Abalone

November 1, 2019

1 Abalone Data Set

We will be predicting the age of an abalone (number of rings) using various models.

1.1 Getting Data

We begin by loading the data.

[63]:
```
import numpy as np
import sklearn
import os
import pandas as pd
import matplotlib.pyplot as plt

cwd = os.getcwd()

ABALONE_PATH = os.path.join(cwd, "abalone.data")


def load_abalone_data(abalone_path = ABALONE_PATH):
    return pd.read_csv(abalone_path,  delimiter = ',',
                       names = ['sex',
                                'length',
                                'diameter',
                                'height',
                                'whole_weight',
                                'shucked_weight',
                                'viscera_weight',
                                'shell_weight',
                                'rings'])

data = load_abalone_data()

def create_dummy(data, column):
    dummy = pd.get_dummies(data[column])
    new_data = pd.concat([data, dummy], axis=1)
    return new_data
```

```
data.head()
```

[63]:
```
  sex  length  diameter  height  whole_weight  shucked_weight  viscera_weight  \
0   M   0.455     0.365   0.095        0.5140          0.2245          0.1010
1   M   0.350     0.265   0.090        0.2255          0.0995          0.0485
2   F   0.530     0.420   0.135        0.6770          0.2565          0.1415
3   M   0.440     0.365   0.125        0.5160          0.2155          0.1140
4   I   0.330     0.255   0.080        0.2050          0.0895          0.0395

   shell_weight  rings
0         0.150     15
1         0.070      7
2         0.210      9
3         0.155     10
4         0.055      7
```

[64]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4177 entries, 0 to 4176
Data columns (total 9 columns):
sex               4177 non-null object
length            4177 non-null float64
diameter          4177 non-null float64
height            4177 non-null float64
whole_weight      4177 non-null float64
shucked_weight    4177 non-null float64
viscera_weight    4177 non-null float64
shell_weight      4177 non-null float64
rings             4177 non-null int64
dtypes: float64(7), int64(1), object(1)
memory usage: 293.8+ KB
```

1.2 Data-Clean Up

We will clean up the training test data. First, we will convert the "sex" variable into a dummy variable.

[65]:
```
data = create_dummy(data, "sex")
data = data.drop("sex", axis=1)
data = data.drop("I", axis=1)
```

[66]:
```
data.describe()
```

[66]:
```
            length     diameter       height  whole_weight  shucked_weight  \
count  4177.000000  4177.000000  4177.000000   4177.000000     4177.000000
mean      0.523992     0.407881     0.139516      0.828742        0.359367
std       0.120093     0.099240     0.041827      0.490389        0.221963
min       0.075000     0.055000     0.000000      0.002000        0.001000
25%       0.450000     0.350000     0.115000      0.441500        0.186000
```

```
50%        0.545000      0.425000      0.140000       0.799500         0.336000
75%        0.615000      0.480000      0.165000       1.153000         0.502000
max        0.815000      0.650000      1.130000       2.825500         1.488000

       viscera_weight  shell_weight        rings            F            M
count     4177.000000   4177.000000  4177.000000  4177.000000  4177.000000
mean         0.180594      0.238831     9.933684     0.312904     0.365813
std          0.109614      0.139203     3.224169     0.463731     0.481715
min          0.000500      0.001500     1.000000     0.000000     0.000000
25%          0.093500      0.130000     8.000000     0.000000     0.000000
50%          0.171000      0.234000     9.000000     0.000000     0.000000
75%          0.253000      0.329000    11.000000     1.000000     1.000000
max          0.760000      1.005000    29.000000     1.000000     1.000000
```

We can see that some of our data is inaccurate. We see that the height of the min albaone is 0, which is impossible. We will throw out these inaccurate data points.

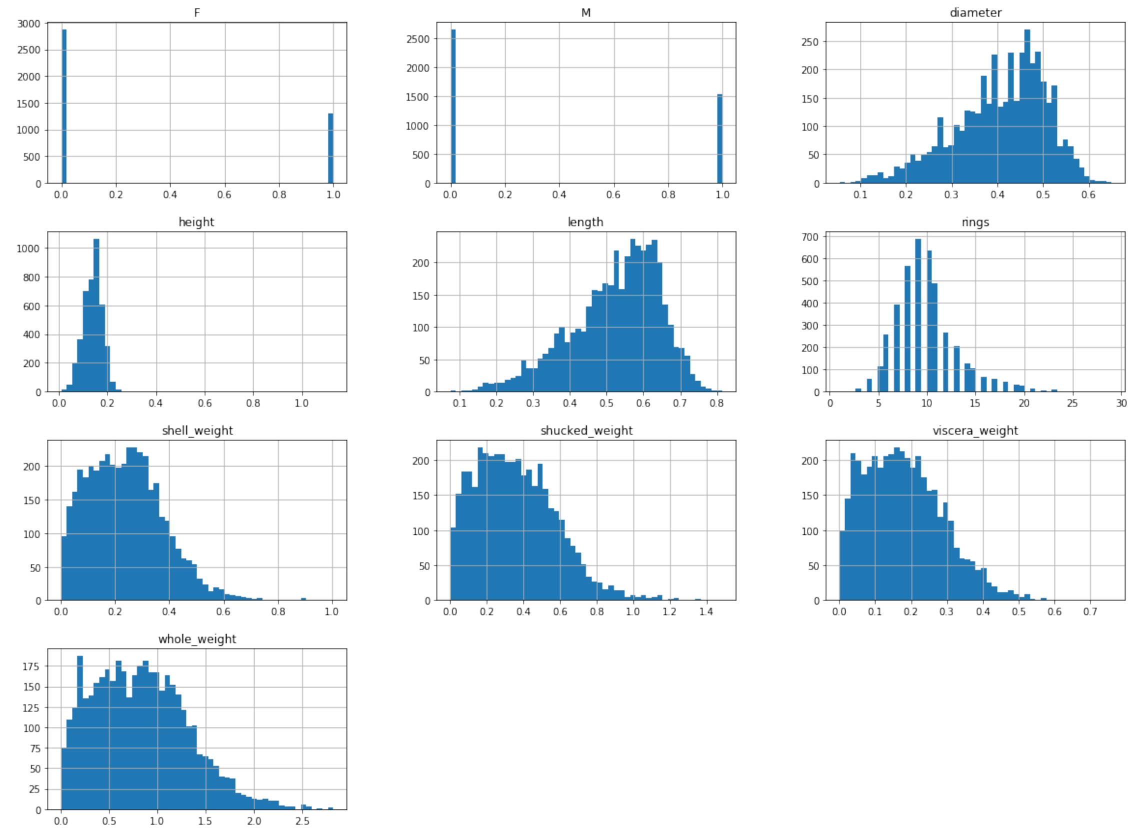
[67]:
```
data = data[data.height > 0]
```

Let's check for missing data. It seems like our data set is pretty clean.

[68]:
```
data.isnull().sum()
```

[68]:
```
length            0
diameter          0
height            0
whole_weight      0
shucked_weight    0
viscera_weight    0
shell_weight      0
rings             0
F                 0
M                 0
dtype: int64
```

[69]:
```
data.hist(bins=50, figsize=(20,15))
plt.show()
```



From the histograms and data description table above, we can observe that the weight data is generally skewed to the right. We will have to standardize this data to make it more normally distributed (later in the process).

1.3 Feature Selection

Now, we will see which features that we want to include in our models. From the correlation matrix below, we can see that all attributes except for female or male have a relatively positive correlation with age (rings).

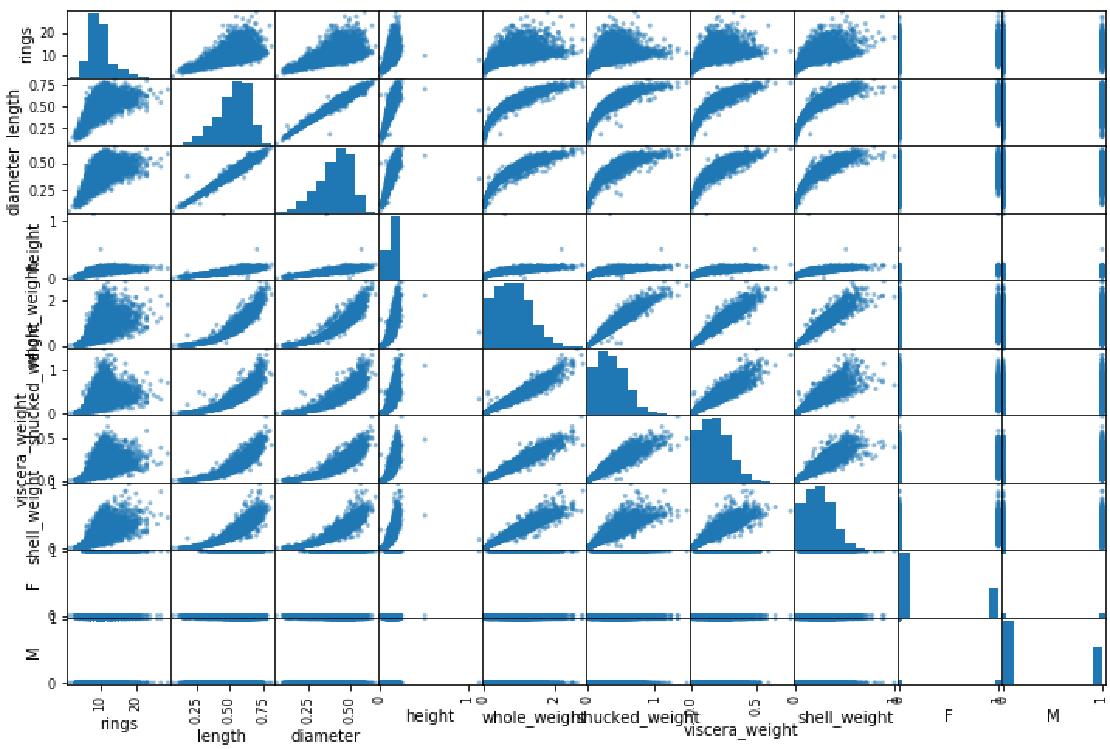
[70]:
```
corr_matrix = data.corr()
corr_matrix["rings"].sort_values(ascending=False)
```

[70]:
```
rings             1.000000
shell_weight      0.627928
diameter          0.574418
height            0.557625
length            0.556464
whole_weight      0.540151
viscera_weight    0.503562
shucked_weight    0.420597
F                 0.250067
```

```
M                    0.181565
Name: rings, dtype: float64
```

[71]:
```python
from pandas.plotting import scatter_matrix

attributes = ['rings',
              'length',
              'diameter',
              'height',
              'whole_weight',
              'shucked_weight',
              'viscera_weight',
              'shell_weight',
              'F',
              'M']
matrix = scatter_matrix(data[attributes], figsize=(12, 8))
```



We can also see from the scatter matrix that the sex of the abalone is a poor indicator of the number of rings (age). Thus, we will drop this from our regression

[72]:
```python
data = data.drop("M", axis=1)
data = data.drop("F", axis=1)
data.head()
```

[72]:
```
   length  diameter  height  whole_weight  shucked_weight  viscera_weight  \
0   0.455     0.365   0.095        0.5140          0.2245          0.1010
1   0.350     0.265   0.090        0.2255          0.0995          0.0485
2   0.530     0.420   0.135        0.6770          0.2565          0.1415
3   0.440     0.365   0.125        0.5160          0.2155          0.1140
4   0.330     0.255   0.080        0.2050          0.0895          0.0395

   shell_weight  rings
0         0.150     15
1         0.070      7
2         0.210      9
3         0.155     10
4         0.055      7
```

We also observe that 'viscera_weight' and 'shucked_weight' aren't as positively correlated with the number of rings as the other features. We will run two models, one with all features except for 'sex' included and one with all features except for 'sex', viscera_weight', 'shucked_weight'.

We now begin splitting the data.

1.4 Train-Test Split

Now we will split our data into training and testing sets

[73]:
```
from sklearn.model_selection import train_test_split
data_y = data["rings"]
data_x = data.drop(columns=['rings'])
data_x_modified = data.drop(columns=['rings', 'viscera_weight',␣
↪'shucked_weight'])



train, test = train_test_split(data, test_size=0.2, random_state=32)
```

[74]:
```
train_y = train["rings"]
train_x = train.drop(columns=['rings'])
train_x_removed_features = train.drop(columns=['rings',␣
↪'viscera_weight','shucked_weight'])

test_y = test["rings"]
test_x = test.drop(columns=['rings'])
test_x_removed_features = test.drop(columns=['rings', 'viscera_weight',␣
↪'shucked_weight'])
```

As discussed previously, our data is skewed. We will apply a standard scaler to scale and standardize the data.

[75]:
```
from sklearn.preprocessing import StandardScaler
```

```
process = StandardScaler()

train_x = process.fit_transform(train_x)
train_x_removed_features = process.fit_transform(train_x_removed_features)
```

1.5 Models

First, we'll write a function to evaluate all our models. This function returns the correct output within a certain acceptable range as well as the Mean Average Error.

[76]:
```
def evaluating_model(predictions, actual, acceptable_range):
    total = len(predictions)
    correct = 0
    mae = 0

    for i in range(total):
        difference = abs(predictions[i] - actual[i])
        if difference <= acceptable_range:
            correct += 1
        mae += difference

    percent_correct = (correct / total) * 100
    mae = mae / total

    print("{:0.2f}% correct within {} year(s) ({} correct guesses)".format(
            percent_correct, acceptable_range, correct))
    print("MAE: {:0.2f}".format(mae))
```

1.6 Linear Regressions

1.6.1 Model with All Features (except 'sex')

[77]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

lin_reg = LinearRegression()
lin_reg.fit(train_x, train_y)

pred_lin = lin_reg.predict(test_x)

lin_mse = mean_squared_error(test_y, pred_lin)
lin_rmse = np.sqrt(lin_mse)

print("Linear Reg 1 mse is {}, rmse is {}".format(lin_mse, lin_rmse))
evaluating_model(pred_lin, test_y.values, 1)
evaluating_model(pred_lin, test_y.values, 2)
```

```
Linear Reg 1 mse is 13.227892184109583, rmse is 3.6370169348120425
10.18% correct within 1 year(s) (85 correct guesses)
MAE: 3.25
23.23% correct within 2 year(s) (194 correct guesses)
MAE: 3.25
```

1.6.2 Model with Removals ('sex', viscera_weight', 'shucked_weight')

[78]:
```
lin_reg_2 = LinearRegression()
lin_reg_2.fit(train_x_removed_features, train_y)

pred_lin_2 = lin_reg_2.predict(test_x_removed_features)

lin_mse_2 = mean_squared_error(test_y, pred_lin_2)
lin_rmse_2 = np.sqrt(lin_mse_2)

print("Linear Reg 2 mse is {}, rmse is {}".format(lin_mse_2, lin_rmse_2))
evaluating_model(pred_lin_2, test_y.values, 1)
evaluating_model(pred_lin_2, test_y.values, 2)
```

```
Linear Reg 2 mse is 15.290392849482291, rmse is 3.9102931922660598
21.92% correct within 1 year(s) (183 correct guesses)
MAE: 2.92
44.55% correct within 2 year(s) (372 correct guesses)
MAE: 2.92
```

1.6.3 Trying Elastic Net Regression

[79]:
```
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
elastic_reg = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_reg.fit(train_x_removed_features, train_y)

pred_elastic = elastic_reg.predict(test_x_removed_features)

elastic_mse = mean_squared_error(test_y, pred_elastic)
elastic_rmse = np.sqrt(elastic_mse)

print("Elastic Reg mse is {}, rmse is {}".format(elastic_mse, elastic_rmse))
evaluating_model(pred_elastic, test_y.values, 1)
evaluating_model(pred_elastic, test_y.values, 2)
```

```
Elastic Reg mse is 10.051310651478637, rmse is 3.1703802061391055
27.78% correct within 1 year(s) (232 correct guesses)
MAE: 2.35
50.42% correct within 2 year(s) (421 correct guesses)
MAE: 2.35
```

1.7 Logistic Regressions

1.7.1 Model with All Features (except 'sex')

[80]:
```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(multi_class="auto",solver="lbfgs", max_iter=1000)
log_reg.fit(train_x, train_y)

pred_log = log_reg.predict(test_x)

log_mse = mean_squared_error(test_y, pred_log)
log_rmse = np.sqrt(log_mse)

print("Test mse is {}, rmse is {}".format(log_mse, log_rmse))
evaluating_model(pred_log, test_y.values, 1)
evaluating_model(pred_log, test_y.values, 2)
```

```
Test mse is 9.178443113772454, rmse is 3.029594546102243
41.68% correct within 1 year(s) (348 correct guesses)
MAE: 2.26
64.55% correct within 2 year(s) (539 correct guesses)
MAE: 2.26
```

1.7.2 Model with Removals ('sex', viscera_weight', 'shucked_weight')

[81]:
```
log_reg_2 = LogisticRegression(multi_class="auto",solver="liblinear",␣
↪max_iter=500)
log_reg_2.fit(train_x_removed_features, train_y)

pred_log_2 = log_reg_2.predict(test_x_removed_features)

log_mse_2 = mean_squared_error(test_y, pred_log_2)
log_rmse_2 = np.sqrt(log_mse_2)

print("Test mse is {}, rmse is {}".format(log_mse_2, log_rmse_2))
evaluating_model(pred_log_2, test_y.values, 1)
evaluating_model(pred_log_2, test_y.values, 2)
```

```
Test mse is 9.487425149700599, rmse is 3.0801664159101207
50.30% correct within 1 year(s) (420 correct guesses)
MAE: 2.14
69.70% correct within 2 year(s) (582 correct guesses)
MAE: 2.14
```

1.8 Decision Trees

Now, let's try using a decision tree. After trial and error, it seems like a max_depth of 5 minimizes the errors.

[82]:
```
from sklearn.tree import DecisionTreeRegressor
tree = DecisionTreeRegressor(max_depth=5, criterion='mae')
tree.fit(train_x, train_y)
pred_tree = tree.predict(test_x)

tree_mse = mean_squared_error(pred_tree, test_y)
tree_rmse = np.sqrt(tree_mse)

print("Tree mse is {}, rmse is {}".format(tree_mse, tree_rmse))
evaluating_model(pred_tree, test_y.values, 1)
evaluating_model(pred_tree, test_y.values, 2)
```

```
Tree mse is 9.480239520958083, rmse is 3.078999759817802
45.15% correct within 1 year(s) (377 correct guesses)
MAE: 2.19
69.58% correct within 2 year(s) (581 correct guesses)
MAE: 2.19
```

[83]:
```
tree_2 = DecisionTreeRegressor(max_depth=5, criterion='mae')
tree_2.fit(train_x_removed_features, train_y)
pred_tree_2 = tree_2.predict(test_x_removed_features)

tree_mse_2 = mean_squared_error(pred_tree_2, test_y)
tree_rmse_2 = np.sqrt(tree_mse_2)

print("Tree mse is {}, rmse is {}".format(tree_mse_2, tree_rmse_2))
evaluating_model(pred_tree_2, test_y.values, 1)
evaluating_model(pred_tree_2, test_y.values, 2)
```

```
Tree mse is 9.190419161676648, rmse is 3.031570411795947
45.99% correct within 1 year(s) (384 correct guesses)
MAE: 2.16
69.82% correct within 2 year(s) (583 correct guesses)
MAE: 2.16
```

1.9 Gradient Boosted Trees

Trying Gradient boosted trees on different learning rates (referenced https://stackabuse.com/gradient-boosting-classifiers-in-python-with-scikit-learn/) to learn how to test the different learning rates.

[84]:
```
from sklearn.ensemble import GradientBoostingRegressor
learning_rate_list = [0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1]

for learning_rate in learning_rate_list:
    gb_tree = GradientBoostingRegressor(n_estimators=20,␣
 ↪learning_rate=learning_rate, max_features=2, max_depth=2, random_state=0,␣
 ↪criterion='mae')
```

```
    gb_tree.fit(train_x, train_y)
    pred1 = gb_tree.predict(train_x)
    pred2 = gb_tree.predict(test_x)

    print("Learning rate: ", learning_rate)
    evaluating_model(pred1, train_y.values, 2)
    evaluating_model(pred2, test_y.values, 2)
```

```
Learning rate:  0.05
70.09% correct within 2 year(s) (2341 correct guesses)
MAE: 1.86
59.04% correct within 2 year(s) (493 correct guesses)
MAE: 2.30
Learning rate:  0.075
73.20% correct within 2 year(s) (2445 correct guesses)
MAE: 1.76
52.34% correct within 2 year(s) (437 correct guesses)
MAE: 2.30
Learning rate:  0.1
74.13% correct within 2 year(s) (2476 correct guesses)
MAE: 1.70
56.41% correct within 2 year(s) (471 correct guesses)
MAE: 2.26
Learning rate:  0.25
75.96% correct within 2 year(s) (2537 correct guesses)
MAE: 1.58
57.13% correct within 2 year(s) (477 correct guesses)
MAE: 2.21
Learning rate:  0.5
75.21% correct within 2 year(s) (2512 correct guesses)
MAE: 1.56
53.53% correct within 2 year(s) (447 correct guesses)
MAE: 2.18
Learning rate:  0.75
76.71% correct within 2 year(s) (2562 correct guesses)
MAE: 1.51
56.89% correct within 2 year(s) (475 correct guesses)
MAE: 2.22
Learning rate:  1
81.17% correct within 2 year(s) (2711 correct guesses)
MAE: 1.64
61.68% correct within 2 year(s) (515 correct guesses)
MAE: 2.29
```

We see above that a learning rate of 0.5 provides a good prediction.

[85]:
```
gb_tree = GradientBoostingRegressor(n_estimators=20, learning_rate=0.25,␣
↪max_features=2, max_depth=2, random_state=0, criterion='mae')
gb_tree.fit(train_x, train_y)
pred = gb_tree.predict(test_x)

gb_tree_mse = mean_squared_error(pred, test_y)
gb_tree_rmse = np.sqrt(gb_tree_mse)

print("Learning rate: ", 0.25)
evaluating_model(pred, test_y.values, 1)
evaluating_model(pred, test_y.values, 2)
```

```
Learning rate:  0.25
32.57% correct within 1 year(s) (272 correct guesses)
MAE: 2.21
57.13% correct within 2 year(s) (477 correct guesses)
MAE: 2.21
```

1.9.1 Baseline

Let's compare our model to the baseline (all predictions will be the average of the rings in the training data)

[86]:
```
# Using the average number of rings found in training data and using
# that as all predictions.
avg_y = train_y.mean()

# Creating an array of size of the test data filled with the
# average ring number obtained above.
avg_y_predictions = np.full(len(test_y), fill_value=avg_y)
avg_mse = mean_squared_error(avg_y_predictions, test_y)
avg_rmse = np.sqrt(avg_mse)

models = [["Baseline", avg_mse, avg_rmse, avg_y_predictions],
          ["Linear Reg 1", lin_mse, lin_rmse, pred_lin],
          ["Linear Reg 2", lin_mse_2, lin_rmse_2, pred_lin_2],
          ["Elastic Reg", elastic_mse, elastic_rmse, pred_elastic],
          ["Logistic Reg 1", log_mse, log_rmse, pred_log],
          ["Logistic Reg 2", log_mse_2, log_rmse_2, pred_log_2],
          ["Decision Tree 1", tree_mse, tree_rmse, pred_tree],
          ["Decision Tree 2", tree_mse_2, tree_rmse_2, pred_tree_2],
          ["Gradient Boosted Tree", gb_tree_mse, gb_tree_rmse, pred]]

for model in models:
    print("{} mse is {:0.2f}, rmse is {:0.2f}".format(
        model[0], model[1], model[2]))
    evaluating_model(model[3], test_y.values, 1)
    evaluating_model(model[3], test_y.values, 2)
```

```python
    print()
```

```
Baseline mse is 10.55, rmse is 3.25
30.90% correct within 1 year(s) (258 correct guesses)
MAE: 2.35
57.49% correct within 2 year(s) (480 correct guesses)
MAE: 2.35

Linear Reg 1 mse is 13.23, rmse is 3.64
10.18% correct within 1 year(s) (85 correct guesses)
MAE: 3.25
23.23% correct within 2 year(s) (194 correct guesses)
MAE: 3.25

Linear Reg 2 mse is 15.29, rmse is 3.91
21.92% correct within 1 year(s) (183 correct guesses)
MAE: 2.92
44.55% correct within 2 year(s) (372 correct guesses)
MAE: 2.92

Elastic Reg mse is 10.05, rmse is 3.17
27.78% correct within 1 year(s) (232 correct guesses)
MAE: 2.35
50.42% correct within 2 year(s) (421 correct guesses)
MAE: 2.35

Logistic Reg 1 mse is 9.18, rmse is 3.03
41.68% correct within 1 year(s) (348 correct guesses)
MAE: 2.26
64.55% correct within 2 year(s) (539 correct guesses)
MAE: 2.26

Logistic Reg 2 mse is 9.49, rmse is 3.08
50.30% correct within 1 year(s) (420 correct guesses)
MAE: 2.14
69.70% correct within 2 year(s) (582 correct guesses)
MAE: 2.14

Decision Tree 1 mse is 9.48, rmse is 3.08
45.15% correct within 1 year(s) (377 correct guesses)
MAE: 2.19
69.58% correct within 2 year(s) (581 correct guesses)
MAE: 2.19

Decision Tree 2 mse is 9.19, rmse is 3.03
45.99% correct within 1 year(s) (384 correct guesses)
MAE: 2.16
```

```
69.82% correct within 2 year(s) (583 correct guesses)
MAE: 2.16

Gradient Boosted Tree mse is 9.47, rmse is 3.08
32.57% correct within 1 year(s) (272 correct guesses)
MAE: 2.21
57.13% correct within 2 year(s) (477 correct guesses)
MAE: 2.21
```

1.10 Cross-Validation

Let's also test our models with cross-validation (k-folds)

[89]:
```python
from sklearn.model_selection import cross_val_score

process = StandardScaler()

data_x = process.fit_transform(data_x)


models = [["Linear Reg 1", lin_reg, data_x],
          ["Linear Reg 2", lin_reg_2, data_x_modified],
          ["Elastic Reg", elastic_reg, data_x],
          ["Logistic Reg 1", log_reg, data_x],
          ["Logistic Reg 2", log_reg_2, data_x_modified],
          ["Decision Tree 1", tree, data_x],
          ["Decision Tree 2", tree_2, data_x_modified],
          ["Gradient Boosted Tree", gb_tree, data_x]]

def cross_validate_model(models):
    for model in models:
        scores = cross_val_score(model[1], model[2], data_y,
                scoring='neg_mean_squared_error', cv=10)

        rmse_scores = np.sqrt(-scores)

        print("Model: {}".format(model[0]))
        print("RMSE:", rmse_scores)
        print("RMSE Mean:", rmse_scores.mean())
        print("Standard deviation:", rmse_scores.std())
        print()

cross_validate_model(models)
```

```
Model: Linear Reg 1
RMSE: [2.62310307 3.42355378 1.74335408 1.71213992 2.06109912 2.93530924
 1.60409556 2.29042571 1.92739841 2.09038944]
RMSE Mean: 2.2410868329189464
```

```
Standard deviation: 0.558151400625446

Model: Linear Reg 2
RMSE: [2.82703037 3.79805477 1.75337714 1.7876121  2.1958502  3.18574085
 1.59566297 2.36595974 2.09701444 2.20884221]
RMSE Mean: 2.381514480058976
Standard deviation: 0.660466437769354

Model: Elastic Reg
RMSE: [2.8466727  3.7864444  1.75382563 1.64143738 1.97392138 3.24835423
 1.56333105 2.33500034 1.99113957 2.11133907]
RMSE Mean: 2.325146575064941
Standard deviation: 0.7010458293447425


/Users/gp/anaconda3/lib/python3.7/site-
packages/sklearn/model_selection/_split.py:657: Warning: The least populated
class in y has only 1 members, which is too few. The minimum number of members
in any class cannot be less than n_splits=10.
  % (min_groups, self.n_splits)), Warning)

Model: Logistic Reg 1
RMSE: [2.76191519 2.62263611 2.63141595 2.66141605 2.42156185 2.60694516
 2.30311698 2.46441669 2.35696469 2.30726175]
RMSE Mean: 2.5137650422264497
Standard deviation: 0.1550147563300376


/Users/gp/anaconda3/lib/python3.7/site-
packages/sklearn/model_selection/_split.py:657: Warning: The least populated
class in y has only 1 members, which is too few. The minimum number of members
in any class cannot be less than n_splits=10.
  % (min_groups, self.n_splits)), Warning)

Model: Logistic Reg 2
RMSE: [3.38349478 3.12727669 3.03174692 2.88167278 2.73970192 2.91979106
 2.64803272 2.72316873 2.67346594 2.61713254]
RMSE Mean: 2.8745484075737124
Standard deviation: 0.23426755025588503

Model: Decision Tree 1
RMSE: [3.26629152 3.97179048 1.49760574 1.46897745 1.55770196 3.62611101
 1.29192289 2.70945144 1.91751363 2.26920263]
RMSE Mean: 2.3576568747148814
Standard deviation: 0.9311544241348121

Model: Decision Tree 2
RMSE: [3.3980363  4.17177518 1.58416204 1.43040296 1.55173926 3.8262629
 1.53222773 2.77190426 1.89281109 2.25688395]
```

```
RMSE Mean: 2.4416205680568885
Standard deviation: 0.9821653342000862

Model: Gradient Boosted Tree
RMSE: [3.319592   4.22966549 1.45420871 1.28469782 1.35642666 3.73767582
 1.29570019 2.63190338 1.95983125 2.15907785]
RMSE Mean: 2.342877918198855
Standard deviation: 1.0345330923947433
```

1.11 Results

1.11.1 Train-Test Split Result

We can see from the models above that generally logistic regression models make the best predictions. The Logistic Regression model that removed the less correlated features ('sex', viscera_weight', 'shucked_weight') performed the best out of all the models.

We can also see that the Decision Tree model worked well with the model with the features removed ('sex', viscera_weight', 'shucked_weight') and beat the baseline. However, the Decision Tree model that included all features did not beat the baseline (although it was very close).

Finally, the Gradient Boosted Tree and Linear Regression models performed worse than or similarily to the baseline in both cases, with the Linear Regression performing the worst.

1.11.2 Cross-Validation Result

However, when we validate our models using the cross-validation k-folds method, we see that the Linear Regression models as well as the Gradient Boosted Tree performed the best. This result implies that our test-train split models were overfitting our training data. This particularly occured because this data set was farily small.