropy_per_attribute ← 0
            for att_val in att_values do
                decision_count ← decision[1]
                prob_per_decision ← decision_count / decision_total
                dec_entropy_per_attribute ← (dec_entropy_per_attribute − prob_per_decision · log_2(prob_per_decision))
            att_gain ← (att_gain − dec_entropy_per_attribute · val_prob)
        if att_gain > best_information_gain then
            best_information_gain ← att_gain
            best_attribute ← att
    return best_attribute

function CREATE_TREE()
    height ← the inputted max height
    train_data ← the inputted training data
    best_att ← calc_best_att(train_data)
    tree_head ← best_att
    add_node(train_data, best_att, tree_head, 1, height)

function ADD_NODE()
    best_att ← the current best attribute
    curr_node ← present "node" in tree
    curr_height ← present height in tree
    max_height ← inputted max height of tree
    unique_decs ← list of unique decisions
    nexts ← empty list
    splits ← empty list
    if length of unique_decs = 1 then
        nexts.append(attribute values of attribute to split on) ← empty list
    else
        unique_vals ← list of unique attribute values
        unique_dfs ← empty list
        for val in unique_vals do
            unique_dfs.append(data[val]
        for i..., len(unique_vals) do
            unique_df ← unique_dfs[i]
            val ← unique_vals[i]
            new_unique_vals ← unique_df[classes]
            splits.append(val)
            if len(new_unique_vals = 1 then
                nexts.append(new_unique_vals)
            else if curr_height ¡ max_height - 1 then
                new_best_att ← calc_best_att(unique_df)
```

```
61:             nexts.append(new Node(new_best_att)
62:             add_node(unique_df, new_att, new_node, curr_height + 1, max_height
63:         else
64:             max_decs ← unique_df.groupby(classes)[classes]
65:
```

C4.5 Decision Tree Testing Algorithm for Weather Prediction:

```
1: function CHECK_SET()
2:     total_correct ← 0
3:     for index, row in data do
4:         total_correct ← total_correct + start_tree(row)
       return total_correct / len(data)
5:
6: function START_TREE()
7:     row ← particular row to classify return pass_thru_tree(row, tree_head)
8:
9: function PASS_THRU_TREE()
10:    row ← particular row to classify
11:    node ← current node on tree
12:    if type(node) is String then
13:        row_desc ← row[classes]
14:        if node = row_desc then return 1
15:        else return 0
16:    else
17:        curr_att ← node.data
18:        val_in_row ← row[curr_att]
19:        if val_in_row not in node.splits then
20:            next_node_index ← random int in range: len(node.nexts)
21:        else
22:            next_node_index ← splits.index(val_in_row
           return pass_thru_tree(row, nexts[next_node_index])
23:
```

Expected Run-time: $O(m \cdot d^2)$
Expected Space Complexity: $O(2 \cdot m \cdot d^2)$

4 Explanation and Intuition

At its core, the ML Decision Tree algorithm relies on entropy, and by extension, the concept of information gain.

Entropy is essentially defined as the "measure of disorder (or purity)". For a concrete example, we can look at the case of flipping a coin. Say that our random variable X models the act of flipping a coin with Heads or Tails as our outcome. In the case where we have Heads and Tails with 50% chance, our entropy is at it's highest since we have no way to measure what the likely outcome is, as there is no majority. On the other end, if we are closer to either 0% or 100% for either Heads or Tails, we have a far lower Entropy.

Now that we understand Entropy, we can better comprehend Information Gain. Information Gain allows us to measure the reduction of disorder when we have additional information about a particular class (features/independent variables). We essentially subtract the entropy of X given that attribute a exists from the total entropy of X, seeing how much our entropy has reduced from the additional attribute a, i.e. how much disorder has been reduced. The greater this reduction, the more information gained about X from the attribute a.

Now we can apply these concepts to our Decision Tree model. In the training process, our data is continuously split by the attribute with the greatest information gain (or biggest reduction in entropy). Classifying future data points therefore relies on traversing this tree because of the "information gained" by matching the appropriate attribute value. This process continues until a final "decision" (classification) is reached. Thus, we have mapped quantitative and qualitative data to a specific outcome by essentially "making decisions" at each step of the tree, narrowing the pool of classifications until a particular leaf node is reached.

5 Implementation in Python

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from node import Node
from sklearn.metrics import confusion_matrix
import seaborn as sns
import math


class DecisionTree:
    """
    A Decision Tree class that allows for classification of
    data into categories based on features of the data.
```

```
"""

def __init__(self, classes, attributes, data, train=0.70, max_height=10,
             get_best_val_score=False):
    """
    Constructs the actual decision tree model based on the given parameters.

    :param classes: The name of the column in the given data corresponding
                    to classes.
    :param attributes: A list of the names of the column in the given data
                       corresponding to attributes.
    :param data: The data to create the tree as a pandas DataFrame.
    :param train: The percentage of the data to split into the training set.
                  The remaining data is split half/half into validation
                  and testing data. Assumed to be a float.
    :param max_height: The max height of the tree to begin.
                       Assumed to be an integer.
    """

    self._classes = classes
    self._attributes = attributes
    self._max_height = max_height

    is_train = np.random.rand(len(data)) < train
    self._train_df = data[is_train]
    leftover_df = data[~is_train]
    is_val = np.random.rand(len(leftover_df)) < (1 - train)
    self._val_df = leftover_df[is_val]
    self._test_df = leftover_df[~is_val]

    self._test_labels = list(self._test_df[self._classes].unique())

    self._curr_tree_head = Node()

    self._heights = list(range(1, self._max_height + 1))
    self._val_accs = []

    self._true_preds = []
    self._model_preds = []

    if get_best_val_score:
        self.create_best_tree()
    else:
        self.create_tree(max_height)
```

```
def create_best_tree(self):
    """
    Determines the Decision Tree with the best accuracy out of trees with
    different height hyper-parameters.
    """
    best_val_acc = 0
    for i in self._heights:
        self.create_tree(i)
        curr_val_acc = self.get_val_accuracy()
        best_tree_head = Node()
        if type(curr_val_acc) == float:
            self._val_accs.append(curr_val_acc)
            if curr_val_acc > best_val_acc:
                best_tree_head = self._curr_tree_head
    self._curr_tree_head = best_tree_head

def calc_best_att(self, df):
    """
    Determines the current best attribute of the data frame, by calculating the
    information gain on each attribute that we are classifying on.

    :param df: The pandas DataFrame containing the data for which to calculate
               the current attribute that gives the best information gain.
               Assumed to contain a column with name = self._classes.
    :returns best_attribute: the attribute with the most information gain
    """
    classes_counts = df.groupby(self._classes)[self._attributes[0]].count()
    total = len(df)
    dec_entropy = 0
    for row in classes_counts.iteritems():
        prob = float(row[1]) / float(total)
        prob = prob * np.log2(prob)
        dec_entropy -= prob

    best_attribute = self._attributes[0]
    best_information_gain = 0
    for att in self._attributes:
        att_values = df.groupby([att])[self._classes].count()
        att_gain = dec_entropy

        for att_val in att_values.iteritems():
            val_count = att_val[1]
            mask_val = df[att] == att_val[0]
            data_new = df[mask_val]
```

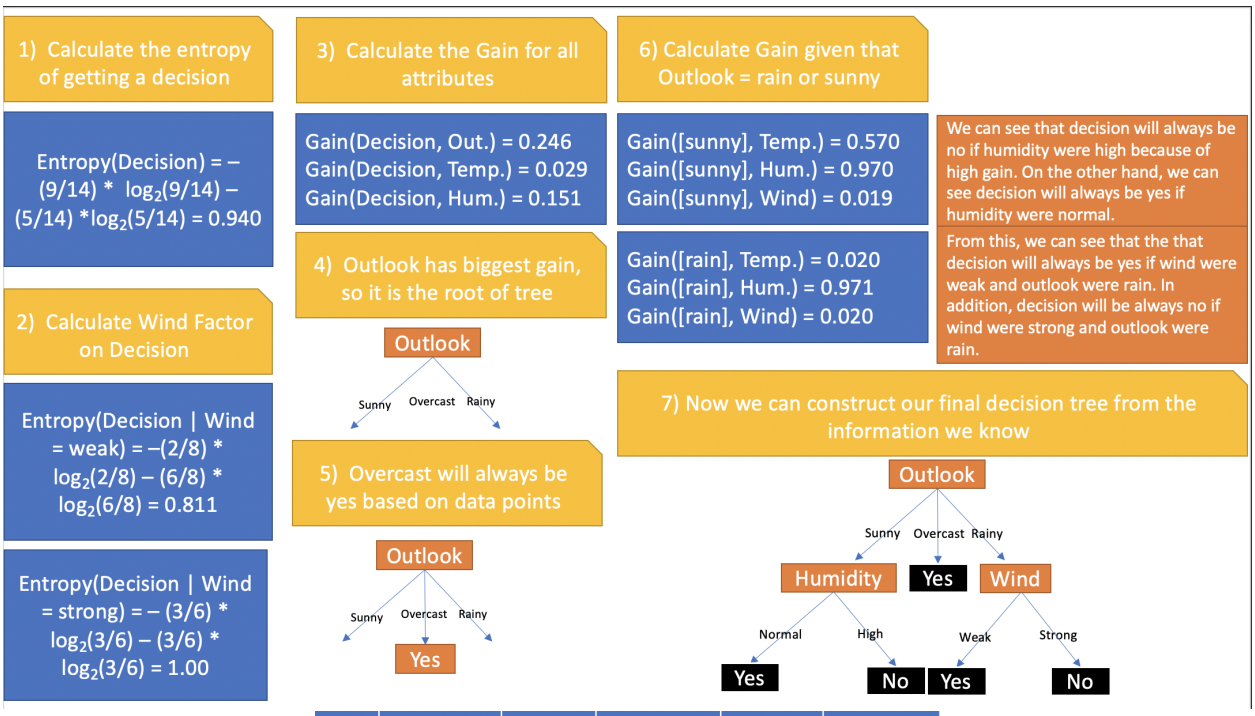
```
            decision_values = data_new.groupby(self._classes)[att].count()
            val_prob = float(val_count) / float(total)
            decision_total = decision_values.sum()
            dec_entropy_per_attribute = 0
            for decision in decision_values.iteritems():
                decision_count = decision[1]
                prob_per_decision = float(decision_count) / float(decision_total)
                dec_entropy_per_attribute -= (prob_per_decision *
                np.log2(prob_per_decision))
            att_gain -= (dec_entropy_per_attribute * val_prob)

        if att_gain > best_information_gain:
            best_information_gain = att_gain
            best_attribute = att
    return best_attribute

def create_tree(self, height):
    """
    Creates the actual Decision Tree based on entropy and information gain.
    """
    best_att = self.calc_best_att(self._train_df)
    self._curr_tree_head.data = best_att
    self.add_node(self._train_df, best_att, self._curr_tree_head, 1, height)

def add_node(self, df, best_att, curr_node, curr_height, max_height):
    """
    Recursively adds nodes to the decision Tree until it reaches the max-height
    or classifies the entire training dataset.

    :param df: the pandas DataFrame that we are classifying.
    :param best_att: the current node's best attribute to split on.
                     Assumed to be a String.
    :param curr_node: the current node of the tree that we're on.
                      Assumed to be a Node.
    :param curr_height: the current height of the overall tree.
                        Assumed to be an int.
    """
    unique_decs = list(df[self._classes].unique())
    curr_node.nexts = []
    curr_node.splits = []
    if len(unique_decs) == 1:
        curr_node.nexts.append(unique_decs[0])
    else:
        unique_vals = list(df[best_att].unique())
        unique_dfs = []
```

```
            for val in unique_vals:
                unique_dfs.append(df[df[best_att] == val])

            for i in range(len(unique_vals)):
                unique_df = unique_dfs[i]
                val = unique_vals[i]
                new_unique_vals = list(unique_df[self._classes].unique())

                curr_node.splits.append(val)
                if len(new_unique_vals) == 1:
                    curr_node.nexts.append(new_unique_vals[0])
                elif curr_height < max_height - 1:
                    new_best_att = self.calc_best_att(unique_df)
                    new_node = Node(new_best_att)
                    curr_node.nexts.append(new_node)
                    self.add_node(unique_df, new_best_att, new_node,
                    curr_height + 1, max_height)
                else:
                    max_decs = unique_df.groupby(self._classes)[self._classes]
                    .count().idxmax()
                    curr_node.nexts.append(max_decs)

    def check_set(self, data, isTest=False):
        """
        Checks the accuracy of the Decision Tree based on the given data.

        :param data: The DataFrame for which to check the accuracy of the model.
        :return a decimal: that represents the accuracy of the model
        """
        if len(data) == 0:
            return "No data to find the accuracy of!"
        else:
            total_correct = 0
            for index, row in data.iterrows():
                total_correct += self.start_tree(row, isTest)
            return total_correct / len(data)

    def start_tree(self, row, isTest):
        """
        Starts the process of passing through the tree and
        checking the given row against the output of the tree.

        :param row: The row for the tree to classify.
        :return method call: calls recursive method that is used to pass
                             through the nodes of the decision tree, while
```

```
                        comparing the given value to the output.
    """
    return self.pass_thru_tree(row, self._curr_tree_head)

def pass_thru_tree(self, row, node, isTest):
    """
    Recursively passes through the tree node by node and
    attempts to classify the given row in the instance's Decision Tree.

    :param row: The row for the tree to classify.
    :param node: the current node of the tree.
    :return recursive method call: in which the node parameter is updated to
                                   represents the next child node in the tree
    """
    if type(node) == str:
        row_desc = row[self._classes]
        if isTest:
            self._model_preds.append(node)
            self._true_preds.append(row_desc)
        if node == row_desc:
            return 1
        else:
            return 0
    else:
        curr_att = node.data
        val_in_row = row[curr_att]
        if val_in_row not in node.splits:
            next_node_index = np.random.randint(len(node.nexts))
        else:
            next_node_index = node.splits.index(val_in_row)
        return self.pass_thru_tree(row, node.nexts[next_node_index])

def get_val_accuracy(self):
    """
    Returns the accuracy of the model on the validation set.

    :return a decimal: that represents the accuracy of the validation set.
    """
    return self.check_set(self._val_df)

def get_train_accuracy(self):
    """
    Returns the accuracy of the model on the training set.

    :return a decimal: that represents the accuracy of the train set.
```

```
        """
        return self.check_set(self._train_df)

    def get_test_accuracy(self):
        """
        Returns the accuracy of the model on the testing set.

        :return a decimal: that represents the accuracy of the test set.
        """
        return self.check_set(self._test_df, True)
```

6 Visual Results and Analysis



Day	Outlook	Temp.	Humidity	Wind	Decision
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
3	Overcast	Hot	High	Weak	Yes
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Strong	Yes
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

Figure 1: Diagram illustrating how the Decision Tree is created

This diagram shows how the Decision Tree is constructed from the given data set.

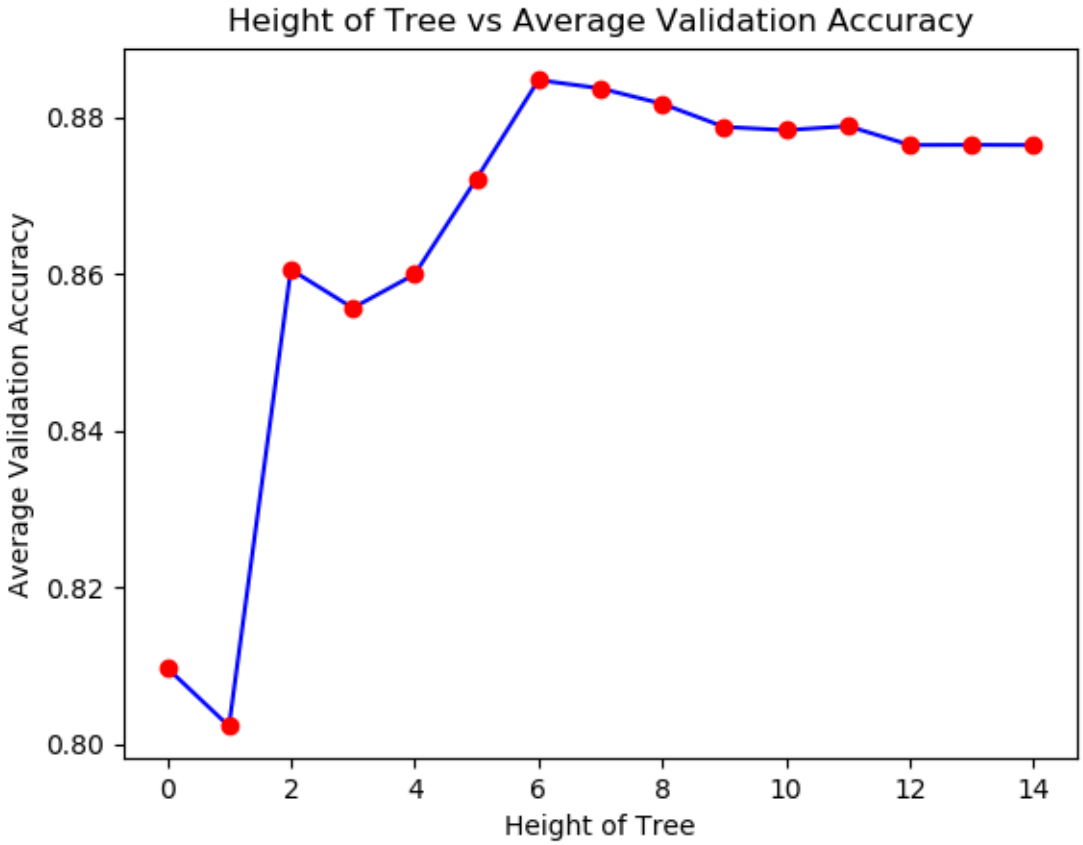


Figure 2: Plot of the height of the Decision Tree versus its validation set accuracy

This plot shows how the height/depth of the decision tree affects its validation accuracy on our provided data. This makes sense because we can see that the accuracy was significantly lower for small heights, only reaching larger numbers of accuracy once the tree's height is larger. This is because the tree needs to actually have height in order for it to make more informed decisions when parsing a new data-point. However, we also notice that the accuracy tends to trend downward after reaching a height of 6. This is because the model begins to suffer from over-fitting as we increase our height, with the tree becoming too heavily fitted to the particular training set and not generalized enough. The reason that there is such a small decline and not a larger one is most likely explained by our particular dataset which contains very, very similar data, so the validation set is more likely to have contents similar to those of the training set, thus allowing for the algorithm to have a larger validation accuracy than expected for larger heights.

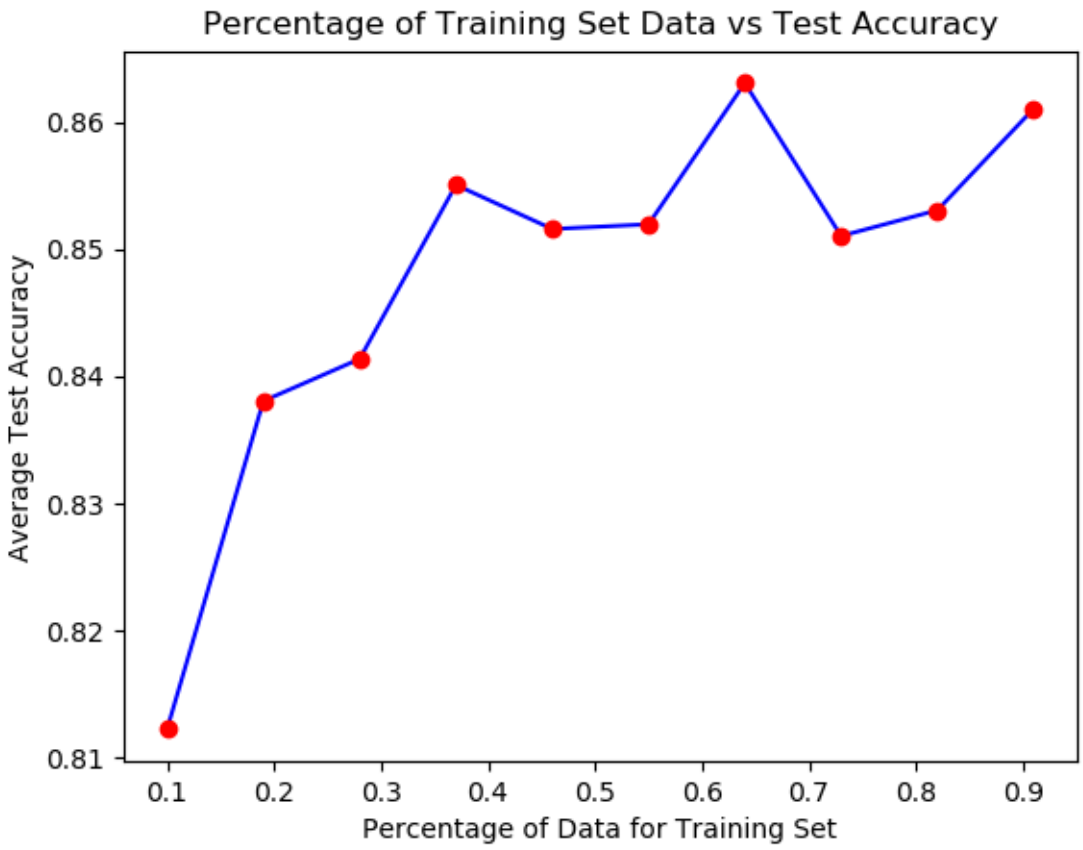


Figure 3: Plot of the training percentage used to make the tree versus its test accuracy

This plot shows how increasing the percentage of the data used for the training set affects the overall accuracy of the data. We can see that for lower training set percentages the program is less accurate. This is because the data that we use to build our tree is smaller, meaning that we have less cases seen and represented within our model. So it makes sense that as we increase the percentage of the dataset that we train on, our testing accuracy similarly increases. What's interesting is that it continues to spike from 0.8 to 0.9. This is most likely in part because of the randomness of the algorithm as well as again our dataset being very similar, as this allows for the testing set to contain more similar content to the training set than expected.

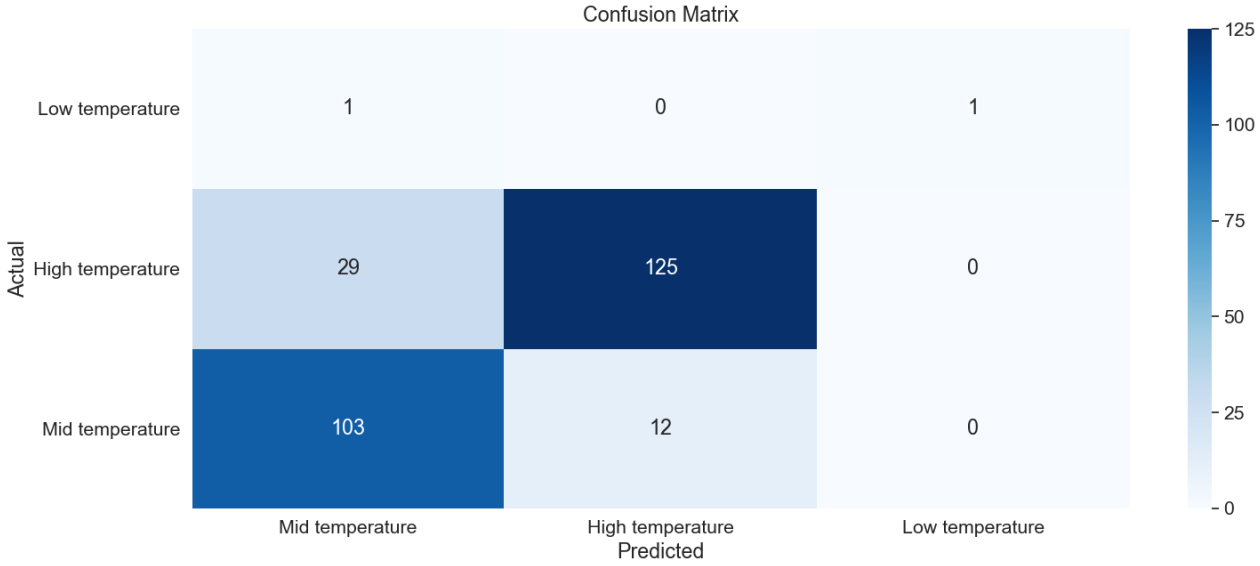


Figure 4: Heatmap of the confusion matrix of a tree of depth 7

This is a Confusion Matrix for a particular instance of our model on our data. A confusion matrix is one such that C_{ij} represents the number of actual elements in i and predicted to be in j. What this means is that we ideally want the diagonal from the bottom left to the top right to contain all the elements of a given row since this means that our model predicted the actual class correctly. We can see that in this case our model generally does predict like this, providing a fairly accurate prediction of what the data actually is. Also, we learned more about the data, noticing that there are very few low temperatures throughout the dataset.

```
High temperature
TempHighF ['High temperature', 'Mid temperature']
Mid temperature
Mid temperature
TempLowF ['Mid temperature', 'Low temperature']
Mid temperature
High temperature
TempLowF ['High temperature', 'Mid temperature']
High temperature
DewPointHighF ['Mid dew points', 'Low dew points', 'High dew points', 'Unknown dew points']
High temperature
```

Figure 5: An inorder printout of a depth 3 tree on our data

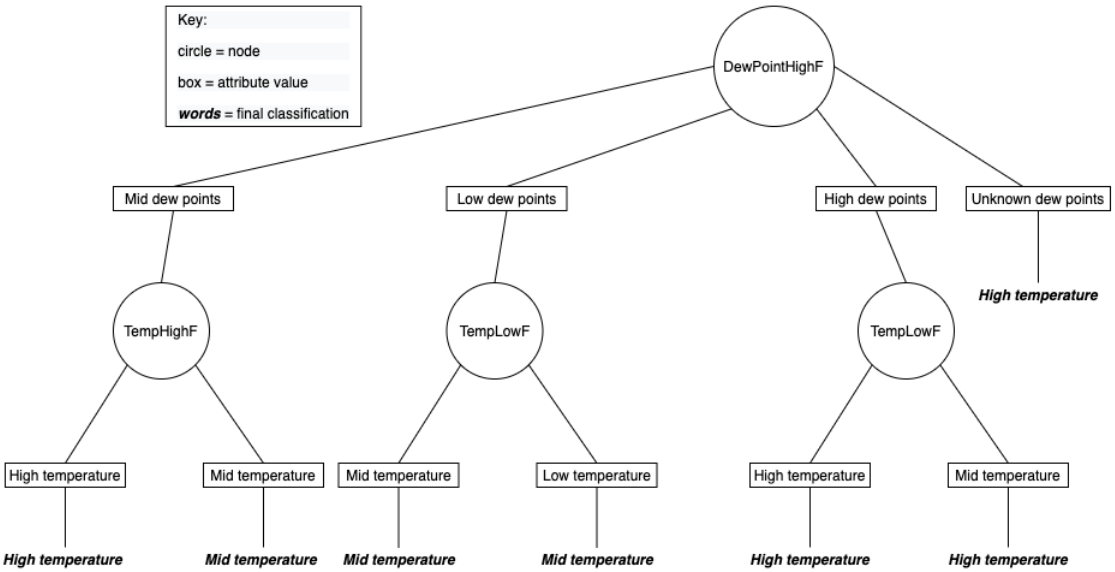


Figure 6: A diagram of the tree corresponding to this printout

7 Statistical Foundation

We split our data set into three different categories: training, test, and validation. The training set made up 70% of the overall data set and, as the name implies, we train our decision tree on this data set. We then test this model on the validation set, which is 15% of the overall data set, to test for the accuracy of the model. At this point, we adjust the hyper-parameter, in our case the height of the decision tree, and re-train our model on the training set. We then test it on the validation set again and record the accuracy value. We continue this process, each time changing the hyper-parameter and recording the effect it has on the accuracy of the model. At the end, we determine the hyper-parameters that we want to use in the final test by looking at the accuracy at a given set of hyper-parameters and logic. For example, the greater the height we have on the decision tree, the more accurate it will be on the validation set; however, after a certain point, we will start to over-fit the data, and will probably not be accurate on a new set of data. After we determine the optimal hyper-parameters, we will then run the model on the test set, which is 15% of our data, and record the final accuracy. The main probability that is used in our Decision Tree is through the processes of Entropy and Information Gain. At each new decision, we calculate the feature that gives us the greatest Information Gain on which class to classify an input into and make a new "decision" node based on that. The formula for Information Gain is $IG(Y, X) = H(Y) - H(Y|X)$, Where Y is the value that we are gaining information on given X and H represents the entropy. In other words, "we subtract the entropy of Y given X from the entropy of just Y to calculate the reduction of uncertainty about Y given an additional piece of information X about Y" (T, Sam 2019). To calculate the Entropy of a given item, we use to following formula: $H(X) = -\sum_{x \in \Omega_X} p_X(x) \cdot \log_2 p_X(x)$. This means, we essentially sum the probabilities of all the elements of in the range of x multiplied by $\log_2$ of the probability of that element. This returns a value between 0 and 1 that measures the entropy of the element or the amount of disorder or purity. So, entropy gives us a mathematical representation of how many different values there are in a particular attribute, giving us a better measure of the "pureness" of that attribute. Based on this, it makes more sense why the information gain formula is what it is. We are trying to gain information on our classes C, so for a particular feature X, $IG(C, X) = H(C) - H(C|X)$. We subtract the purity of C from the purity of C given that X is a certain value, thus returning the amount of new information, or the amount of increased purity or decreased disorder in C when given the feature X. Thus, we then find the best information gain and then split on that attribute, continually splitting on the feature that gives us the most data about our final decisions.

References

T, Sam. "Entropy: How Decision Trees Make Decisions" *Towards Data Science*, 10 Jan. 2019,
https://towardsdatascience.com/entropy-how-decision-trees-make-decisions-2946b9c18c8.
Accessed 16 July 2020.

Saha, Summit. "What is the C4.5 algorithm and how does it work?" *Towards Data*

Science, 20 Aug. 2018,
https://towardsdatascience.com/what-is-the-c4-5-algorithm-and-how-does-it-work-2b971a9e7db0.
Accessed 16 July 2020. (contains pseudocode)
Serengil, Sefik. "A Step by Step ID3 Decision Tree Example" *sefiks.com*, 20 Nov. 2017,
https://sefiks.com/2017/11/20/a-step-by-step-id3-decision-tree-example/

Su, Jiang, and Harry Zhang. A Fast Decision Tree Learning Algorithm. University of New Brunswick, citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.329.3635&rep=rep1&type=pdf.