Chapter 1

Introduction

1.1 Introduction to Machine Learning

Businesses accumulate and analyze large amounts of data to make predictions about the future. This type of analysis, known as predictive data analytics, is used in many industries. Examples include using past data to help a doctor diagnose a patient, and using past data to analyze the amount of claims an insurance policy has paid in order to determine future risk and premiums.

Since analyzing large amounts of data is so prevalent in our society, we are interested in developing an optimal way to analyze these large amounts of data. That is where machine learning comes in. Kelleher et al. (2015) define machine learning as “an automated process that extracts patterns from data”. In other words, machine learning is a way for past data to be interpreted to provide relevant insight for the future.

Machine learning works by searching through data and coming up with a variety of patterns that occur between the descriptive features (features we are analyzing) and the target feature (feature we are looking to gain insight on from the descriptive features). Most of the time, these descriptive features and the target feature are organized in what is known as an Analytics Based Table (ABT). Unfortunately, most of the time, the descriptive features are not all found in one data set, therefore constructing the ABT is often a crucial part in the machine learning process.

<table>
<thead>
<tr>
<th>ID</th>
<th>Loan Amount($)</th>
<th>Salary($)</th>
<th>Occupation</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>250000</td>
<td>90000</td>
<td>Manager</td>
<td>Repay</td>
</tr>
<tr>
<td>2</td>
<td>300000</td>
<td>70000</td>
<td>Analyst</td>
<td>Default</td>
</tr>
<tr>
<td>3</td>
<td>90000</td>
<td>80000</td>
<td>Analyst</td>
<td>Repay</td>
</tr>
<tr>
<td>4</td>
<td>500000</td>
<td>110000</td>
<td>Manager</td>
<td>Repay</td>
</tr>
</tbody>
</table>
This is an example of a potential ABT from a credit agency. As we can see, there are various ways in which the descriptive features are represented. Some are numerical, while others are categorical. Other data types found in ABTs include textual, interval, binary, or ordinal.

The quality of the data is an integral part of creating a successful model. Before descriptive features are placed into the ABT, they are often analyzed alone by measures of central tendency (mean, median, and mode) and variation (variance, percentiles). Once the descriptive features are individually analyzed and outliers, missing values, and other miscellaneous irregularities are accounted for, the descriptive feature is placed in the ABT. After this, relationships between features in the ABT should be analyzed as well.

Multiple models can be constructed from the variety of patterns that occur between the descriptive features. Therefore, machine learning is often denoted as an “ill-posed problem”. This is referring to the idea that it is quite difficult to choose the optimal model from a machine learning program since a variety of models can potentially fit the data well.

Since fitting the data well is common among many different models, it cannot be used to select a given model. Different programmers or companies have their own ideas on how an analysis should go, therefore the way in which a model is selected is by the assumptions or inductive bias that is made by the selector.

1.2 Introduction to Information Based Learning

The general idea behind information-based machine learning algorithms is to determine the descriptive features that carry the most information and chronologically test these features in order to create a decision tree.

Intuitively, this makes sense because to find a credible solution to a problem, we would first start by asking a question that significantly narrows down our options. From there we can continue to ask questions that will further narrow our options and finally we can arrive at a reasonable solution.

Creating a decision tree requires an algorithm, and in this report we will use the ID3 algorithm. We will discuss the ID3 algorithm in more detail later, however, the goal of the algorithm is to create the shallowest decision tree through a process of testing the descriptive features and choosing the descriptive feature that carries the most information at each stage in the process.

We would prefer shallower trees because this will allow our program to flow nicely and obtain our solution fast and effectively. In order to produce shallow trees we need to break down our descriptive features and use those features that are the most informative. But how
do we break down our descriptive features? We use what are known as impurity measures. The most common impurity measures that are found in information-based learning are entropy and gini index.

As stated, the goal of the impurity measure is to break down the descriptive features so that we can learn some information about them. But once we have all of this information about the descriptive features from the impurity measures, what do we do with it? We need some way of selecting the feature that holds the most information. We accomplish this task by using what is known as the selection criteria.

The selection criteria allows us to choose the descriptive feature with the most information based on the impurity measure. The most common selection criteria are information gain and information gain ratio.

Once we use our selection criteria to select the first descriptive feature, the first part of the decision tree is created, and the process starts over again, partitioning the data from this chosen descriptive feature. This process continues until the decision tree is completed.

Here is an example of a simple decision tree. The goal is to predict if a policyholder will file a claim within the year given how old he or she is and if he or she was in an accident the prior year.

**Prediction of a policyholder filing a claim**

1.3 Entropy

In the majority of this report, we will focus our attention on the impurity measure entropy and the selection criteria information gain. In general, entropy is the amount of uncertainty within the result of an action or event. For example, if you knew for sure (probability = 1)
that an event would occur, then that event would have zero uncertainty or zero entropy. On the other hand, if you knew that the probability of an event occurring was very small (the probability is close to 0), then you would be very uncertain of the occurrence of that event. Thus, we would say that that event would have a very large entropy. With this intuition, we see that events with high probabilities should have relatively low entropies and events with low probabilities should have relatively high entropies. Therefore our goal is to map high probabilities to low entropies and map low probabilities to high entropies. In order to map probability to entropy, we can use the negative log function since a low input will result in a large output and a high input will result in a low output. This is the intuition behind Claude Shannon’s entropy model.

Claude Shannon developed a model which states that the entropy of a set is the weighted average of the entropies of each outcome within the set. This is analogous to the expected value calculation which we are familiar with from probability theory and statistics. Mathematically,

\[ H(t) = -\sum_{i=1}^{l} (P(t = i) \times \log_s(P(t = i))). \]

Source (Kelleher et al. 2015)

In the formula above, \( P(t = i) \) is a probability measure representing the probability that \( t \) is equal to \( i \). Also, notice that the log function has base \( s \). In practice the log base 2 is most commonly used to calculate entropies.

Shannon’s entropy model is known as an impurity measure, meaning that entropy is a mathematical model that returns the heterogeneity within a set. There are several other impurity measures found in the computer science literature, one of which we will discuss in detail later.

Kelleher et al. (2015) work through excellent examples to build the intuition behind entropy, information gain, and decision trees. We will analyze some of these examples and supplement them with the code that we created.

First, we would like to consider a deck of cards. Our intuition tells us that if we were to choose 1 card randomly out of the deck (a deck has 52 cards), the probability of choosing the correct card is 1/52. This is a relatively low probability; we are very uncertain that we will pick the correct card. Therefore we would expect the entropy of this event to be relatively high. In order to use Shannon’s entropy model, we need to understand that we are finding the entropy of choosing any card randomly out of the deck. Therefore since there are 52 cards to choose from, we have to find the weighted average of all 52 cards’ entropies in the set. We present
our code below:

```python
from __future__ import division
import numpy as np
import math

card = np.zeros(52)
for i in range(52):
    card[i] = 1/52

def entropy(prob):
    sum = 0
    for i in range((prob.size)):
        sum = sum + prob[i]*(-math.log(prob[i]+1.e-20)/math.log(2))
    return (-sum)

print(entropy(card))
```

output:

```
5.70043971814
```

The idea behind our code is that we first create an array of 52 zeroes. Next for each element in the array we assign the value $\frac{1}{52}$ because that is the associated probability of choosing the correct card from the deck. Next we move onto our entropy function. The function takes an input named “prob” which is an array of probabilities. The function begins by creating a variable named sum and initializing it to 0. Next we use a for loop to iterate over all the elements in the array called “prob”. With each iteration, we are adding the entropy of the particular event to sum. Then after iterating through all the events within the set, we return (-sum) to account for the fact that the entropy equation has a negative on the outside due to the negative log component. Note that we have a +1.e-20 term inside the function. This is used to prevent a domain error in the case of prob[i]=0, which results in log(0), an undefined term.

If we run the code we see that the returned output is 5.7. Keep this value in mind as we move onto the next example. Now we are still going to choose a card randomly out of the 52 but we are only interested in the suit of that card. The probability of choosing the correct suit is $\frac{1}{4}$ since there are 4 suits in a deck of cards. This is a higher probability than choosing one exact card, and our intuition tells us then that the entropy of this event should be lower than that of the previous example.
import math

suit = np.zeros(4)
for i in range(4):
    suit[i] = .25

def entropy(prob):
    sum = 0
    for i in range(prob.size):
        sum = sum + prob[i]*math.log(prob[i] + 1.e-20)/math.log(2)
    return (-sum)

print(entropy(suit))

output:

2.0

Coding this example is analogous to coding the previous example. We will start with creating an array of zeros and placing the value ¼ in each element in the array. Then we call the entropy function with this array of probabilities.

Our output is now 2. This is exactly what we expected! When we choose one exact card out of a deck of cards, we are very uncertain that we will choose the correct card. However, if we are interested in just the suit, we are less uncertain that we will choose the correct suit relative to choosing the correct card. This is why we obtain a value of 2 from our calculation of finding the suits entropy and a value of 5.7 when analyzing the entropy of choosing a specific card.

It is important to note that we are not interested in the exact values of the entropies but rather the relationship between them. We did this example to show that our intuition was correct in reasoning that choosing a card at random has more entropy (5.7) than choosing a suit (2.0). Since we are not interested in the exact value, some would reason that it is strange that we are using such a specific value for our log base (2 in this case). It might make more sense to just use the natural logarithm. Although we understand this critique, we have chosen to use log base 2 in our calculations because it is the most commonly used log base in the literature and we prefer to be consistent with the literature.
1.4 Information Gain

Using the impurity measure of entropy, we can calculate the amount of information gained from testing each descriptive feature.

To test a descriptive feature is to partition the dataset into smaller partition sets based on the different values that are associated with the descriptive feature. For a categorical descriptive feature, a smaller partition set would be created for each category the descriptive feature can attain. For example, if a descriptive feature had three possible categories, low, medium, and high, then three partition sets would be created, one for each category.

Once, we partition the dataset on a descriptive feature and focus on a specific level (category) of that descriptive feature, we can calculate the entropy within that partition. This is referred to as the partition entropy and is calculated in the same way entropies were calculated in the previous section. Continuing the example from the previous paragraph, if we are analyzing a descriptive feature that has categories of low, medium, and high, and we find that the target values vary when we are just looking at the low category, then we would conclude that there is entropy or uncertainty in predicting the target values when partitioning on this specific level. If we assume the same case for the medium and high categories then this will allow us to present the next concept.

The concept of remainder is a significant function inside of the information gain analysis. It is formally presented below and is intuitively the sum of the partition entropies within a descriptive feature, weighted by the amount of instances that level holds within the descriptive feature.

\[ \text{rem}(d, D) = \sum_{i=1}^{l} \frac{D_{d=i}}{D} \times H(t, D_{d=i}). \]

Source (Kelleher et al. 2015).

In the above formula, \(d\) represents the descriptive feature and \(D\) represents the dataset. Note that we are summing from index \(i\) equal to 1 up until \(l\), where \(l\) is referring to the number of levels (or categories) within the descriptive feature. Now, continuing the previous example, we construct the following table.

<table>
<thead>
<tr>
<th>Descriptive Feature Levels</th>
<th>Instances</th>
<th>Partitioned Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>5</td>
<td>1.5</td>
</tr>
<tr>
<td>Medium</td>
<td>10</td>
<td>2.25</td>
</tr>
<tr>
<td>High</td>
<td>7</td>
<td>1.94</td>
</tr>
</tbody>
</table>
There are 22 total instances in partitioning on this descriptive feature, therefore the weighted probability of Low, Medium, and High would be (5/22), (10/22), and (7/22), respectively. Now, according to the equation, the remainder would be calculated as follows:

\[(5/22) \times 1.5 + (10/22) \times 2.25 + (7/22) \times 1.94 = 1.981\]

Intuitively, this means that after we test this descriptive feature, there is still uncertainty left in predicting the target values. This uncertainty (or entropy) has a value of 1.981.

Now that we have an understanding of remainder, we can move on to information gain. In order to calculate information gain, we subtract the entropy of the dataset after testing the descriptive feature (this is referred to as the remainder) from the entropy of the original dataset. This makes sense intuitively, as the entropy of the original dataset is the amount of uncertainty in the original dataset, and the entropy of the dataset after testing the descriptive feature is the amount of uncertainty left after testing a descriptive feature. So the difference in entropies, or the amount of uncertainty eliminated, is the information gained. This process is summarized by the following equation:

\[IG(d, D) = H(t, D) - rem(d, D).\]

Source (Kelleher et al. 2015)

\[IG(d, D)\] is the information gain from testing a descriptive feature, \(d\), for a dataset, \(D\). \(H(t, D)\) is the entropy of the dataset, \(D\), with respect to the target feature, \(t\). Lastly, \(rem(d,D)\) is the remaining entropy after partitioning the original dataset based on the descriptive feature.

Below is an example of python code finding remainder and information gain. Note, \(prob\_type(dataset)\) returns the entropy of a given dataset. We will analyze this function in detail in the vegetation example.

```python
def remainder(totalset, group):
    sum = 0
    for i in range(len(group)):
        sum = sum + (len(group[i]) / len(totalset)) * prob_type(group[i])
    return sum
```

The remainder of entropy left in a dataset after it is split into partitions based on the descriptive feature is a sum of each partitions weighted entropy contribution. We initialize our variable, sum, to 0. Then we iterate through each partition of the group and add the entropy of the group multiplied by the fraction (\(len(group[i])/len(totalset)\)) which represents how much of the entire dataset is represented in each partition. We sum up these weighted
entropies and return the value as remaining entropy.

```python
1. def Info(totalset, group):
   2. return prob_type(totalset) - remainder(totalset, group)
```

For information gain, we subtract remaining entropy from original entropy. Here totalset is referring to the dataset we are analyzing, and group refers to the descriptive feature being tested from that dataset. Therefore, prob_type(totalset) returns the entropy of the original dataset and remainder(totalset, group) calculates the entropy remaining after testing the descriptive feature.

### 1.5 The ID3 Algorithm (Iterative Dichotomizer 3)

Before jumping into the ID3 algorithm, it is advantageous to examine the glossary section of the report in order to understand many of the terms that are found in the literature.

The ID3 algorithm is widely-used in practice. Created by Ross Quinlan, the algorithm’s primary goal is to create shallow decision trees. The first part of the process is to calculate the information gain for each possible descriptive feature. This is done using an impurity measure such as entropy. In an attempt to create the shallowest possible tree, the data set is partitioned based on the descriptive feature that carries the most information gain. The tree grows, with a new path being created for each possible value of the descriptive feature. Along each of these paths, the process repeats itself. The information gain of each remaining descriptive feature is computed, and the descriptive feature with the highest information gain is selected to partition the dataset. This process repeats until a leaf node is created. This happens in one of three ways. The first is the case where all instances in a partition have the same target value. In this case, we say the partition set is pure and we label this leaf node with the corresponding target value. The second case is when there are simply no more descriptive features to test. In this case, a leaf node is labeled with the target value that occurs most along its path. Lastly, if there are no data instances associated with a path, a leaf node is labeled with the target value that occurs most often within the entire dataset.

The following steps generalize the intuition of the algorithm:

1. Calculate the information gain for each descriptive feature
2. Partition the dataset on the descriptive feature with the highest information gain
3. Determine if a leaf node should be created along each path of the descriptive feature
4. Repeat steps (1-3) until there is a leaf node along every path
The ID3 attempts to create the shallowest decision tree possible by choosing the feature with the most information gain at each individual step, however it does not guarantee the shallowest tree. Another issue is that there needs to be some modifications to the algorithm if the data has continuous features. These modifications will be addressed later in this report. Additionally, because of the ID3 algorithm's attempt to organize the dataset into pure leaf nodes, overfitting can occur. By overfitting we mean that the model generated by the algorithm is representing the data very closely. Since the model is representing the data closely, outliers and other undesirable data are being implemented into the model. This could cause our model to inaccurately predict new values. Instead, we would prefer the algorithm create a model that generalizes the data well, so that new data can be predicted accurately. One method to prevent overfitting is to use a pruning method, which we discuss in Chapter 6. Next, we have an in-depth example of the ID3 algorithm being put to use with code and an explanation included.

1.6 Vegetation Dataset

We will now turn our attention to the vegetation dataset presented in Kelleher et al. (2015). Although this dataset is very small and the authors progress through the analysis of it, they do not show any code associated with the calculations in this analysis. Therefore our goal is to supplement the analysis of this dataset with supporting code.

<table>
<thead>
<tr>
<th>ID</th>
<th>Stream</th>
<th>Slope</th>
<th>Elevation</th>
<th>Vegetation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>false</td>
<td>steep</td>
<td>high</td>
<td>chapparal</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
<td>moderate</td>
<td>low</td>
<td>riparian</td>
</tr>
<tr>
<td>3</td>
<td>true</td>
<td>steep</td>
<td>medium</td>
<td>riparian</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
<td>steep</td>
<td>medium</td>
<td>chapparal</td>
</tr>
<tr>
<td>5</td>
<td>false</td>
<td>flat</td>
<td>high</td>
<td>conifer</td>
</tr>
<tr>
<td>6</td>
<td>true</td>
<td>steep</td>
<td>highest</td>
<td>conifer</td>
</tr>
<tr>
<td>7</td>
<td>true</td>
<td>steep</td>
<td>high</td>
<td>chapparal</td>
</tr>
</tbody>
</table>

Before we jump into the code, we should take a few moments to understand the dataset. The intuition behind this dataset is that a researcher is trying to find a relationship between an area's vegetation and the slope, elevation, and presence of a stream. Therefore, in this case we have three descriptive features; stream, slope, and elevation. Stream is a binary variable meaning it can only take two values: true or false. Slope is a categorical variable taking the three values: flat, moderate, and steep. Elevation is another categorical variable taking the values: low, medium, high, and highest. Using these three descriptive features, our goal is to
develop a decision tree that is consistent with the data, and provides a prediction for type of vegetation based on an input of slope, elevation, and stream. Note that the target feature has three values: chapparal, conifer, and riparian.

```python
from __future__ import division
import numpy as np
import csv
import math

def entropy(prob):
    sum = 0
    for i in range((prob.size)):
        sum = sum + prob[i]*(math.log(prob[i] + 1.e-20)/math.log(2))
    return (-sum)

with open('vegetation.csv') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    for row in reader:
        print(row)
    vegi = list(reader)

def prob_type(set):
    count_ch = 0
    count_ri = 0
    count_co = 0
    total = len(set)
    for i in range(total):
        if set[i][4] == "chapparal":
            count_ch = count_ch + 1
        if set[i][4] == "riparian":
            count_ri = count_ri + 1
        if set[i][4] == "conifer":
            count_co = count_co + 1
    prob_ch = count_ch/(total+1.e-20)
    prob_ri = count_ri/(total+1.e-20)
    prob_co = count_co/(total+1.e-20)
    prob_total = np.array([prob_ch, prob_ri, prob_co])
    return entropy(prob_total)

print 'Total entropy is ', prob_type(vegi)
print '
'
```
The first part of our code lists different modules that we import (division, numpy, csv, and math). Refer to Appendix A for Notes on Modules. This allows us to use functions that others have made in our own code, thereby making our process a little easier.

The first function is entropy which we discussed in detail in a previous section. In short, this function will return the entropy of an array of probabilities. The next block of code is the process by which we import the data into Python. We will not go into detail here.

The function named prob_type takes one input, set, which is short for dataset. The goal of this function is to find the probabilities of the target feature associated with the dataset inputted. We find that the first part of the code is initializing three variables to 0 and determining the length of the set. This will allow us to divide each respective feature by the total length of the set in order to find the associated probability of getting that feature. The main block of this function is the for loop, which loops through the dataset, and assigns the number of instances of a specific target value to a variable. For example, if we find that when we loop through our dataset and find 2 instances of “conifer”, the value 2 will be assigned to the variable count_co (which stands for counting the number of instances of conifer). The function then divides each counting variable by the total length of the set to obtain the associated probability of obtaining that feature. The final step is formatting these probabilities in an array so we can input it into our other critical function; entropy.

After developing our main function, we would like to use it to find the total entropy of the dataset. In line 38, this value is calculated as prob_type(vegi) and printed to the screen as “Total entropy is 1.55665670746”.

Now that we have found the total entropy of the dataset we need to analyze the descriptive features and find the information gained from each descriptive feature. In order to do this, we need to analyze each instance within a descriptive feature and see how they relate to the target feature.

```python
true_set = []
false_set = []

for i in range(len(vegi)):
    if vegi[i][1] == "TRUE":
        true_set.append(vegi[i])
    if vegi[i][1] == "FALSE":
        false_set.append(vegi[i])
```
stream = []
stream.append(true_set)
stream.append(false_set)

output:
Partition Entropy of TRUE in stream is 1.5
Partition Entropy of FALSE in stream is 0.918295834054

First we will analyze the descriptive feature stream. Since stream is composed only of true and false, we will begin by making an empty list called true_set and false_set. Next we will use a for loop to append the location of the data set to true_set or false_set if the location holds the correct value. This way we can use the function we created previously by going through the locations in true_set and false_set and counting how many instances of chapparal, riparian, and conifer there are to determine the respective probabilities. Next we return the respective partitioned entropy of true_set and false_set. Our last step in this process is appending the partitioned entropy of true_set and false_set to the new created list named “stream”. This will allow us to later find remainder and information gain.

Before finding remainder and information gain, we use the same process as above to find the partitioned entropies of flat, moderate, and steep in slope and the partitioned entropies of low, medium, high, and highest in elevation.
\begin{verbatim}
print ('\n')
slope = []
slope.append(flat_set)
slope.append(mod_set)
slope.append(steep_set)
low_set = []
med_set = []
high_set = []
highest_set = []

for i in range(len(vegi)):
    if vegi[i][3] == "low":
        low_set.append(vegi[i])
    if vegi[i][3] == "medium":
        med_set.append(vegi[i])
    if vegi[i][3] == "high":
        high_set.append(vegi[i])
    if vegi[i][3] == "highest":
        highest_set.append(vegi[i])

print ('Partition Entropy of flat in slope is', prob_type(low_set))
print ('Partition Entropy of moderate in slope is', prob_type(med_set))
print ('Partition Entropy of steep in slope is', prob_type(steep_set))
print ('Partition Entropy of highest in elevation is', prob_type(highest_set))

print ('\n')
elevation = []
elevation.append(low_set)
elevation.append(med_set)
elevation.append(high_set)
elevation.append(highest_set)

output:

Partition Entropy of flat in slope is 0.0
Partition Entropy of moderate in slope is 0.0
Partition Entropy of steep in slope is 1.37095059445

Partition Entropy of low in elevation is 0.0
Partition Entropy of medium in elevation is 1.0
Partition Entropy of high in elevation is 0.918295834054
\end{verbatim}
Partition Entropy of highest in elevation is 0.0

Our next step is to determine the remainder of each descriptive feature. Remainder is the information that remains required to organize the instances into pure sets after we have split them using the descriptive feature. From our remainder function, we see that it is calculated as the weighted average of the instance multiplied by the instance's partitioned entropy.

```python
def remainder(totalset, group):
    sum = 0
    for i in range(len(group)):
        sum = sum + (len(group[i]) / len(totalset)) * prob_type(group[i])
    return sum

def Info(totalset, group):
    return prob_type(totalset) - remainder(totalset, group)
```

Output:

```
Remainder of stream is 1.25069821459
Remainder of slope is 0.97925042461
Remainder of elevation is 0.679269643166

IG of stream is 0.305958492868
IG of slope is 0.577406282852
IG of elevation is 0.877387064297
```

Finally, we can calculate the information gain of each descriptive feature by subtracting the remainder of the descriptive feature from the total entropy of the data set. By printing the information gain values, we observe that elevation has the highest value and therefore should begin our decision tree. Since elevation is a categorical variable with four instances, it will have four branches coming out of it. To analyze these branches, we look back at our partitioned entropy for elevation. We notice that the partitioned entropy for
low elevation and highest elevation is zero. This means that there is only one target feature associated with this instance. Therefore we can create a terminal node at low elevation, predicting the target feature riparian. And we can create a terminal node at highest elevation, predicting the target feature conifer.

Since the other branches (medium and high) contain positive values of partitioned entropy, we need to use the same process and further partition our dataset so that we can find the next best descriptive feature to choose.

```python
med_true = []
med_false = []
med_flat = []
med_moderate = []
med_steep = []
total_set = elevation[1]

for i in range(len(total_set)):
    if total_set[i][1] == "TRUE":
        med_true.append(total_set[i])
    if total_set[i][1] == "FALSE":
        med_false.append(total_set[i])

print ("Partition Entropy of TRUE at medium level in elevation is ", prob_type(med_true))
print ("Partition Entropy of FALSE at medium level in elevation is ", prob_type(med_false))
print ("\n")

med_stream = []
med_stream.append(med_true)
med_stream.append(med_false)

output:
Partition Entropy of TRUE at medium level in elevation is 0.0
Partition Entropy of FALSE at medium level in elevation is 0.0

Notice that we obtain 0 for both partition entropies after analyzing stream when elevation is medium. This means that all the target features hold the same value along the paths elevation medium -> stream true and elevation medium -> stream false.

```
med_moderate.append(total_set[i])
if total_set[i][2] == "steep":
    med_steep.append(total_set[i])

print ('Partition Entropy of flat at medium level of elevation is', prob_type(med_flat))
print ('Partition Entropy of moderate at medium level of elevation is', prob_type(med_moderate))
print ('Partition Entropy of steep at medium level of elevation is', prob_type(med_steep))
print ('\n')

med_slope = []
med_slope.append(med_flat)
med_slope.append(med_moderate)
med_slope.append(med_steep)

output:
Partition Entropy of flat at medium level of elevation is 0.0
Partition Entropy of moderate at medium level of elevation is 0.0
Partition Entropy of steep at medium level of elevation is 1.0

Notice that we do not get 0 entropy for all instances of slope when analyzing the medium
elevation. When slope is steep, we have entropy of 1 therefore there is some uncertainty of
the vegetation state if we are in the medium elevation and have a steep slope.

med_stream = []
med_stream.append(med_flat)
med_stream.append(med_moderate)
med_stream.append(med_steep)

print ('Remainder of stream in medium elevation is', remainder(total_set, med_stream))
print ('IG of stream in medium elevation is', Info(total_set, med_stream))
print ('\n')

print ('Remainder of slope in medium elevation is', remainder(total_set, med_slope))
print ('IG of slope in medium elevation is', Info(total_set, med_slope))
print ('\n')

output:
Remainder of stream in medium elevation is 0.0
IG of stream in medium elevation is 1.0
Remainder of slope in medium elevation is 1.0
IG of slope in medium elevation is 0.0

After finding partitioned entropy we can find the remainder and information gain in
order to further develop our decision tree. The information gain of stream is greater than the
information gain of slope. This is not a surprise because there was no entropy with either instance (true or false) in stream when we found their partitioned entropy. Therefore stream is chosen for the next branch in the tree. Since the remainder is 0, there is no uncertainty left along this path and two leaf nodes are created; the prediction of riparian is placed along the path elevation medium → stream true, and the prediction of chapparal is placed along the path elevation medium → stream false.

```python
high_true = []
high_false = []
high_flat = []
high_moderate = []
high_steep = []
total_set = elevation[2]

for i in range(len(total_set)):
    if total_set[i][1] == "TRUE":
        high_true.append(total_set[i])
    if total_set[i][1] == "FALSE":
        high_false.append(total_set[i])

print ('Partition Entropy of TRUE at high level in elevation is', prob_type(high_true))
print ('Partition Entropy of FALSE at high level in elevation is', prob_type(high_false))
print ('\n')

high_stream = []
high_stream.append(high_true)
high_stream.append(high_false)

for i in range(len(total_set)):
    if total_set[i][2] == "flat":
        high_flat.append(total_set[i])
    if total_set[i][2] == "moderate":
        high_moderate.append(total_set[i])
    if total_set[i][2] == "steep":
        high_steep.append(total_set[i])

print ('Partition Entropy of flat at high level of elevation is', prob_type(high_flat))
print ('Partition Entropy of moderate at high level of elevation is', prob_type(high_moderate))
```
Now that we have completely analyzed the instance medium in elevation we need to again use the same process to analyze the instance high in elevation to complete the tree. We will not go into detail regarding this process as it is the same as the previous processes. However, there is one crucial difference in this case. Through the analysis we find that there is no entropy remaining after partitioning on slope from high elevation. Therefore slope is chosen as the next descriptive feature to look at if high elevation were found. A leaf node would be placed at each instance of slope since it perfectly separates the data along this path. In other words, when elevation is high and slope is flat there is only one instance of
conifer therefore a leaf node with the prediction conifer is placed along that path. Also, when
elevation is high and slope is steep, there are two instances but both are chapparal therefore
a leaf node with the prediction chapparal is placed along that path. But what about when
elevation is high and slope is moderate? There are no instances of that particular path in our
dataset, however it is a possibility. According to the ID3 algorithm, when a path along the
tree results in an instance that is not present in the dataset, we create a leaf node containing
the most likely value among the target features. In other words, for this dataset, we have no
data on a vegetation site with high elevation and a moderate slope, therefore if we were to
predict this event we would choose chapparal since in our data chapparal has come up three
out of the seven vegetation sites while conifer and riparian have only been found twice out of
the seven vegetation sites.

From the analysis above, we can manually construct a decision tree. The final product
looks like this:

![Decision Tree Diagram]

Now that we have created our decision tree, we can code it in Python. We have presented
a sample decision tree code below.

```python
import numpy as np

def predict(description):
    # User types in incorrect input
        print ("Please type in your responses again using the correct format.")
        return
        print ("Please type in your responses again using the correct format.")
        return
```

Page 21 of 53
print ("Please type in your responses again using the correct format.")
return
if description[0] != "true" and description[0] != "false":
    print ("Please type in your responses again using the correct format.")
return
# Begin Tree
if description[2] == "low":
    print ("riparian")
if description[2] == "medium":
    if description[0] == "true":
        print ("riparian")
    else:
        print ("chaparral")
if description[2] == "high":
    if description[1] == "flat":
        print ("conifer")
    else:
        print ("chaparral")
if description[2] == "highest":
    print ("conifer")

# obtain input from the user

stream = input("Is there a stream (true or false)? ")
slope = input("Describe the slope (flat, moderate or steep)? ")
elevation = input("Describe the elevation (low, medium, high, highest)? ")
description = np.array([stream, slope, elevation])
predict (description)

output:
Is there a stream (true or false)? true
Describe the slope (flat, moderate or steep)? flat
Describe the elevation (low, medium, high, highest)? low riparian

In this code, we can see that we only import the module numpy, since we will be working with arrays. The goal of the function that we create here is to have an input which contains each of the descriptive features (stream, slope, and elevation), and then predict the correct outcome according to our decision tree using conditional statements. It might be more intuitive to explain this code on the later lines (32 - 38) rather than beginning with the
function predict. As we can see beginning on line 32, we assign the variable named stream to a string provided by the user. In other words, the user will type in “true” or “false” when prompted with this question. Furthermore, the next two lines are very similar, the user will simply type in the values he or she is interested in and they will be stored in the respective variable. Next, we will create a new variable called description which is an array, holding all three of the variables we obtained from the user. Then we simply call the function predict with our variable description.

It is important to note that the function will not run correctly if the user incorrectly types the values of the inputs. We have made a simple debugger (lines 4 - 13) which tells the user that he or she has incorrectly typed an input.

Since our decision tree begins with elevation, we will continue our function at that node (line 15). The value of elevation is the third element in the array called description. In Python, array indices begin at 0 instead of 1 therefore the third element in the array called description is found by typing description[2]. We look at this value to see what value it holds in order to progress to the next step. According to our decision tree, if elevation is low, then we predict that the vegetation is riparian. If elevation is medium, then we need to look at what value stream holds. Therefore there is a conditional statement inside the branch where elevation is medium. This condition looks at stream (description[0]) and sees if the value is true or false. If it is true, then our prediction of the vegetation site is riparian. If it is false, then our prediction is chapparal. This process continues for every pathway in the tree.
Chapter 2

Gini Index

Gini index is an impurity measure which can be used interchangeably with entropy. The Gini index considers the proportions of the target values within a dataset. The Gini index asks how often would miscalculation occur if predictions were solely based on these target value proportions. Here, we would expect a target value with a low proportion to be misclassified often, which the Gini index interprets as high impurity. If a target value has high proportion of occurrence within a dataset, we would expect it to be rarely misclassified, the Gini index interprets this as low impurity.

For example, when flipping a coin, there are two outcomes of equal likelihood: heads and tails. The proportions of the target levels in this dataset are 0.5 for heads and 0.5 for tails. The expected rate of miscalculation here would be 0.5, given that a certain outcome occurs. Given a set with three equal likelihood outcomes, the expected rate of miscalculation would be 2/3. Here we see that the target value with relatively lower proportion has a higher rate of miscalculation.

\[
Gini(t, D) = \sum_{l \in \text{levels}(t)} (P(t = l) \times (1 - P(t = l))) = 1 - \sum_{l \in \text{levels}(t)} (P(t = l)^2)
\]

The Gini index is calculated by summating the product of the probability of an instance occurring and the probability of this instance being misclassified. Here, the probability of misclassifying an instance is equivalent to the probability of the target value not occurring. This makes sense intuitively, as we are finding the probability of each individual target feature being misclassified and examining the weighted collection of these probabilities. This equation simplifies to summating the probability of an instance having each target level squared, then subtracting this value from one.

```python
from __future__ import division
```
Here we have a coded example calculating the Gini index for the vegetation dataset. The function gini(prob) accepts an array of probabilities as an input. Next, the sum is initialized to zero. For every probability in the array, the value squared is added to the sum. The gini function then returns 1 minus this sum. Here you can see the Gini index of the vegetation example is approximately 0.6531.

Gini index and entropy are both impurity measures, therefore, their application to information gain and the ID3 algorithm will be the same. Using entropy, recall that information gain was calculated by subtracting the entropy remaining after testing the descriptive feature from the entropy of the dataset. Using gini index, information gain is calculated by subtracting the gini index remaining after testing the descriptive feature from the gini index of the dataset.
Chapter 3

Information Gain Ratio

Despite the power of entropy as an impurity measure, it does have issues when applied to information gain. Information gain is maximized when datasets are partitioned into pure sets. Sets naturally tend to have more purity when they are smaller. Therefore, when entropy is used to calculate information gain in the ID3 algorithm, descriptive features that partition a dataset into many, small partition sets will be preferred. This is an issue because this large information gain may be due to this preference towards small partition sets, not an actual correlation between the descriptive feature and the dataset.

To combat this issue, information gain ratio can be used as the selection criteria.

$$GR(d, D) = \frac{IG(d, D)}{-\sum_{l \in \text{levels}(d)} (P(d = l) \times \log_2(P(d = l)))}$$

Source (Kelleher et al. 2015)

In this case, IG(d,D) is the information gain we have previously calculated. The denominator is an entropy calculation which rectifies the bias in information gain toward features that split the dataset into many small partition sets. Thus it measures the impurity of the dataset when partitioned using the descriptive feature. When there is large impurity, the dataset will be partitioned into many smaller partition sets. In this case, the denominator is larger, and scales down the information gain based on the severity of this impurity. When there is less impurity, the dataset will be partitioned into larger sets. Here, the denominator is smaller, and in turn this yields a relatively large information gain ratio.

Naturally, one may wonder which selection criteria is best to use. Information gain requires less computation than information gain ratio, which can be an advantage for large datasets. Information gain ratio may be preferable when there is large range of values within each descriptive feature. However, there is not a clear-cut winner for all cases.
Chapter 4

Continuous Descriptive Features

It is very likely that at least some of the data we collect is going to be continuous rather than categorical. Therefore, we need to be able to handle this type of feature in our algorithm if we want to develop a useful decision tree. The method for handling continuous features in Kelleher et al. (2015) is to create boundaries within the continuous feature and separate the continuous feature on the boundary points. The separations should occur where there is a change in the target feature.

<table>
<thead>
<tr>
<th>House Price ($)</th>
<th>Consumer Opinion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$200,000.00</td>
<td>Low</td>
</tr>
<tr>
<td>$250,000.00</td>
<td>Low</td>
</tr>
<tr>
<td>$450,000.00</td>
<td>Medium</td>
</tr>
<tr>
<td>$550,000.00</td>
<td>Medium</td>
</tr>
<tr>
<td>$800,000.00</td>
<td>High</td>
</tr>
<tr>
<td>$1,000,000.00</td>
<td>High</td>
</tr>
</tbody>
</table>

For example, in the table above, we have a list of house prices on the left and the consumer’s opinion of how low or high the house price is in the market. Since this is a continuous feature, we would rank the instances from smallest to largest (we have already done so), and we would search to find the change in the target values. There is a change from Low to Medium after the $250,000.00 house, therefore our boundary point would be the average of the two instances surrounding the boundary point: $350,000.00. The next change in target values occurs after the $550,000.00 house. Therefore this boundary point separating Medium and High priced houses is $675,000.00.

The major difference between using categorical features and continuous features comes from their implementation in the decision tree. For continuous features, each boundary is measured individually for information gain. If the information gain (or another selection
criteria) of a boundary is greater than all other boundaries and categorical features, then a
subtree is created. This subtree will only have 2 paths, one greater than the boundary point
and one less than the boundary point. The tree will then continue the analysis of finding the
information gains of the other features and other boundaries partitioned according to the
first selected boundary.

Using the same example above, let’s say the decision tree begins at the low boundary
point ($350,000.00). The decision tree would start by looking if the price is less than or greater
than $350,000.00. If it is less than $350,000.00 then the tree will create a leaf node and return
Low. If it is greater, the tree will have to find a new feature or boundary to test since there is
still entropy left in the dataset. We see that the next step would be to use the other boundary
at $675,000.00. If the instance is less than $675,000.00, then we create a leaf node and return
Medium (remember we are already inside the subtree of price being greater than $350,000.00).
If the price is greater than $675,000.00, then create a leaf node and return High.

This example shows one of the great characteristics of continuous features. With contin-
uous features, one can test the same descriptive feature along the tree. Recall with categorical
descriptive features, once we chose a certain feature, we were done using that feature for the
duration of the tree.
Chapter 5

Continuous Target Features

What happens if our target feature is continuous? If it is, several changes need to be made to the algorithm. First, when we create a leaf node in our tree it should not hold a specific instance of the target feature, since they will all be different. Instead, it could hold the average of the instances of the target feature that have survived to that node. Intuitively, we would want the error of the instances surviving to that node and the average of the instances to be minimized. Consequently, we will need to use a different impurity measure than entropy or gini index. Instead, we will use variance and seek to minimize the variance at each node. The selection criteria in this case is known as the weighted variance and it is found by looking at a descriptive feature and adding up all the variances of the partitions within that descriptive feature and accounting for the relative proportion of data points within that partition. In general, this is proportionally weighting all the variances within a descriptive feature. Then we should choose the descriptive feature with the lowest weighted variance.

<table>
<thead>
<tr>
<th>ID</th>
<th>Sunny</th>
<th>Weekday</th>
<th>Sales ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TRUE</td>
<td>TRUE</td>
<td>1500</td>
</tr>
<tr>
<td>2</td>
<td>FALSE</td>
<td>TRUE</td>
<td>800</td>
</tr>
<tr>
<td>3</td>
<td>FALSE</td>
<td>FALSE</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>TRUE</td>
<td>FALSE</td>
<td>2000</td>
</tr>
<tr>
<td>5</td>
<td>FALSE</td>
<td>TRUE</td>
<td>700</td>
</tr>
<tr>
<td>6</td>
<td>TRUE</td>
<td>TRUE</td>
<td>1700</td>
</tr>
</tbody>
</table>

The table displayed above is an example of the daily sales at small ice cream shop. This data will provide a simple example for us to describe the numerical calculations for the first step in the process to build a decision tree with a continuous target feature. The descriptive features in this case are "Sunny" and "Weekday". "Sunny" is a binary variable describing if a day had sunshine or not. "Weekday" is also a binary variable describing if a day was a
weekday or not. The ice cream shop wants to analyze how much these factors affect their sales. Therefore, the target feature in this case is sales and it represents the revenue that the ice cream shop receives each day.

The algorithm begins the same way as in the previous examples, by analyzing each descriptive feature. However instead of calculating the information gained from testing each descriptive feature, we will be calculating the weighted variance. Recall the formula for sample variance:

$$\text{var}(t, D) = \frac{\sum_{i=1}^{n} (t_i - \bar{t})^2}{n - 1}$$

Source (Kelleher et al. 2015).

Where \( t \) is the target feature, \( \bar{t} \) is the mean of the target feature, \( n \) is the number of values in the dataset \( D \). We will be calculating the variance of each level within each descriptive feature in order to calculate the weighted variance. The weighted variance formula is:

$$\text{weightedvar}(t, D) = \sum_{l \in \text{levels}(d)} \frac{D_{d=l}}{D} \times \text{var}(t, D_{d=l}).$$

Source (Kelleher et al. 2015).

Where again \( t \) is the target feature and \( D \) is the dataset, while \( l \) is the level within each descriptive feature, \( d \). This formula will make more sense after analyzing the following example.

<table>
<thead>
<tr>
<th>Descriptive Feature</th>
<th>Level</th>
<th>ID</th>
<th>Weight</th>
<th>( \text{var}(t, D_{d=l}) )</th>
<th>( \text{weightedvar}(t, D) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunny</td>
<td>TRUE</td>
<td>1, 4, 6</td>
<td>0.5</td>
<td>63,333.333</td>
<td>43,333.333</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>2, 3, 5</td>
<td>0.5</td>
<td>23,333.333</td>
<td></td>
</tr>
<tr>
<td>Weekday</td>
<td>TRUE</td>
<td>1, 2, 5, 6</td>
<td>2/3</td>
<td>249,166.667</td>
<td>332,777.78</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>3, 4</td>
<td>1/3</td>
<td>500,000.000</td>
<td></td>
</tr>
</tbody>
</table>

The table above shows the analysis of testing each descriptive feature in the ice cream shop example. The table is organized into the levels within the descriptive features. For example, in the descriptive feature "Sunny", the level TRUE holds three instances which are ID1, ID4, and ID6. The target feature of ID1, ID4, and ID6 are $1500, $2000, and $1700 respectively. Notice also that the weight associated with this level is 0.5. This is because there are three instances in this level and the weight is calculated by dividing the number of instances within a level of the descriptive feature by the total number of instances that the descriptive feature
holds (3/6 in this example). Next the sample variance of this dataset (1500, 2000, and 1700) is estimated to be 63,333.333. At this point we move onto the other level within the descriptive feature "Sunny", which is the level FALSE. This level also holds three instances which are ID2, ID3, and ID5 ($800, $1000, $700). Again, since there are three instances in this level out of the six total within the descriptive feature, the weight associated with this level is 0.5. The sample variance from this dataset yields 23,333.333. Now we are finally ready to use the weighted variance formula. Using the formula in this example yields the following calculation:

\[(0.5 \times 63,333.333) + (0.5 \times 23,333.333) = 43,333.333\]

The same process can be done to analyze the descriptive feature "Weekday". Note that in the level TRUE, there are four instances instead of three. Therefore, the respective weight of this level is (4/6) or (2/3).

Once the sample variances of each level's dataset have been calculated, the weighted variance can be calculated using the weighted variance formula. Here, the weighted variance associated with the descriptive feature "Weekday" is 332,777.78. Now we have all of the weighted variances we need to progress in the algorithm. The algorithm specifies that we choose the weighted variance with the lowest value, which is in this case choosing the descriptive feature "Sunny". The intuition behind this method is that we are partitioning the dataset on the descriptive feature that reduces the variation in the target values the most.

We will not progress toward analyzing this example any further as it was designed to provide the intuition behind the weighted variance approach to the ID3 algorithm.

When developing a decision tree with continuous target features, overfitting can occur easily if one is not careful. By overfitting, we mean that the model is not generalizing the data well. In other words, if we have many descriptive features, and we find that our partitions result in several groups of one data point, then it is likely that our tree is overfitting the data. To account for this, we may add another condition to the algorithm which states that a partition cannot occur if the resulting dataset is less than x% of the entire dataset. The value of x is up to the programmer or business. This remedy is known as the early stopping criteria.

Therefore there are two main differences in the algorithm when accounting for continuous target features. The first is that there will be no perfect partition when splitting datasets and thus we will need the early stopping criteria. The second is simply the change in the impurity measure and selection criteria to variance and weighted variance.
Chapter 6

Pruning

Tree pruning is a method for eliminating a node that overfits the data and replacing it with the most occurring target level in the dataset or the average target level along that path. Overfitting the data can be due to sampling variance (which we saw in the previous section) and noisy data. Noisy data refers to unreliable data, which can come in various forms depending on the field of research. Outliers and measurement errors are two large components of noisy data (Phillips 2017).

Intuitively we know that as we delve deeper into the tree, the dataset is being split more times and thus will result in a smaller dataset. This means that our chance of overfitting the model increases as our tree gets larger and thus care is needed as we approach the leaf nodes of our tree. As we saw in the previous section, an early stopping criteria is a very useful method for pruning a tree. In the literature, the early stopping criteria would be known as a pre-pruning method since we would determine our stopping criteria before we even create the tree. An example that shows how different pre-pruning strategies affect the decision tree can be found in the first two trees presented in Chapter 8.

One downfall of pre-pruning is that the stopping criteria is arbitrary and in some cases, relevant subtrees may be overlooked since very small datasets are accompanied by them. Because of this slight downfall, some analysts prefer to use post pruning techniques, which means they allow their tree to grow completely, and then they analyze each of the paths along the tree.

A common post-pruning technique is analyzing the prediction accuracy of the tree when it contains a specific subtree and when it does not. This prediction accuracy is based on a part of the dataset that you do not include in the construction of the decision tree. In other words, at first, you set aside a portion of the dataset to be used later; typically it is called the validation dataset. Then after creating your decision tree with the remaining data, you can
test the accuracy of a specific subtree. This is done by using the instances in the validation
dataset, and computing the number of instances that are correct with the subtree compared
to the number of instances that are correct when the subtree is removed. If there are more
correct instances when the subtree is included in the tree (compared to not included) then
the subtree should be included.

In the literature, it is often explained as an error rate, meaning the observer is looking to
compare the number of incorrect instances utilizing the subtree relative to the number of
incorrect instances if that subtree was not used. In this case, we would prefer the option with
the lowest number of incorrect instances. However, this is the same as the above intuition;
either choose the largest number of correct instances or the lowest number of incorrect
instances option.

Using this process, we can go from the bottom of the tree all the way to the root node
of the tree and test whether pruning a section reduces the error of the tree. This method is
known as reduced error pruning.

In general, there are many ways in which pruning helps us. It not only makes our decision
tree more accurate, but it also shortens the tree, thus making it easier to follow.
Chapter 7

Model Ensembles

In practice, it is often beneficial to consider multiple prediction models as opposed to just one. A model ensemble, which is a prediction model made up of several models, is an application of this idea. Our intuition tells us that a model ensemble should be more accurate than a single predictive model. This hypothesis was tested empirically at Cornell University, where two studies found that model ensembles significantly outperformed single predictive models (Caruana et al. 2006; Caruana et al. 2008).

When we create a single predictive model, there will be instances in which our model does not predict the data well. To remedy this, we can create a new model that focuses primarily on these instances that were incorrectly predicted in the first model. Once we create this new model we can add it to the first model; thus, creating an ensemble. However, the ensemble will still have instances that are incorrectly predicted so we can add another model that focuses primarily on predicting these instances correctly. Once we create this model, we add it to the ensemble. This process continues at the business or programmer’s discretion. Typically, the business or programmer will set a number of models in the ensemble and therefore the process will end when the ensemble is composed of this specified number of models. This method of creating an ensemble is known as boosting.

Another ensemble method is called bagging or bootstrap aggregation. Recall from statistics that bootstrapping is a common technique used to estimate a specific statistic (mean, median, and standard deviation are common). If we had a sample with 5,000 values and we were interested in finding the mean, we could calculate the mean of this sample directly. Alternatively we could create independent subsets of this sample, perhaps of 1,000 subsets of 1,000 values each (with replacement) and find the means of each of these subsets and then find the grand mean of these means. This method is very powerful and can reduce the variance of the statistic we are measuring. Note that when creating subset samples, each
subset is created with replacement to ensure that all of the subsets are independent.

This bootstrap method can be applied to decision trees. In this case, instead of finding
the mean of all of the subset means, we are creating independent trees from each subset
and then aggregating them together to create our ensemble. Note that all of the decision
trees created contain the same descriptive features. If we were to randomly assign descriptive
features to a tree in the bootstrapping process and then aggregate the trees, this ensemble
would be known as a random forest.

As we have learned, there many ways to create a single predictive model, and there are
also many ways to create model ensembles. This leads us to reiterate that there are many
viable options when creating decision trees. Some methods will work very well in some cases,
and others will work less efficiently. The business or programmer should consider many of
the available options and examine which models work well for the task at hand.
Chapter 8

Insurance Data

8.1 General Intuition of Real World Applications

The following data was gathered from “Generalized Linear Models for Insurance Data” by Piet de Jong. It represents one-year automobile insurance policies that were purchased in 2004 or 2005.

There are several features to analyze in this dataset. The first is veh_value which denotes the vehicle value in $10,000 terms. The second feature, named exposure, is continuous and represents the risk exposure of the individual from 0 to 1. Next, clm represents the existence of a claim, so this is a binary variable with values 0 or 1; 0 meaning that the policyholder did not file a claim and 1 meaning that the policyholder filed at least one claim. The feature numclaims represents the number of claims filed by the policyholder. This feature takes the values 0, 1, 2, 3 or 4 in this dataset. The feature called claimcst0 represents the total monetary value of the claims that the policyholder filed in the policy year. The type of vehicle that the policyholder drives is denoted as veh_body. This feature can take the values BUS, CONVT, COUPE, HBACK, HDTOP, MCARA, MIBUS, PANVN, RDSTR, SEDAN, STNWG, TRUCK, and UTE. In our code we have elected to use numbers to represent these values. For example, 0 would refer to BUS, 1 to CONVT, and so on until 12 which would refer to UTE.

Next, veh_age represents the age of the vehicle. However, this is not in years but rather in an interval with 1 being the youngest and 4 being the oldest. the gender of the policyholder is also included with M standing for male and F for female (0 for female and 1 for male in our code). The second to last feature is area which describes the general location of the policyholder’s residence. This feature takes 6 values: A, B, C, D, E, F (0, 1, 2, 3, 4, 5 in our code). Finally, the last feature is agecat which denotes the age of the policyholder in an interval from 1 to 6; 1 being the youngest and 6 being the oldest.
Using the package sci-kit learn, we can generate decision trees and display them efficiently. However, several important remarks need to be made about sci-kit learn before we proceed. Sci-kit needs numerical values to perform its analysis. It is possible to have sci-kit learn analyze textual values but that requires somewhat complex methods that we will not discuss. Also, if we choose to represent the values in our descriptive feature as numbers (as we did for veh_body and area), we have consequently turned our descriptive feature into an ordinal variable instead of a categorical variable. An ordinal variable is similar to a categorical variable but in an ordinal variable, there is a clear order for the values. For instance, agecat is an ordinal variable since it is a variable representing a clear order from 1 (youngest) to 6 (oldest). Therefore, representing veh_body as an ordinal variable is not 100% accurate since there is not a clear order to the values. Therefore, in practice, we would not use this method. However, for the purposes of this section, we have elected to progress through our examples in this manner since it enables us to display and interpret the trees nicely. We will show in section 8.2 how to change features into categorical variables so that sci-kit can analyze the data correctly. Lastly, sci-kit learn utilizes binary classification. This is analogous to the ideas presented in Chapter 4, in which each boundary within the descriptive feature is tested for information gain. A subtree is created at the boundary with the highest information gain. This subtree will only have two branches coming from it, either greater than the boundary point, or less than the boundary point. The benefit of binary classification is that descriptive features can be used multiple times along a tree instead of just once. The disadvantage of binary classification is that the trees may become extremely large and complicated if one is not careful to use pruning strategies.

For the first example, we have chosen to make our target feature the number of claims from the policyholder and have our descriptive features be veh_value, exposure, clm, claim-cst0, veh_age, and agecat. It is important to note that the overwhelming majority of the claims from this dataset are 0 (roughly 93%), however we think analyzing number of claims will be a very good place to begin our analysis.

The first decision tree we created is shown below and it is made from 66499 data points from the dataset. We believe it is good practice to remove at least some data points from the data set to be used as the validation set.

We elected to use one pre-pruning strategy in this analysis. Since some of the descriptive features are continuous and there is a relatively large amount of data, our pre-pruning strategy prevents the decision tree from continuing if there are less than 1000 data points in the remaining subtree. This allows our tree to not over-fit the data.
First, we will explain how to read the tree correctly. The first line in each testing node represents the feature that is being tested. For example, at the root node, the feature \texttt{claimcst0} is being tested to see if the value is less than or equal to $100.00$. The next line shows the entropy of this feature. The following line shows the amount of data points in the set that is being tested. And finally, the last line shows the values of the data points within the set. So, at the root node, there are 61982 instances of 0 claims, 4235 instances of 1 claim, 263 instances of 2 claims, and 17 instances of 3 claims, and 2 instances of 4 claims.

There are several interesting conclusions that can be drawn from this tree. First we notice that the tree begins by testing if the total claim amount is less than or greater than $100.00$. If it is less than $100.00$, then the tree predicts that the number of claims is 0 since all of the resulting instances are in the first column of values (value = [61982, 0, 0, 0, 0]). If the total claim amount is greater than $100.00$, then we continue along the tree and test if the total claim amount is below $399.77$. Therefore if the total claim amount is in the interval ( $100.00$, $399.77$), then we predict that the number of claims is 1 since all of the resulting instances are in the second column of values (value = [0, 1542, 0, 0, 0]). If the total claim amount is greater than $399.77$, then we need to test another feature. We find that risk exposure has
the most information gained, so we test this feature on being less than or equal to 0.5188. If exposure is less than 0.5188, then we will still predict the number of claims to be 1 since the majority of instances in the resulting partition have 1 claim. Even if exposure is greater than 0.5188, the tree will predict the number of claims is 1 since 1 is found the majority of the time within this partition.

Although this tree has some interesting qualities, it may not be exactly what a business is looking for since it is only returning 2 values: predicted number of claims is either 0 or 1. This is due to the overwhelming large number of values of 0 and 1 in the dataset. Nonetheless, we would like to see if we can improve our model.

Now, we have changed our pre-pruning strategy. Our pre-pruning strategy prevents the decision tree from continuing if there are less than 500 data points in the remaining subtree. The resulting subtree is created:

In this case, our tree becomes longer and more complicated. However if we look closely, we notice that the tree will actually predict the same outcome as before, since the first two paths are the same and the other paths still have number of claims equal to 1 being the
majority target level within the partition. Therefore, this tree does not provide any additional information.

Obviously, an insurance company would not use these trees to predict outcomes because the target feature (number of claims) is perfectly correlated with some of the descriptive features (existence of a claim and total claim amount). However, this exercise was designed to show how to interpret the decision tree and how a different pre-pruning strategy may affect the tree. Now we can turn to more realistic applications of using decision trees for prediction.

Here we create a decision tree with the target feature of the existence of a claim (clm with value either 0 or 1) and all descriptive features excluding exposure, number of claims, and total claim amount. Notice our pre-pruning strategy prevents the decision tree from continuing if there are less than 5000 data points in the remaining subtree. Unfortunately, this decision tree is not informative, because it labels every leaf node as no claim (or clm with value of 0). We recognize this because 93.2% of all instances in this dataset have no claim. This overwhelming majority of no claim data points drown out any interesting results we may salvage from this tree.

This example shows that although we may have an intuitive appeal to solve a problem, our solution may not be what we expected. Machine learning can be a trial and error process in many cases. Therefore, we may have to develop creative solutions to solve our problems.
After analyzing the number of claims and the existence of a claim as target features, we may be interested in analyzing the total claim amount as the target feature. One way to do this is to first analyze the portion of the data that has at least one claim. Gathering this data yields 4517 data points; there are 4517 instances of a given policyholder having at least one claim. If we were to develop a tree at this point, our tree would be too complex to show since the target feature is a continuous variable (total claim amount) and the tree would be split at every boundary point within the feature. Therefore, we decided to rank the values and place them into roughly equal size bins. We chose to have four bins which means each bin will hold about 1130 values. The first bin will hold relatively low values of claim amounts compared to the fourth bin. Also, our pre-pruning strategy prevents the decision tree from continuing if there are less than 350 data points in the remaining subtree.

Interpreting the values at the root node of the tree we see that we have 1131 values in the first bin, 1130 in the second bin, 1129 in the third bin, and 1127 in the fourth bin. This can be interpreted as the first bin holding 1131 relatively low claim amounts, the second bin holding 1130 medium to low claim amounts, the third bin holding 1129 medium to high claim amounts, and the fourth bin holding 1127 high claim amounts.
Now that we understand the target feature, we utilize the following descriptive features: veh_value, veh_body, veh_age, gender, area, and agecat. Starting at the root node of the tree, if the area that the policyholder lives in is either 0, 1, or 2 (A, B, or C), and the age category of the policyholder is 1, 2, or 3, then the decision tree will predict that if a claim occurs, it will be in the highest claim amount interval. This is because if we look at the value list, 126 is larger than any of the other values and since 126 is the fourth value, it represents the highest bin category. Intuitively, this branch of the tree makes sense because the age category of the individual is 1, 2, or 3 which means these are the youngest and least experienced drivers on the road. The tree is predicting that if a policyholder lives in a certain area and is relatively young, that policyholder is likely to have a large claim amount if a claim occurs for that individual.

A similar analysis can be done for the remaining leaf nodes, however since there are 10 leaf nodes we will not analyze each one directly. We will note though, that the there is at least one path in which each bin is chosen. More specifically, the tree will predict bin 1 along 5 paths, bin 2 along 3 paths, and bin 3 and bin 4 along 1 path each. Since there is some variation in the predictions, this tree may be a candidate to progress toward a model ensemble. Although we will not carry out this process, utilizing one of the methods explained in the previous section (boosting, bagging, and random forests), one may be able to create a fairly accurate predictor of relative claim size.
In this tree, we used the same data and process as the previous example, however we used the impurity measure gini index instead of entropy. It is interesting to note that in this case we obtained the exact same tree. However, gini index trees and entropy trees will not always result in the same tree. Since they are different mathematical measures, they could produce different results. Once again, since we obtained variation in our target values, we believe that using a model ensemble method would be a very good option in order to enhance our predictability.

8.2 Prediction

The previous section focused on the general intuition behind creating decision trees with real world data and interpreting the results of the tree. However, in practice, businesses are interested in how accurate the tree is at predicting the target feature. Therefore, in this section we will focus on the actual results of a decision tree that could be used in practice.

Our prediction example will have the target feature numclaims and the descriptive features will be veh_value, veh_body, veh_age, gender, area, and agecat. The entire dataset is used for this example, 70 % of which will be used as the training dataset and 30 % will
be used as the validation (or testing) dataset. We allow the tree to go to completion in this example with no pre-pruning method because we know that there are very few instances of high numbers of claims and we would be interested in discovering whether the model predicts these claims accurately. For example, there are only 2 instances of a person having 4 claims and only 18 instances of a person having 3 claims in the entire dataset.

```python
import pandas as pd
import numpy as np
from sklearn import tree
from sklearn.cross_validation import train_test_split
auto = pd.read_csv('car.csv')
```

The first block of code presented shows the modules that we will use for the duration of the analysis. The module pandas is used here to allow us to import the csv file into Python. It will also be used later to create column names in order to allow us to reference specific data. Next, from scikit learn (sklearn), we import load_iris, tree, and train_test_split in order to create a decision tree from the data. We will go into more detail about these functions when they are used later in the report. Lastly, after importing the various modules, we import the dataset on line 5.

```python
auto_data = pd.DataFrame(auto, columns=['veh_value', 'veh_age', 'veh_body', 'gender', 'area', 'agecat'])
data = pd.get_dummies(auto_data, columns=['veh_body', 'gender', 'area'], drop_first=True)
target = pd.DataFrame(auto, columns=['numclaims'])
auto_target = target.as_matrix()
auto_target = auto_target.flatten()
auto_target = auto_target.astype(int)
```

Line 1 represents the code used for choosing the descriptive features we want. We are essentially choosing the data from columns 'veh_value', 'veh_age', 'veh_body', 'gender', 'area', and 'agecat' from the dataframe auto, and assigning this data to the new dataframe created which we named auto_data. Line 2 represents the method that is used to enable us to use categorical features correctly in scikit learn. First we should think about the features in our new dataframe auto_data. veh_value is a continuous feature which sci-kit learn can handle already. veh_age and agecat are ordinal features since the greater the vehicle's or person's age, the greater the value of the feature. However, veh_body, gender, and area are all categorical features, therefore we need to correctly input them as categorical features into python. In order to do this we can use pd.get_dummies which is a function in pandas that takes a specific column from a dataframe and makes a dummy variable for each value within
the column. Therefore in line 2 above, we are creating dummy variables for all of the values within the features "veh_body", "gender", and "area". Note that there is a parameter called drop_first which we have assigned the value "True". This is telling python to delete the first dummy variable in each feature since we will be double counting if we do not. Recall from statistics that if we have a categorical variable with k values, it can be represented by k-1 dummy variables. This is essentially what is happening on this line.

Next, on line 23, we assign the column named 'numclaims' as 'target' since this will be our target feature for this example. Lines 24 through 26 show the code for how we take the data from the column in the dataset and form a matrix with integer values. We then write auto_target on line 27 to print the matrix to check if we have done everything correctly. The output is as follows.

```
array([[0, 0, 0, ..., 0, 1, 0])
```

This is the array of numclaims from the dataset. Notice that it will not print the entire array as it is around 67,000 values long. However we see that it is of the format that we want as this will allow scikit learn to run properly.

Now that we have our target feature as an array, we would like our descriptive feature to be presented as an array as well. The follow code is used to complete this task and then we will print out our array to make sure that we completed this task correctly.

```
data=np.array(data)
data
```

Below, we can see that our output is in the correct format.

output:

```
array([[ 1.06 , 3.  , 2.  , ..., 0.  , 0.  , 0.  ],
       [ 1.03 , 2.  , 4.  , ..., 0.  , 0.  , 0.  ],
       [ 3.26 , 2.  , 2.  , ..., 0.  , 1.  , 0.  ],
       ..., 
       [ 1.93 , 4.  , 3.  , ..., 0.  , 0.  , 0.  ],
       [ 0.97 , 3.  , 2.  , ..., 0.  , 0.  , 0.  ],
       [ 1.02 , 3.  , 1.  , ..., 0.  , 0.  , 0.  ]])
```

We notice that the first column is the values from veh_value. The following column represents veh_age, followed by agecat. The columns after agecat represent the dummy variables that were created on page 44. Since veh_body has 11 dummy variables, gender has 1 dummy variable, and area has 5 dummy variables, python chooses not to list all of them and instead just lists the last three dummy variables after the ellipsis ("...").

Now that we have our descriptive features and target features in the correct format, our
next step is to split the data into a training set and a testing set. The training set will be the
dataset that will be used in order to build our decision tree. Then after creating the decision
tree, we will test it’s accuracy by using the testing set. In order to do this, we can use the
train_test_split function that we imported at the beginning of this section. This function
allows us to specify which percentages of the dataset to use as the training set and the testing
set. We now present the following code:

```python
X_train, X_test, y_train, y_test = train_test_split(
    data, auto_target, test_size=0.30, random_state=11)
```

X_train represents the descriptive feature’s values in the training set while X_test rep-
resents the descriptive feature’s values in the testing set. The variable y_train represents
the target feature values in the training set and y_test represents the target feature values
in the testing set. Recall that auto_data holds all of the descriptive feature's values in the
dataset and auto_target holds all of the target feature's values in the dataset. Therefore, this
function is splitting up auto_data into X_train and X_test with 30 % of the values going to
X_test (this is what test_size = 30 is used for), and 70 % of the values going to X_train. Similarly,
The corresponding target values for X_test are stored in y_test and the corresponding target
values for X_train are stored in y_train. Note that the data is split up randomly so if we are
interested in saving this particular trial we can choose an arbitrary number and assign it to
the random_state input. This allows us to be able to regenerate the same random numbers if
we want to use our code at a later date.

Now that we have split our data correctly, we can use scikit learn to create our decision
tree. Since the creation of the ID3 algorithm in 1986, several modifications have taken place.
The most recent revision to the algorithm elects to create a binary tree, which still partitions
the data based upon the highest information gain, however after a descriptive feature is
chosen at a node, it can still be chosen at future nodes along the tree. This is analogous to the
modifications specified in Chapter 4 (Continuous Descriptive Features). This modification to
the ID3 algorithm is known as the CART algorithm and is the algorithm used in scikit learn.

```python
clf = tree.DecisionTreeClassifier(criterion = "entropy", min_samples_leaf = 1)
clf = clf.fit(X_train, y_train)
```

This block of code shows how to create a decision tree in scikit learn. There are several
inputs that can be specified however, we have elected to only specify that we want the
impurity measure to be "entropy" and we have chosen to not pre-prune this tree since we are
interested to see if the tree will predict high values of number of claims accurately. As you can
see, we list our specifications and assign this tree design to clf. Then we can fit our training
set (X_train, and y_train) to the tree design we specified.
Once we have fit our training set we would like to predict how accurate our tree is. Therefore, examining the code shown below we use `clf.predict(X_test)` to predict the values of the testing set and assign these values to `y_pred`. Once we have our predictions of the testing set, we can compare them to the actual values in the testing set to find our prediction accuracy. Using the function `accuracy_score` from scikit learn, we can determine how accurate our model is.

```python
y_pred = clf.predict(X_test)
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
```

Output:
```
0.88402023873851743
```

We can see that the model predicted the testing set with about 88.40 % accuracy. This value is calculated as the number of instances correct (`y_test` and `y_pred` match) divided by the total number of instances in the testing dataset. Once we have this percentage, we can analyze it further by means of the confusion matrix. The confusion matrix is a very useful tool that is commonly used in predictive analytics. It displays the true values of the validation set as the rows and the predicted values by the tree as the columns. The confusion matrix code and output is displayed below.

```python
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, y_pred)
```

Output:
```
array([[17925,  957,   66,   1,   0],
       [ 1238,   70,   7,   0,   0],
       [   81,    4,   1,   0,   0],
       [    5,    1,   0,   0,   0],
       [    1,    0,   0,   0,   0]])
```

Here is a table that we created that depicts the confusion matrix a little more clearly:

<table>
<thead>
<tr>
<th>Actual Number of Claims</th>
<th>Predicted Number of Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>17925</td>
<td>957 66 1 0</td>
</tr>
<tr>
<td>1238</td>
<td>70 7 0 0</td>
</tr>
<tr>
<td>81</td>
<td>4 1 0 0</td>
</tr>
<tr>
<td>5</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>
If we were to interpret the confusion matrix, we would start by analyzing the top left number. This tells us that there were 17925 instances of actual numclaims = 0 that we predicted as numclaims = 0 (our predictions were correct). Moving along to the right, we see that there were 957 instances of actual numclaims = 0 that we predicted to be the numclaims = 1 (our predictions were incorrect). Furthermore, there were 66 instances of numclaims = 0 that we incorrectly predicted as numclaims = 2, 1 instance of numclaims = 0 that we incorrectly predicted as numclaims = 3, and 0 instances of numclaims = 0 that we incorrectly predicted as numclaims = 4. This process continues for the remaining rows. For one last interpretation, the number in the second row and first column represents that there were 1238 instances of numclaims = 1 that we incorrectly predicted as numclaims = 0.

In general, the values along the main diagonal of the matrix represent the instances that we predicted correctly. The overall accuracy of this predictive model was 88.3529 %. This value can also be confirmed via the confusion matrix. The accuracy is calculated by summing the number of instances predicted correctly and dividing that sum by the total number of instances in the testing data set. For example, using the confusion matrix, we can do the following calculation.

\[
\frac{(17925 + 70 + 1)}{(17925 + 957 + 66 + 1 + 1238 + 70 + 81 + 4 + 1 + 5 + 1 + 1)} = 0.884020
\]

This confirms the prediction accuracy value that we obtained from accuracy_score. After confirming the prediction accuracy we can analyze the accuracy. We notice that the predictive model is reasonably accurate at predicting the value 0 for number of claims, however it is not as accurate when predicting the other values. Recall that approximately 93 % of the dataset contains no claims. This certainly could be a valid reason for this occurrence. Also, recall that we did not pre-prune this tree because we were interested to see if the model would predict any of the larger valued number of claims correctly. Unfortunately, this was not the case, therefore we may actually be able to increase our overall accuracy by inputting a pre-pruning condition.

Although pre-pruning and post pruning methods may result in a better model, the most effective way to create a more accurate model would be to use model ensemble methods. Using boosting, bagging, or random forests are all viable options for this tree. Although we will not discuss the model ensemble methods for this tree, this would be the next step in the process to develop a very accurate model.
Chapter 9

Conclusion

Machine learning is a growing field in the data analytics industry. Many companies are using machine learning to solve complex problems in order to grow their business and progress technology.

This report focused primarily on the basic intuition behind information based machine learning. We explained the intuition behind the impurity measures entropy and gini index, along with the selection criteria information gain and information gain ratio. Once we built this foundation we could apply it to the ID3 algorithm. The ID3 algorithm is a commonly used algorithm that has been studied extensively over the past several years. Throughout this process, new algorithms have emerged including the C4.5 and the CART algorithm.

After understanding the algorithm, we could extend our analysis to more sophisticated topics in the field such as pruning and model ensembles. Model ensembles are used a great deal in practice because of their ability to predict outcomes more accurately than individual prediction models.

Our final analysis discussed the implications of everything we learned and how they could relate to real world examples.
Appendix A

Notes on Modules

The module csv allows us to import a csv datafile in Python. We have chosen to store our data in this format, therefore we need this module in order to input the data into Python.

The module division allows us to use the division operator but also allows us to carry a large amount of significant digits. Since we are comparing numerical values several times in the report, having a large number of significant digits is critical.

The module math allows us to use the logarithmic function. Importing this module gives us an easy access to use the logarithmic function since it is used heavily in entropy calculations.

The module numpy allows us to work with arrays in Python. This is important as it allows the computation to be efficient.

The module pandas allows us to primarily create column names in our dataset and reference these named columns. The module also can be used to import datasets into Python.

The module sklearn (scikit learn) allows us to create decision trees from our datasets and use metrics to analyze the decision trees we create.
References


Glossary

Branch (or edge): A branch connects two nodes by answering a query.

Descriptive Feature: A characteristic that we use to learn more about the target feature.

Leaf node: A node in which there are no further branches or nodes coming from it.

Level: Synonymous with value. Target level is equivalent to target value.

Node: A section of the tree where a query or prediction is being made.

Partition: Once a branch has connected a node, the data is split by analyzing the values in the data that have the quality answered by the query.

Partitioned entropy: The entropy calculated after partitioning the dataset on only one value of the descriptive feature. The entropy associated with an individual category after partitioning a descriptive feature.

Path: A path begins at the root node and takes a specific route to a leaf node. There are many different paths in a tree.

Pure set (or pure leaf node): a data set that contains only one value of the target feature.

Query: A question about the information contained in the data.

Root node: This is the very top of the tree and denotes the first query.

Target Feature: The characteristic that we are interested in learning more about. We plan to learn enough about this feature so that we can make adequate predictions about it in the future.

Target value: A value that the target feature can take.

Testing a descriptive feature: Synonymous with analyzing a descriptive feature. This is
done by partitioning.

Tree: An organized structure that shows the flow of decisions made by the programmer or computer.