chapter-4-2

February 2, 2022

Run in Google Colab

TO BE DONE

- Random Forest e ensamble methods

1 Supervised Models: Decision Trees

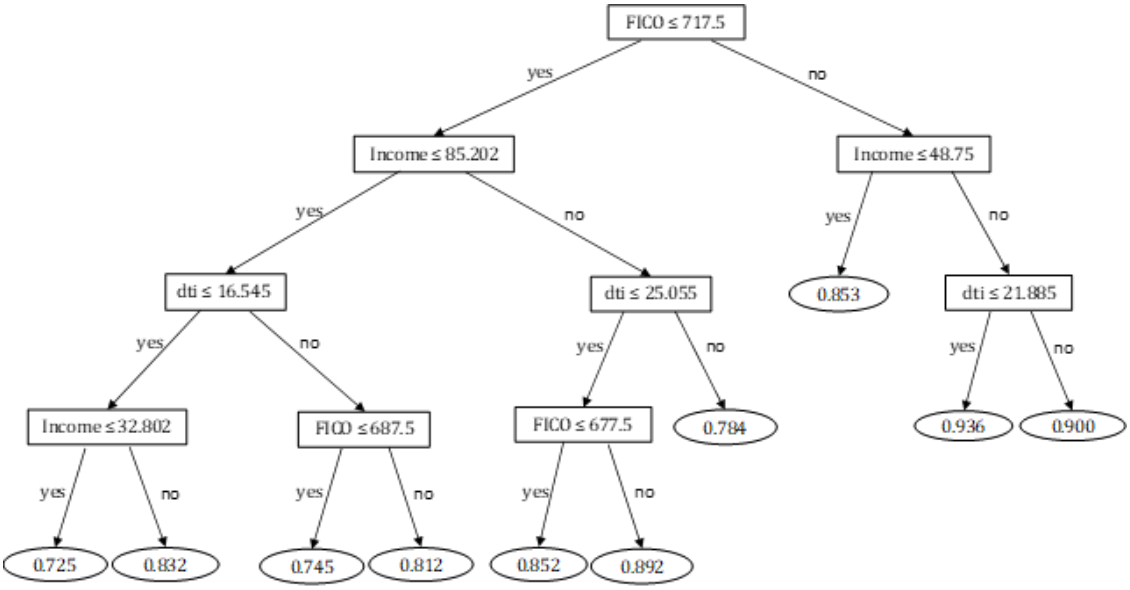
1.1 What is a decision tree?

Decision trees can be used for regression (continuous real-valued output, e.g. predicting the price of a house) or classification (categorical output, e.g. predicting email spam vs. no spam), but here we will focus on classification. **A decision tree classifier is a binary tree where predictions are made by traversing the tree from root to leaf — at each node, we go left if a feature is less than a threshold, right otherwise. Finally, each leaf is associated with a class, which is the output of the predictor**.

1.1.1 Why Decision Trees?

- Decision tress often mimic the human level thinking so its so simple to understand the data and make some good interpretations;
- Decision trees actually make you see the logic for the data to interpret(not like black box algorithms like SVM,NN,etc..);
- There is non requirement that the relationship between the target and the features be linear;
- The tree automatically selects the best features to make the prediction;
- A decision tree is less sensitive to outlying observations than a regression.

For example : if we are classifying bank loan application for a customer, the decision tree may look like this Here we can see the logic how it is making the decision.



It's simple and clear. So what is the decision tree?

1.2 Building a Decision Tree

During model training on feature-target relationships, a tree is grown from a **root** (parent) node (all data containing feature-target relationships), which is then recursively split into child nodes (subset of the entire data) in a binary fashion. Generally, each split is performed on a single feature in the parent node. You will have different kind of splitting depending on the nature of the feature:

- you can have a very simple decision tree in which the decision is a boolean condition (true, false) . . .
- . . . or numeric data

In the last case we have to define a desired **threshold** value of the feature. For instance, during each split of the parent node, we go to left node (with the corresponding subset of data) if a feature is less than the threshold, and right node otherwise. But how do we decide on the split? The feature to put at the root node is the one with the most **information gain**.

1.2.1 The Impurity Concept

Decision trees use the concept of **impurity** to describe how homogeneous or "pure" a node is. **A node is pure if all its samples belong to the same class**, while **a node with many samples from many different classes is called impure**. The difference between the impurity of a node and that of the child nodes is called **Information Gain**.

The goal of a decision tree is, at each layer, to try to split the data into two (or more) groups, **so that data that fall into the same group are most similar to each other (homogeneity), and groups are as different as possible from each other (heterogeneity)**.

In order to split the nodes at the most informative features, we need to define an objective function that we want to optimize via the tree learning algorithm. Here, our objective function is to maximize the **IG** at each split, which we define as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j) \tag{1}$$

Here,

- f is the feature to perform the split;
- and are the dataset of the parent and jth child node;
- I is our impurity measure;
- is the total number of training examples at the parent node;
- and is the number of examples in the jth child node.

As we can see, the information gain is simply the difference between the impurity of the parent node and the sum of the child node impurities—the lower the impurities of the child nodes, the larger the information gain. However, for simplicity and to reduce the combinatorial search space, most libraries (including scikit-learn) implement binary decision trees. This means that each parent node is split into two child nodes, D_{left} and D_{right}:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right}) \tag{2}$$

1.2.2 Measures of Information Gain

The three impurity measures or splitting criteria that are commonly used in binary decision trees are **Entropy** (), **Gini impurity** () and the **Classification Error** (). In the following we will focus only on the first two.

Entropy One measure of information gain is based on *entropy*.

Suppose that there are n possible outcomes and p_i is the probability of outcome i with $\sum_{i=1}^{n} p_i = 1$, entropy can be defined as:

$$\text{Entropy} = -\sum_{i=1}^{n} p_i \ln(p_i)$$

For example let's suppose some data for job application and we find that the 20% of them received an offer. Suppose further that 50% of job applicants have a relevant degree. If both those with a relevant degree and those without a relevant degree had a 20% chance of receiving a job offer, there would be no information gain to knowing wheter an applicant has a relevant degree. Suppose however that: - 30% of those with a relevant degree received a job offer - 10% of those without a relevant degree received a job offer

then there is clearly some information gain to knowing whether an applicant has a relevant degree. Let's calculate the information gain with entropy as a measure.

[150]:
```
import math

# initial entropy, the only information we have is that 20% of applicants find a␣
 ↪job
p = 0.2

entropy_ini = -p*math.log(p) - (1-p)*math.log(1-p)
print(entropy_ini)
```

```
0.5004024235381879
```

If a candidate has a relevant degree this becomes

[151]:
```
p = 0.3

entropy_1 = -p*math.log(p) - (1-p)*math.log(1-p)
print(entropy_1)
```

```
0.6108643020548935
```

if a candidate does not have a relevant degree:

[152]:
```
p = 0.1

entropy_2 = -p*math.log(p) - (1-p)*math.log(1-p)
print(entropy_2)
```

```
0.3250829733914482
```

Because 50% of candidates have a relevant degree, the expected value of entropy after it is determined whether a candidate has a relevant degree is

[153]:
```
entropy_exp = .5*entropy_1 + 0.5*entropy_2
print('Expected Entropy = ' + str(entropy_exp) + '\n')
#
# Now we can easily calculate the expected uncertainty reduction as:
#
print('Information Gain = ' + str(entropy_ini - entropy_exp))
```

```
Expected Entropy = 0.46797363772317085


Information Gain = 0.032428785815017014
```

When constructing the decision tree, we first search for the feature that has the biggest information gain. This is put at the root of the tree. For each branch emanating from the root we then search for a feature (other than the one at the root) that has the biggest information gain and so on ...

Gini Impurity An alternative to entropy for quantifying information gain is the Gini Impurity, which is basically a concept to quantify how homogeneous or "pure" a node is, with relation to the

distribution of the targets in the node. A node is considered pure (G=0) if all training samples in the node belong to the same class, while a node with many training samples from many different classes will have a Gini Impurity close to 1.

$$G = 1 - \sum_{k=1}^{n} p_k^2 = 1 - \sum_{k=1}^{n} \left(\frac{m_k}{m}\right)^2 \tag{3}$$

where p_k is the fraction of samples belonging to class k, n is the number of classes, m are all the training samples in the node and m_k are the training examples in each class.

For example if a node contains five samples, with two of class 1, two of class 2, one of class 3 and none of class 4, then

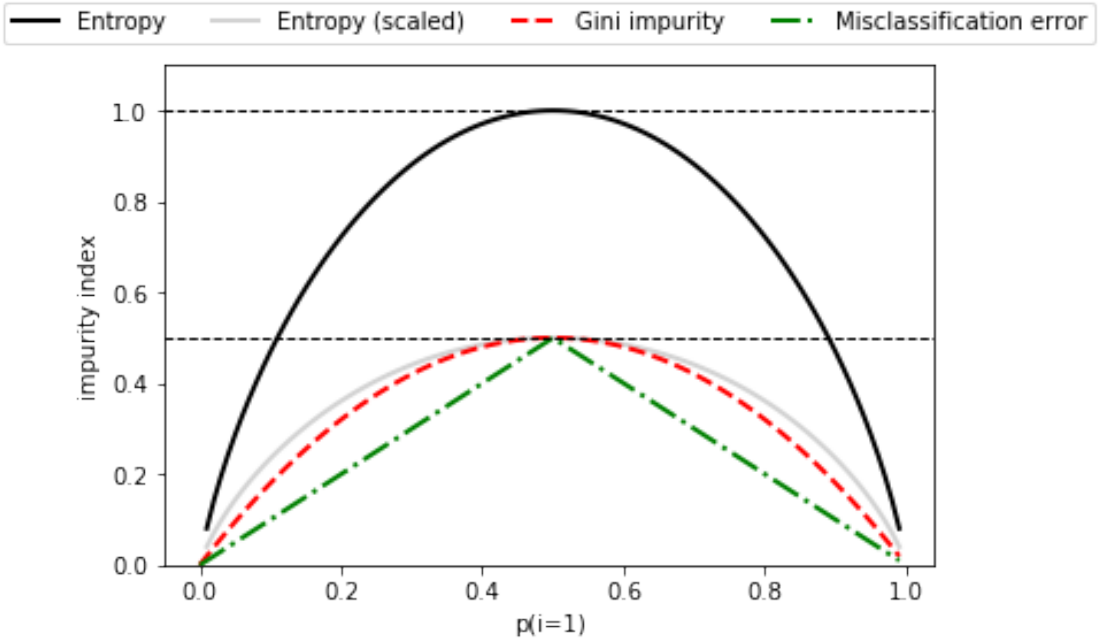
$$G = 1 - \left(\frac{2}{5}\right)^2 - \left(\frac{2}{5}\right)^2 - \left(\frac{1}{5}\right)^2 = 0.64$$

Comparison For a more visual comparison of the three different impurity criteria that we discussed previously, let's plot the impurity indices for the probability range $[0, 1]$ for class 1.

[154]:
```
import matplotlib.pyplot as plt
import numpy as np

def gini(p):
    return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))

def entropy(p):
    return - p*np.log2(p) - (1 - p)*np.log2((1 - p))

def error(p):
    return 1 - np.max([p, 1 - p])

x = np.arange(0.0, 1.0, 0.01)
ent = [entropy(p) if p != 0 else None for p in x]
sc_ent = [e*0.5 if e else None for e in ent]
err = [error(i) for i in x]
fig = plt.figure()
ax = plt.subplot(111)
for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
    ['Entropy', 'Entropy (scaled)',
    'Gini impurity',
    'Misclassification error'],
    ['-', '-', '--', '-.'],
    ['black', 'lightgray',
    'red', 'green', 'cyan']):
    line = ax.plot(x, i, label=lab,
    linestyle=ls, lw=2, color=c)
```

```
ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15), ncol=5, fancybox=True,␣
 ↪shadow=False)
ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
plt.ylim([0, 1.1])
plt.xlabel('p(i=1)')
plt.ylabel('impurity index')
```

[154]: `Text(0, 0.5, 'impurity index')`



1.3 Application to Credit Decision (Hull J. C. Chapter 4)

[155]:
```
import math
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree, export_graphviz,␣
 ↪export_text
from IPython.display import Image
from sklearn.metrics import accuracy_score, recall_score, precision_score,␣
 ↪f1_score
from sklearn.metrics import confusion_matrix, classification_report,␣
 ↪roc_auc_score
```

```
from sklearn.metrics import roc_curve, auc, average_precision_score
```

We now apply the decision tree approach using the entropy method to the Lending Club Data Introduced in chapter 4.1.

[156]:
```
#
# load file lendingclub_traindata.xlsx
#
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[157]:
```
train = pd.read_excel(path + 'lendingclub_traindata.xlsx', engine='openpyxl')
# 1 = good, 0 = default
print(train.head())
```

```
   home_ownership  income    dti  fico_low  loan_status
0               1  44.304  18.47       690            0
1               0  38.500  33.73       660            0
2               1  54.000  19.00       660            0
3               1  60.000  33.98       695            0
4               0  39.354  10.85       685            0
```

[158]:
```
# remove target column to create feature only dataset
X = train.drop('loan_status',axis=1)
# store target column
y = train['loan_status']
print(X.shape, y.shape)
```

```
(8695, 4) (8695,)
```

1.3.1 Step-by-Step Analysis

Remember that without any further information, the probability of a good loan is about 82.76%, infact we have a total of 8695 observations ot those 7196 were for good loans and 1499 were for the bad loans that defaulted.

[159]:
```
majority_class = y.mode()[0]         # predict fully paid only
prediction = np.full(shape=y.shape, fill_value=majority_class)
p_good = accuracy_score(y, prediction)
print(p_good)
```

```
0.8276020701552617
```

The initial entropy is therefore

[160]:
```
entropy_ini = -p_good * math.log(p_good) - (1-p_good)* math.log(1-p_good)
print(entropy_ini)
```

```
0.45966813572655896
```

We will consider the same four features and same data as in the previous notebook:

- A categorical variable indicating wheter the applicant rests or owns a home
- The applicant's income
- The applicant's debt to income ratio (dti)
- The applicantion credit score (FICO)

[161]:
```
X.columns = ['Owns Home','Income','dti','FICO']
```

First of all let's make a step-by-step calculation before use the magic of scikit-learn library. Let's calculate the percentage of applicants that own their house

Assuming Home Ownership as Root

[162]:
```
owners = train[train['home_ownership'] == 1]
renter = train[train['home_ownership'] == 0]

owners_perc = owners.shape[0]/X.shape[0]
print('App Owner = ' + str(owners_perc))
print('App Rent  = ' + str(1-owners_perc))
```

```
App Owner = 0.5913743530764808
App Rent  = 0.4086256469235192
```

Then find the percentage of loans that were good for applicants that own their house and for applicants that rent their house

[163]:
```
#
# Find the number of loans that were good for owners
#
n1 = owners[owners['loan_status']==1]
p1 = n1.shape[0] / owners.shape[0]
print(p1)
```

```
0.8444185141968106
```

[164]:
```
#
# Find the number of loans that were good for renters
#
n2 = renter[renter['loan_status']==1]
p2 = n2.shape[0] / renter.shape[0]
print(p2)
```

```
0.8032648466084998
```

The expected entropy if home ownership (*but no other feature*) becomes known is therefore:

[165]:
```
entropy_exp = -owners_perc*(p1*math.log(p1) + (1-p1)*math.log(1-p1)) -␣
 ↪(1-owners_perc)*(p2*math.log(p2) + (1-p2)*math.log(1-p2))
print(entropy_exp)
```

```
0.4582474114672313
```

The expected reduction in entropy is therefore a modest:

[166]:
```
print(entropy_ini - entropy_exp)
```

```
0.0014207242593276548
```

[167]:
```
results = pd.DataFrame(columns=('Feature','Treshold','Expected H', 'IG'))
results.loc[0] = ['Home Ownership', 0, entropy_exp, entropy_ini - entropy_exp]
```

Assuming Applicant's Income as Root The calculation of the expected reduction in entropy from Income **requires the specification of a threshold income**. Define

- p_1 : probability that income is greater than the threshold
- p_2 : probability that, if income is greater than the threshold, the borrower does not default
- p_3 : probability that if income is less than the threshold the borrower does not default

The expected entropy is

$$\hat{E} = p_1 \left[-p_2 \log(p_2) - (1-p_2) \log(1-p_2)\right] + (1-p_1) \left[-p_3 \log(p_3) - (1-p_3) \log(1-p_3)\right]$$

[168]:
```
#
# p1 computation
#
threshold = 50
# applicants with income grater than the threshold
sample_1 = train[train['income'] >  threshold]
# applicants with income less than the threshold
sample_2 = train[train['income'] <= threshold]
# applicants with income greater than the threshold which dont default
sample_3 = sample_1[sample_1['loan_status']==1]
# applicants with income less than the threshold which dont default
sample_4 = sample_2[sample_2['loan_status']==1]

p1 = sample_1.shape[0] / train.shape[0]
p2 = sample_3.shape[0] / sample_1.shape[0]
p3 = sample_4.shape[0] / sample_2.shape[0]

print(p1)
print(p2)
```

```
print(p3)
```

```
0.6803910293271995
0.8466869506423259
0.7869737315581145
```

[169]:
```
entropy_exp = p1*(-p2*math.log(p2) - (1-p2)*math.log(1-p2)) + (1-p1)*(-p3*math.
 ↪log(p3) - (1-p3)*math.log(1-p3))
print(entropy_exp)
```

```
0.45702631325619647
```

We can make a function which takes as input a threshold and return the entropy

[170]:
```
def ExpectedEntropy1(threshold, df):
    s1 = df[df['income'] >  threshold]
    # applicants with income less than the threshold
    s2 = df[df['income'] <= threshold]
    # applicants with income greater than the threshold which dont default
    s3 = s1[s1['loan_status']==1]
    # applicants with income less than the threshold which dont default
    s4 = s2[s2['loan_status']==1]

    p1 = s1.shape[0] / df.shape[0]
    p2 = s3.shape[0] / s1.shape[0]
    p3 = s4.shape[0] / s2.shape[0]

    e = p1*(-p2*math.log(p2) - (1-p2)*math.log(1-p2)) + (1-p1)*(-p3*math.log(p3)␣
 ↪- (1-p3)*math.log(1-p3))
    return(e)

# sanity check
print(ExpectedEntropy1(50, train))
```

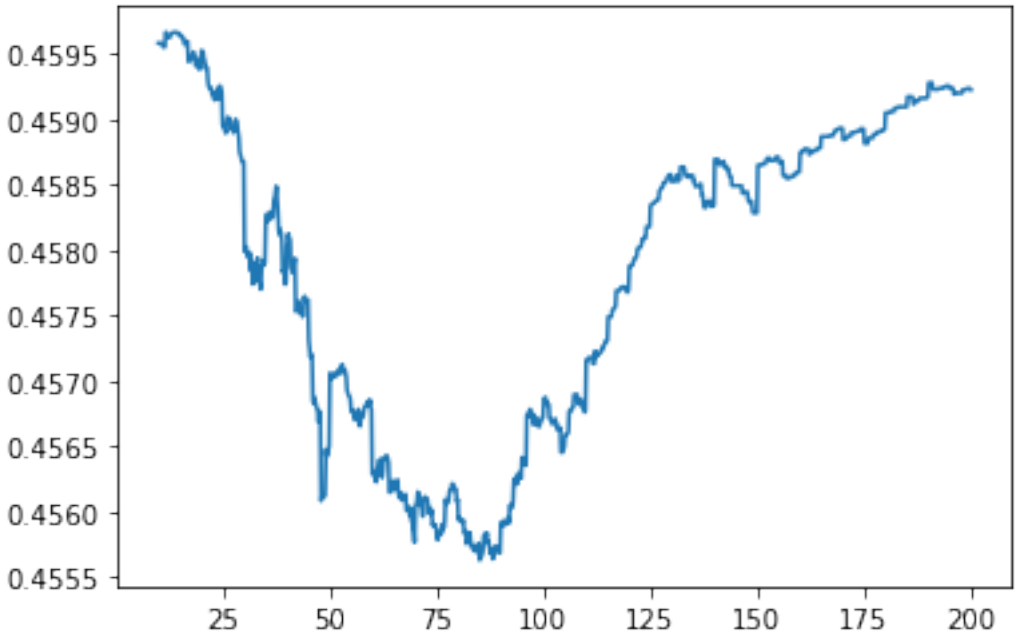
```
0.45702631325619647
```

We carry out an iterative search to determine the value of the threshold that minimizes this expected entropy for the training set

[171]:
```
import matplotlib.pyplot as plt

# 100 linearly spaced numbers
xps = np.linspace(10,200,1000)
yps = []
for xp in xps:
    yps.append(ExpectedEntropy1(xp, train))

yps = np.array(yps)
```

```
plt.plot(xps, yps)
```

[171]: [<matplotlib.lines.Line2D at 0x1544e3a3cc8>]



[172]:
```
minimum = np.where(yps == np.amin(yps))
xps_min = xps[minimum][0]
print(xps_min)
print(ExpectedEntropy1(xps_min, train))
```

```
85.12512512512512
0.4556300235689159
```

[173]:
```
entropy_exp = ExpectedEntropy1(85.193, train)
print(entropy_exp)
print(entropy_ini - entropy_exp)
```

```
0.4556300235689159
0.004038112157643048
```

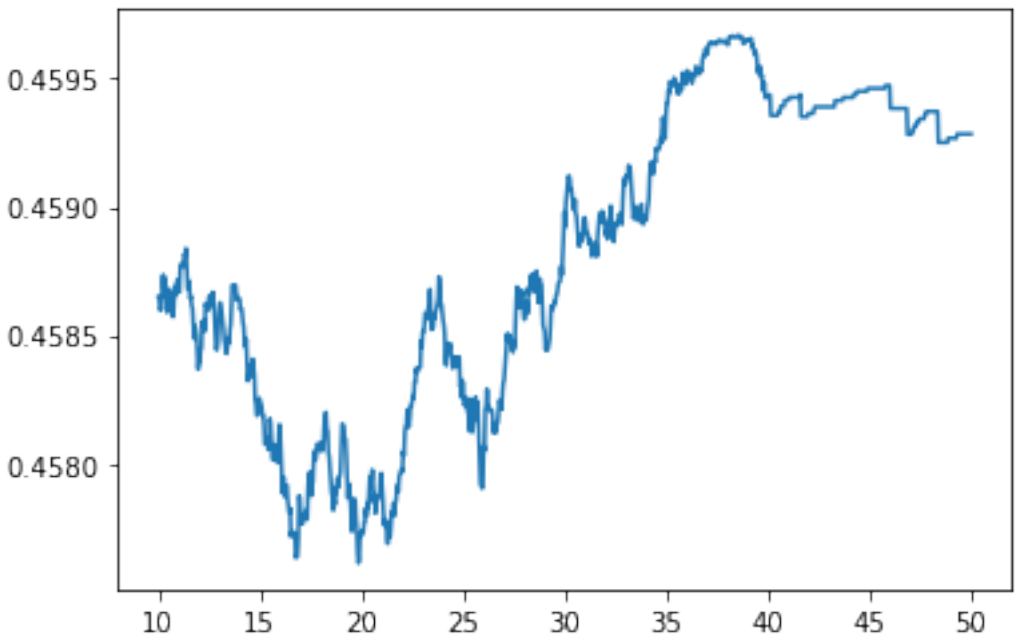
[174]:
```
results.loc[1] = ['Income',xps_min, entropy_exp, entropy_ini - entropy_exp]
```

Assuming Applicant's dti (debt to income ratio) as Root

[175]:
```
def ExpectedEntropy2(threshold, df):
    s1 = df[df['dti'] >  threshold]
    # applicants with income less than the threshold
    s2 = df[df['dti'] <= threshold]
    # applicants with dti greater than the threshold which dont default
    s3 = s1[s1['loan_status']==1]
    # applicants with dti less than the threshold which dont default
    s4 = s2[s2['loan_status']==1]

    p1 = s1.shape[0] / df.shape[0]
    p2 = s3.shape[0] / s1.shape[0]
    p3 = s4.shape[0] / s2.shape[0]

    e = p1*(-p2*math.log(p2) - (1-p2)*math.log(1-p2)) + (1-p1)*(-p3*math.log(p3)␣
 ↪- (1-p3)*math.log(1-p3))
    return(e)
```

[176]:
```
# 100 linearly spaced numbers
xps = np.linspace(10,50,1000)
yps = []
for xp in xps:
    yps.append(ExpectedEntropy2(xp, train))

yps = np.array(yps)

plt.plot(xps, yps)
```

[176]:
```
[<matplotlib.lines.Line2D at 0x1544cc323c8>]
```



[177]:
```
minimum = np.where(yps == np.amin(yps))
xps_min = xps[minimum][0]

entropy_exp = ExpectedEntropy2(xps_min, train)

print(xps_min)
print(entropy_exp)
print(entropy_ini - entropy_exp)
```

```
19.84984984984985
0.45761406059416887
0.0020540751323900874
```

[178]:
```
results.loc[2] = ['dti',xps_min, entropy_exp, entropy_ini - entropy_exp]
```

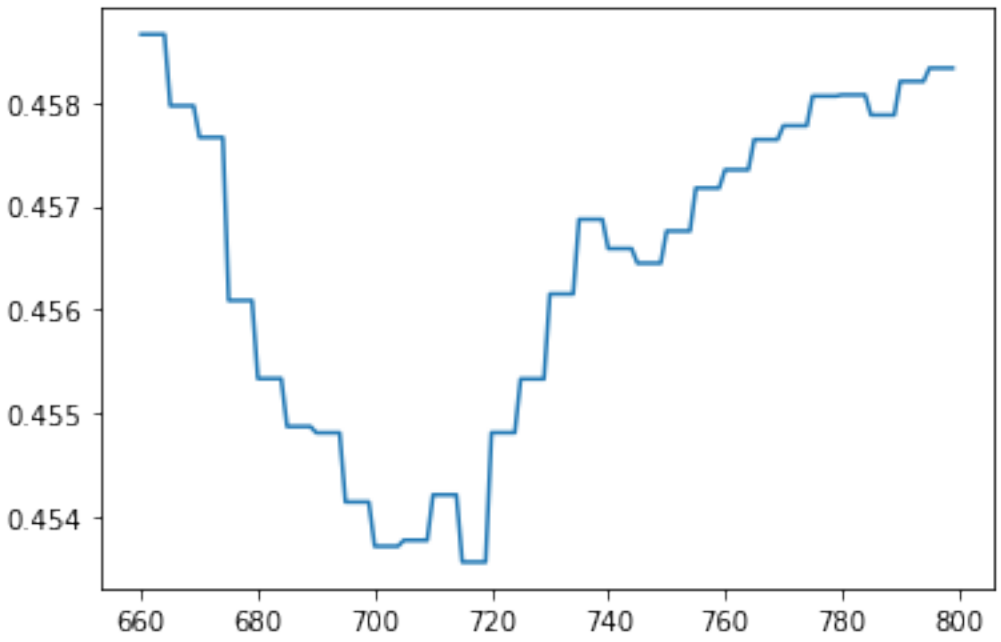
Assuming Applicant's FICO as Root

[179]:
```
def ExpectedEntropy3(threshold, df):
    s1 = df[df['fico_low'] >  threshold]
    # applicants with income less than the threshold
    s2 = df[df['fico_low'] <= threshold]
    # applicants with fico greater than the threshold which dont default
    s3 = s1[s1['loan_status']==1]
    # applicants with fico less than the threshold which dont default
    s4 = s2[s2['loan_status']==1]

    p1 = s1.shape[0] / df.shape[0]
    p2 = s3.shape[0] / s1.shape[0]
    p3 = s4.shape[0] / s2.shape[0]

    e = p1*(-p2*math.log(p2) - (1-p2)*math.log(1-p2)) + (1-p1)*(-p3*math.log(p3)␣
 ↪- (1-p3)*math.log(1-p3))
    return(e)
```

[180]:
```
a = train['fico_low'].min()

xps = np.arange(a, 800)
yps = []

for xp in xps:
    yps.append(ExpectedEntropy3(xp, train))

yps = np.array(yps)
```

```
plt.plot(xps, yps)
```

[180]: [<matplotlib.lines.Line2D at 0x1544e268848>]



[181]:
```
minimum = np.where(yps == np.amin(yps))
xps_min = xps[minimum][0]

entropy_exp = ExpectedEntropy3(xps_min, train)

print(xps_min)
print(entropy_exp)
print(entropy_ini - entropy_exp)
```

```
715
0.45355502172339535
0.006113114003163611
```

[182]:
```
results.loc[3] = ['FICO',xps_min, entropy_exp, entropy_ini - entropy_exp]
```

[183]:
```
results
```

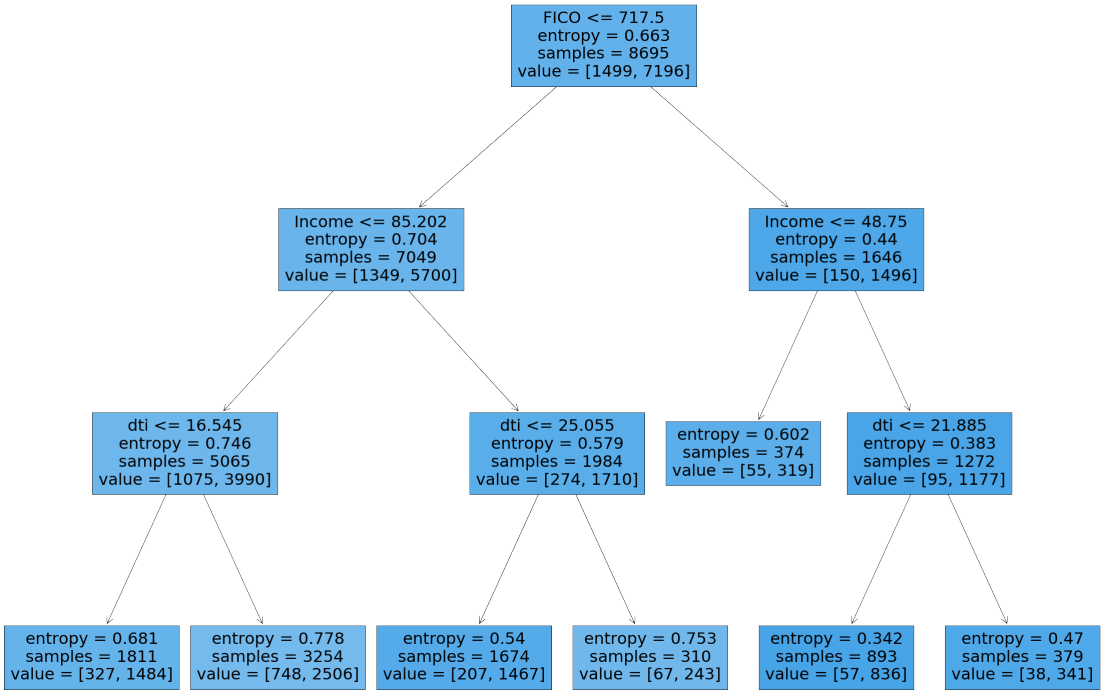
[183]:

	Feature	Treshold	Expected H	IG
0	Home Ownership	0	0.458247	0.001421
1	Income	85.1251	0.455630	0.004038
2	dti	19.8498	0.457614	0.002054
3	FICO	715	0.453555	0.006113

The FICO score with a threshold of 715 has the greatest information gain. It is therefore put at the node of the three.

1.3.2 Using the DecisionTreeClassifier

[184]:
```
clf = DecisionTreeClassifier(criterion='entropy',max_depth=3,min_samples_split=1000,min_samples_lea
clf = clf.fit(X,y)
fig, ax = plt.subplots(figsize=(40, 30))
plot_tree(clf, filled=True, feature_names=X.columns, proportion=False)
plt.show()
```



[185]:
```
r = export_text(clf,feature_names=['Owns Home','Income','dti','FICO'])
```

[186]:
```
print(r)
```

```
|--- FICO <= 717.50
|   |--- Income <= 85.20
|   |   |--- dti <= 16.55
|   |   |   |--- class: 1
|   |   |--- dti >  16.55
```

```
|   |   |   |--- class: 1
|   |--- Income >  85.20
|   |   |--- dti <= 25.05
|   |   |   |--- class: 1
|   |   |--- dti >  25.05
|   |   |   |--- class: 1
|--- FICO >  717.50
|   |--- Income <= 48.75
|   |   |--- class: 1
|   |--- Income >  48.75
|   |   |--- dti <= 21.88
|   |   |   |--- class: 1
|   |   |--- dti >  21.88
|   |   |   |--- class: 1
```

[187]:
```
#
# load file lendingclub_testdata.xlsx
#
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[188]:
```
test = pd.read_excel(path + 'lendingclub_testdata.xlsx', engine='openpyxl')
# 1 = good, 0 = default
print(test.head())
```

```
   home_ownership  income    dti  fico_low  loan_status
0               1   127.0  10.94       675            0
1               1   197.0  15.64       710            0
2               1    25.5  28.75       670            0
3               1    80.0  20.16       660            0
4               0    57.0  30.60       675            0
```

[189]:
```
# remove target column to create feature only dataset
X_test = test.drop('loan_status',axis=1)
# store target column
y_test = test['loan_status']
y_pred = clf.predict(X_test)
```

Note that for the default prediciton (i.e., with threshold=0.5), it totally missed all the bad loans.

[190]:
```
n_test = len(y_test)
cm = (confusion_matrix(y_test,y_pred,labels=[1, 0], sample_weight=None)/
 ↪n_test)*100
```

```
plt.figure(figsize=(6, 4))       # format the plot size
ax = plt.subplot()
sns.heatmap(cm, annot=True, ax = ax, fmt='.4g', cmap="Blues")
ax.set_xlabel('\nPredicted labels'); ax.set_ylabel('True labels\n')
ax.xaxis.tick_top()
ax.yaxis.set_ticklabels(['Good','Defaulted'],verticalalignment='center')
ax.xaxis.set_ticklabels(['Good','Defaulted'])
plt.show()
```

[191]:
```
THRESHOLD = [.5, .75, .80, .85]
results = pd.DataFrame(columns=["THRESHOLD", "accuracy", "recall", "tnr", "fpr",␣
 ↪"precision", "f1_score"]) # df to store results
results['THRESHOLD'] = THRESHOLD                                                ␣
 ↪                          # threshold column
n_test = len(y_test)
Q = clf.predict_proba(X_test)[:,1]
j = 0
for i in THRESHOLD:                                                             ␣
 ↪                          # iterate over each threshold
                                                                        # fit␣
 ↪data to model
    preds = np.where(Q>i, 1, 0)                                        # if prob␣
 ↪> threshold, predict 1
```

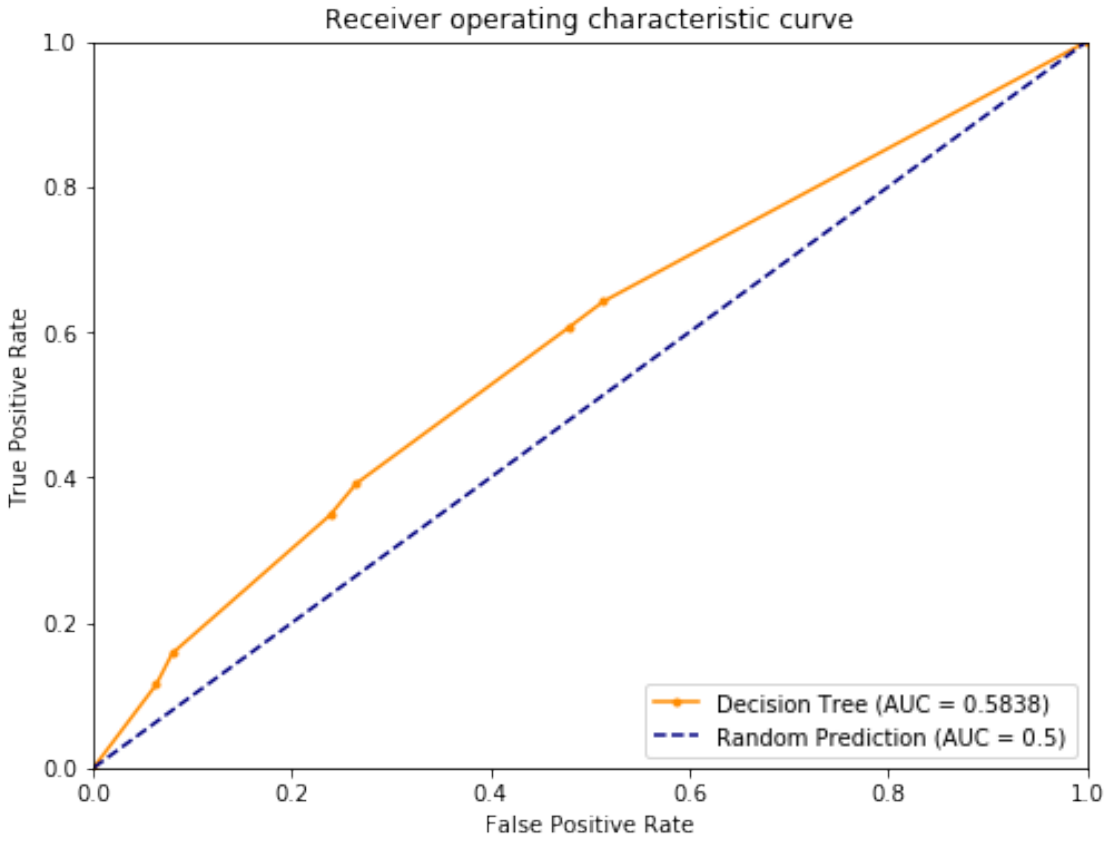
```
    cm = (confusion_matrix(y_test, preds,labels=[1, 0], sample_weight=None)/
↪n_test)*100
    # confusion matrix (in percentage)

    print('Confusion matrix for threshold =',i)
    print(cm)
    print(' ')

    TP = cm[0][0]                                                          ␣
↪                           # True Positives
    FN = cm[0][1]                                                          ␣
↪                           # False Positives
    FP = cm[1][0]                                                          ␣
↪                           # True Negatives
    TN = cm[1][1]                                                          ␣
↪                           # False Negatives

    results.iloc[j,1] = accuracy_score(y_test, preds)
    results.iloc[j,2] = recall_score(y_test, preds)
    results.iloc[j,3] = TN/(FP+TN)                                         ␣
↪                           # True negative rate
    results.iloc[j,4] = FP/(FP+TN)                                         ␣
↪                           # False positive rate
    results.iloc[j,5] = precision_score(y_test, preds)
    results.iloc[j,6] = f1_score(y_test, preds)

    j += 1

print('ALL METRICS')
print(results.T.to_string(header=False))
```

```
Confusion matrix for threshold = 0.5
[[82.11629479  0.        ]
 [17.88370521  0.        ]]

Confusion matrix for threshold = 0.75
[[82.11629479  0.        ]
 [17.88370521  0.        ]]

Confusion matrix for threshold = 0.8
[[49.76335362 32.35294118]
 [ 8.5530764   9.3306288 ]]

Confusion matrix for threshold = 0.85
[[32.15010142 49.96619337]
 [ 4.73292765 13.15077755]]
```

```
ALL METRICS
THRESHOLD       0.5      0.75       0.8      0.85
accuracy   0.821163  0.821163   0.59094  0.453009
recall            1         1  0.606011  0.391519
tnr               0         0  0.521739   0.73535
fpr               1         1  0.478261   0.26465
precision  0.821163  0.821163  0.853333  0.871677
f1_score   0.901801  0.901801  0.708714  0.540341
```

[192]:
```
# Compute the ROC curve and AUC
fpr, tpr, _ = roc_curve(y_test, Q)
roc_auc = auc(fpr,tpr)
```

[193]:
```
plt.figure(figsize=(8,6))       # format the plot size
lw = 1.5
plt.plot(fpr, tpr, color='darkorange', marker='.',
         lw=lw, label='Decision Tree (AUC = %0.4f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--',
         label='Random Prediction (AUC = 0.5)' )
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic curve')
plt.legend(loc="lower right")
plt.show()
```



1.4 Continuous Target Variables

So far we have considered the use of decision trees for classification. We now describe how they can be used to predict the value of a continuous variable. Suppose that the feature at the root node is X and the threshold value for X is Q. We choose X and Q to minimize the expected mean squared error (mse) in the prediction of the target for the training set. In other words, we minimize

$$Prob(X > Q) \times (\text{mse if } X > Q) + Prob(X \leq Q) \times (\text{mse if } X \leq Q)$$

The feature at the next node and its threshold are chosen similarly. The value predicted at a tree leaf is the average of the values for the observations corresponding to the leaf.

We will illustrate this procedure for the house price data considered in the previous notebook (see also J. C. Hull Chapter 3). To keep the example manageable we consider only two features: Overall Quality (scale 1 to 10) and Living Area (Square Feet).

[132]:
```
import pandas as pd
import numpy as np
import seaborn as sns
```

```
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor, plot_tree, export_graphviz,␣
 ↪export_text
from IPython.display import Image
from sklearn.metrics import accuracy_score, recall_score, precision_score,␣
 ↪f1_score
from sklearn.metrics import confusion_matrix, classification_report,␣
 ↪mean_squared_error
from sklearn.metrics import roc_curve, auc, average_precision_score
import math
```

[133]:
```
#
# load file IOWA_Training_Data.xlsx
#
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[134]:
```
train = pd.read_excel(path + 'IOWA_Training_Data.xlsx', engine='openpyxl')
# 1 = good, 0 = default
print(train.head())
```

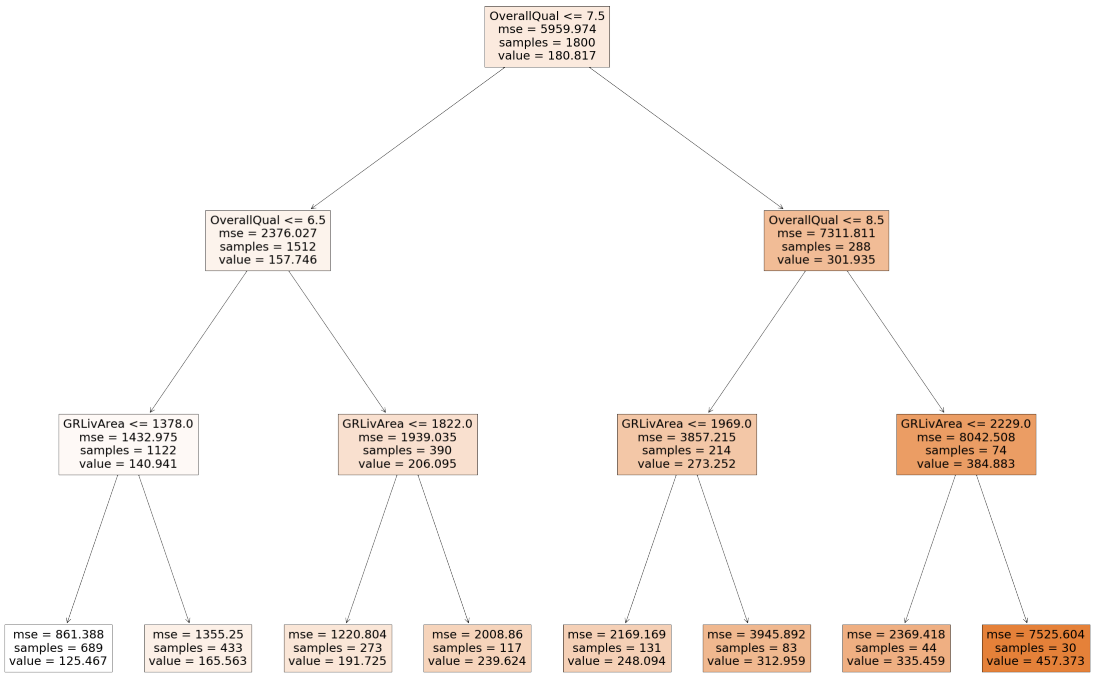
```
   OverallQual  GrLivArea  Sale Price
0            7       1710       208.5
1            6       1262       181.5
2            7       1786       223.5
3            7       1717       140.0
4            8       2198       250.0
```

[135]:
```
# remove target column to create feature only dataset
X_train = train.drop('Sale Price',axis=1)
# store target column
y_train = train['Sale Price']
print(X_train.shape, y_train.shape)
```

```
(1800, 2) (1800,)
```

[136]:
```
X_train.columns = ['OverallQual','GRLivArea']
```

[137]:
```
pred = DecisionTreeRegressor(criterion='mse',max_depth=3,random_state=0)
pred = pred.fit(X_train,y_train)
fig, ax = plt.subplots(figsize=(40, 30))
plot_tree(pred, filled=True, feature_names=X_train.columns, proportion=False)
plt.show()
```



[138]:
```
r = export_text(pred,feature_names=['OverallQual','GRLivArea'])
```

[139]:
```
print(r)
```

```
|--- OverallQual <= 7.50
|   |--- OverallQual <= 6.50
|   |   |--- GRLivArea <= 1378.00
|   |   |   |--- value: [125.47]
|   |   |--- GRLivArea >  1378.00
|   |   |   |--- value: [165.56]
|   |--- OverallQual >  6.50
|   |   |--- GRLivArea <= 1822.00
|   |   |   |--- value: [191.72]
|   |   |--- GRLivArea >  1822.00
|   |   |   |--- value: [239.62]
|--- OverallQual >  7.50
|   |--- OverallQual <= 8.50
|   |   |--- GRLivArea <= 1969.00
|   |   |   |--- value: [248.09]
|   |   |--- GRLivArea >  1969.00
|   |   |   |--- value: [312.96]
|   |--- OverallQual >  8.50
```

```
|   |   |--- GRLivArea <= 2229.00
|   |   |   |--- value: [335.46]
|   |   |--- GRLivArea >  2229.00
|   |   |   |--- value: [457.37]
```

[140]:
```
#
# load file IOWA_Validation_Data.xlsx
#
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[141]:
```
validation = pd.read_excel(path + 'IOWA_Validation_Data.xlsx', engine =␣
 ↪'openpyxl')
# 1 = good, 0 = default
print(validation.head())
```

```
   OverallQual  GrLivArea  Sale Price
0            6       1045       127.0
1            6       1378       128.9
2            6       1944       103.5
3            5       1306       130.0
4            5       1464       129.0
```

[142]:
```
# remove target column to create feature only dataset
X_validation = validation.drop('Sale Price',axis=1)
# store target column
y_validation = validation['Sale Price']
print(X_validation.shape, y_validation.shape)
```

```
(600, 2) (600,)
```

[143]:
```
#
# load file IOWA_Test_Data.xlsx
#
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[144]:
```
test = pd.read_excel(path + 'IOWA_Test_Data.xlsx', engine='openpyxl')
# 1 = good, 0 = default
print(test.head())
```

```
   OverallQual  GrLivArea  Sale Price
0            5       1053      142.10
1            5       1144      120.00
2            6       1721      174.85
3            5        922      116.00
4            5       1411      130.00
```

[145]:
```
# remove target column to create feature only dataset
X_test = test.drop('Sale Price',axis=1)
# store target column
y_test = test['Sale Price']
print(X_test.shape, y_test.shape)
```

```
(508, 2) (508,)
```

[146]:
```
y_pred_train=pred.predict(X_train)
mse = mean_squared_error(y_pred_train,y_train)
rmse=math.sqrt(mse)
print("rmse for training set")
print(rmse)
```

```
rmse for training set
38.660405648678584
```

[147]:
```
y_pred_validation=pred.predict(X_validation)
mse = mean_squared_error(y_pred_validation,y_validation)
rmse=math.sqrt(mse)
print("rmse for validation set")
print(rmse)
```

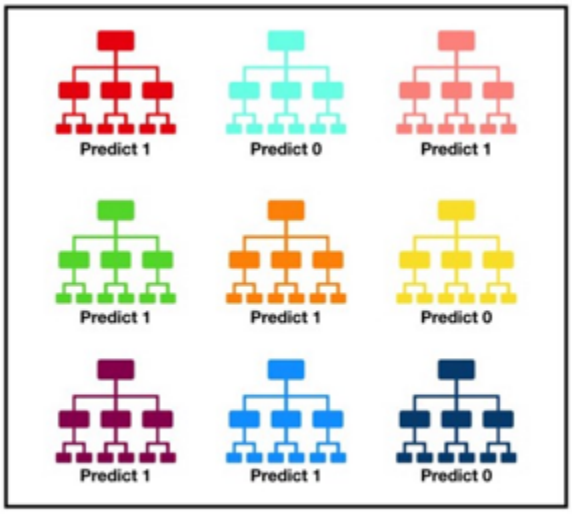
```
rmse for validation set
40.462035147026455
```

[148]:
```
y_pred_test=pred.predict(X_test)
mse = mean_squared_error(y_pred_test,y_test)
rmse=math.sqrt(mse)
print("rmse for test set")
print(rmse)
```

```
rmse for test set
39.048800554631995
```

1.5 Random Forest

Random forest, like its name implies, consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction and the class with the most votes becomes our model's prediction.



The reason that the random forest model works is that a large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models. The low correlation between models is the key. Just like how investments with low correlations (like stocks and bonds) come together to form a portfolio that is greater than the sum of its parts, uncorrelated models can produce ensemble predictions that are more accurate than any of the individual predictions. The reason for this wonderful effect is that the trees protect each other from their individual errors (as long as they don't constantly all err in the same direction). While some trees may be wrong, many other trees will be right, so as a group the trees are able to move in the correct direction.

1.6 References

[ ]: