1 Tree-Based Methods

1.1 8.1 Classification tree

This problem involves the OJ dataset (Data/OJ.csv)

Orange Juice Data

The data contains 1070 purchases where the customer either purchased Citrus Hill or Minute Maid Orange Juice. A number of characteristics of the customer and product are recorded.

A data frame with 1070 observations on the following 18 variables:

- Purchase: A factor with levels CH and MM indicating whether the customer purchased Citrus Hill or Minute Maid Orange Juice
- WeekofPurchase: Week of purchase
- StoreID: Store ID
- PriceCH: Price charged for CH
- PriceMM: Price charged for MM
- DiscCH: Discount offered for CH
- DiscMM: Discount offered for MM
- SpecialCH: Indicator of special on CH
- SpecialMM: Indicator of special on MM
- LoyalCH: Customer brand loyalty for CH
- SalePriceMM: Sale price for MM
- SalePriceCH: Sale price for CH
- PriceDiff: Sale price of MM less sale price of CH
- Store7: A factor with levels No and Yes indicating whether the sale is at Store 7
- PctDiscMM: Percentage discount for MM
- PctDiscCH: Percentage discount for CH

In [1]: import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

from sklearn import tree
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier

import graphviz

%matplotlib inline
ListPriceDiff: List price of MM less list price of CH
STORE: Which of 5 possible stores the sale occurred at

1.1.1  a)
Create a training set containing a random sample of 800 observations, and a test set containing the remaining observations.

In [2]: np.random.seed(1)
OJ = pd.read_csv('Data/OJ.csv')

# sampling indices for training
trainIndex = np.random.choice(OJ.shape[0], size=800, replace=False)
train = OJ.iloc[trainIndex]  # training set
test = OJ.iloc[-OJ.index.isin(trainIndex)]  # test set

In [3]: OJ.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1070 entries, 0 to 1069
Data columns (total 18 columns):
Purchase        1070 non-null object
WeekofPurchase  1070 non-null int64
StoreID         1070 non-null int64
PriceCH         1070 non-null float64
PriceMM         1070 non-null float64
DiscCH          1070 non-null float64
DiscMM          1070 non-null float64
SpecialCH       1070 non-null int64
SpecialMM       1070 non-null int64
LoyalCH         1070 non-null float64
SalePriceMM     1070 non-null float64
SalePriceCH     1070 non-null float64
PriceDiff       1070 non-null float64
Store7          1070 non-null object
PctDiscMM       1070 non-null float64
PctDiscCH       1070 non-null float64
ListPriceDiff   1070 non-null float64
STORE           1070 non-null int64
dtypes: float64(11), int64(5), object(2)
memory usage: 150.5+ KB

1.1.2  b)
Learn a classification tree from the training data using the function sklearn.tree.DecisionTreeClassifier(), with Purchase as the output and the other variables as inputs. Don’t forget to handle qualitative variables correctly. To avoid severe overfit, you have to add some constraints to the tree, using, e.g., a maximum depth of 2 (max_depth=2).
In [4]:
X_train = train.copy().drop(columns=['Purchase'])
   # Need to transform the qualitative variables to dummy variables
X_train = pd.get_dummies(X_train, columns=['Store7'])
y_train = train['Purchase']
model = tree.DecisionTreeClassifier(max_depth=2)
model.fit(X=X_train, y=y_train)

Out[4]:
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=2,
               max_features=None, max_leaf_nodes=None,
               min_impurity_decrease=0.0, min_impurity_split=None,
               min_samples_leaf=1, min_samples_split=2,
               min_weight_fraction_leaf=0.0, presort=False, random_state=None,
               splitter='best')

1.1.3  c)

Use sklearn.tree.export_graphviz() and the Python module graphviz to visualize the tree and interpret the result. How many terminal nodes does the tree have? Pick one of the terminal nodes, and interpret the information displayed. Type 0J.info() to get information about all input variables in the data set.

In [5]:
dot_data = tree.export_graphviz(model, out_file=None, feature_names=X_train.columns,
                                class_names=model.classes_, filled=True, rounded=True,
                                leaves_parallel=True, proportion=True)

    graph = graphviz.Source(dot_data)

Out[5]:

```
LoyalCH <= 0.482
gini = 0.479
samples = 100.0%
value = [0.601, 0.399]
class = CH
PriceDiff <= 0.31
gini = 0.338
samples = 37.1%
value = [0.215, 0.785]
class = MM
True
LoyalCH <= 0.765
gini = 0.283
samples = 62.9%
value = [0.829, 0.171]
class = CH
False
    gini = 0.266
    samples = 29.2%
    value = [0.158, 0.842]
    class = MM
    gini = 0.49
    samples = 7.9%
    value = [0.429, 0.571]
    class = MM
    gini = 0.432
    samples = 30.1%
    value = [0.685, 0.315]
    class = CH
    gini = 0.073
    samples = 32.8%
    value = [0.962, 0.038]
    class = CH
```
1.1.4  d) Predict the response on the test data, and produce a confusion matrix comparing the test labels to the predicted test labels. What is the test error rate?

In [6]: X_test = test.copy().drop(columns=['Purchase'])
   X_test = pd.get_dummies(X_test, columns=['Store7'])
   y_test = test['Purchase']
   y_predict = model.predict(X_test)
   print('Accuracy rate is %.2f' % np.mean(y_predict == y_test))
   pd.crosstab(y_predict, y_test)

Accuracy rate is 0.80

Out[6]:

<table>
<thead>
<tr>
<th></th>
<th>CH</th>
<th>MM</th>
</tr>
</thead>
<tbody>
<tr>
<td>row_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>142</td>
<td>24</td>
</tr>
<tr>
<td>MM</td>
<td>30</td>
<td>74</td>
</tr>
</tbody>
</table>

1.1.5  e) Explore the different parameters you can pass to the tree command, such as splitter, min_samples_split, min_samples_leaf, min_impurity_split etc.

1.2  8.2 Random Forest

In this exercise we will use the email-spam data that has been presented in a couple of lectures. Go to the URL http://archive.ics.uci.edu/ml/datasets/Spambase for more information about the data.

1.2.1  a) Load the dataset Data/email.csv

In [7]: email = pd.read_csv('Data/email.csv')

1.2.2  (b) Create a training set containing a random sample of 75% of the observations, and a test set containing the remaining observations.

In [8]: np.random.seed(1)
   trainIndex = np.random.choice(email.shape[0], size=int(len(email)*.75), replace=False)
   train = email.iloc[trainIndex]  # training set
   test = email.iloc[-email.index.isin(trainIndex)]  # test set
1.2.3 (c)

Fit a classification tree with Class as output. Compute the test error.

In [9]:
X_train = train.copy().drop(columns=['Class'])
y_train = train['Class']
X_test = test.copy().drop(columns=['Class'])
y_test = test['Class']

In [10]: model = tree.DecisionTreeClassifier(max_leaf_nodes=5)
model.fit(X=X_train, y=y_train)
y_predict = model.predict(X_test)
print('Test error rate is %.3f' % np.mean(y_predict != y_test))

Test error rate is 0.111

1.2.4 (d)

Use the bagging approach to learn a classifier sklearn.ensemble.BaggingClassifier(). What test error do you get?

In [11]: model = BaggingClassifier()
model.fit(X_train, y_train)
error = np.mean(model.predict(X_test) != y_test)
print('Test error: %.3f' % error)

Test error: 0.063

1.2.5 (e)

Learn a random forest classifier using the function sklearn.ensemble.RandomForestClassifier(). What test error do you get?

In [12]: model = RandomForestClassifier()
model.fit(X_train, y_train)
error = np.mean(model.predict(X_test) != y_test)
print('Test error: %.3f' % error)

Test error: 0.048

1.3 8.3 Bootstrap

The bootstrap method has been introduced to reduce the variance in decision trees. However, the bootstrap method is widely applicable in other context as well, for example to estimate the variance in a parameter (cf. bias-variance tradeoff).
1.3.1 (a)

Generate $n = 100$ samples $\{y_i\}_{i=1}^n$ from $\mathcal{N}(4, 1^2)$.

In [13]:
np.random.seed(0)
    n = 100
    y = np.random.normal(4, 1, n)

1.3.2 (b)

We want to learn a model $y = \beta_0 + \epsilon$ (with $\mathbb{E}[\epsilon] = 0$) from the data $\{y_i\}_{i=1}^n$. Estimate $\beta_0$ with least squares, i.e. $\hat{\beta}_0 = \frac{1}{n} \sum_{i=1}^n y_i$.

In [14]: beta_0 = np.mean(y)
beta_0

Out[14]: 4.059808015534485

1.3.3 (c)

To estimate the variance in $\hat{\beta}_0$, $\text{Var}[\hat{\beta}_0]$ (a possible ‘quality measure’ of an estimator), repeat (a)-(b) 1000 times to get 1000 estimates of $\beta_0$, which we call $\hat{\beta}_0^1, \ldots, \hat{\beta}_0^{1000}$. Plot a histogram over the estimates $\hat{\beta}_0^1, \ldots, \hat{\beta}_0^{1000}$.

In [15]: beta = []
    for i in range(1000):
        beta.append(np.mean(np.random.normal(4, 1, n)))

plt.hist(beta, bins=14)
plt.show()
In practice we only have access to \( \{y_i\}_{i=1}^n \) (\( n = 100 \)) and cannot estimate \( \hat{\beta}_1, \ldots, \hat{\beta}_{1000} \) by generating new data. However, with the bootstrap method we can obtain ‘new’ data sets by sampling from the original data set \( \{y_i\}_{i=1}^n \).

1.3.4 (d)

Sample \( n \) indices \( \{i_1, i_2, \ldots, i_n\} \) with replacement from the set \( \{0, 1, \ldots, n - 1\} \) using the function \texttt{numpy.random.choice()}. Note that some indices will appear multiple times and some will not appear at all.

In [16]: indices = np.random.choice(np.arange(100), size=n, replace=True)
indices
Out[16]: array([77, 91, 78, 64, 51, 12, 89, 8, 33, 70, 35, 74, 84, 91, 74, 78, 75, 80, 40, 35, 97, 56, 16, 38, 86, 80, 98, 63, 11, 53, 32, 30, 75, 44, 7, 41, 97, 77, 60, 36, 63, 27, 61, 57, 66, 66, 48, 2, 25, 93, 56, 78, 33, 71, 64, 14, 68, 25, 28, 26, 76, 22, 30, 23, 51, 16, 24, 27, 74, 66, 80, 91, 43, 37, 6, 97, 91, 4, 83, 94, 62, 71, 1, 28, 64, 45, 19, 0, 72, 11, 86, 59, 90, 41, 13, 78, 25, 30, 48, 28])

1.3.5 (e)

Generate a ‘new’ data set \( \{y_{j}\}_{j=1}^n \) based on the indices generated in (d) and estimate \( \hat{\mu} \) from this data set.

In [17]: y_new = y[indices]

1.3.6 (f)

Repeat (d)-(e) 1000 times to get 1000 estimates of \( \hat{\mu} \), which we call \( \hat{\mu}_1^*, \ldots, \hat{\mu}_{1000}^* \). Plot a histogram over the estimates \( \hat{\mu}_1^*, \ldots, \hat{\mu}_{1000}^* \) and compare with the estimate achieved in (c).

In [18]: mu = []
    for i in range(1000):
        y_new = np.random.choice(y, size=n, replace=True)
        mu.append(np.mean(y_new))

    plt.hist(mu, bins=14)
    plt.show()
The distribution of the estimates very similar to the first histogram, but centered around 4.04 which was the mean of this data set. Consequently, we got a fairly accurate estimate of the uncertainty of the estimate without generating new data. For a linear estimator like this one, the uncertainty can be computed analytically (try to do that!), but in many nonlinear cases, this is not tractable or even possible. Then, the bootstrap method is very powerful.