Regression Intro: Transforming Target

- Applying transformations to make target variable more normally distributed for regression
- Applying inverse transformations to be able to use these in a regression context

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

Loading in Boston Data

In [2]:
```
#loading boston dataset from sklearn
from sklearn.datasets import load_boston
boston = load_boston()
boston_data = pd.DataFrame(boston.data, columns=boston.feature_names) # Creates dataframe with the feature names as columns
boston_data['MEDV'] = boston.target # Appends the MEDV or target onto the dataframe
boston_description = boston['DESCR'] # Description of the columns
```

Determining Normality

Making our target variable normally distributed often will lead to better results

If our target is not normally distributed, we can apply a transformation to it and then fit our regression to predict the transformed values.

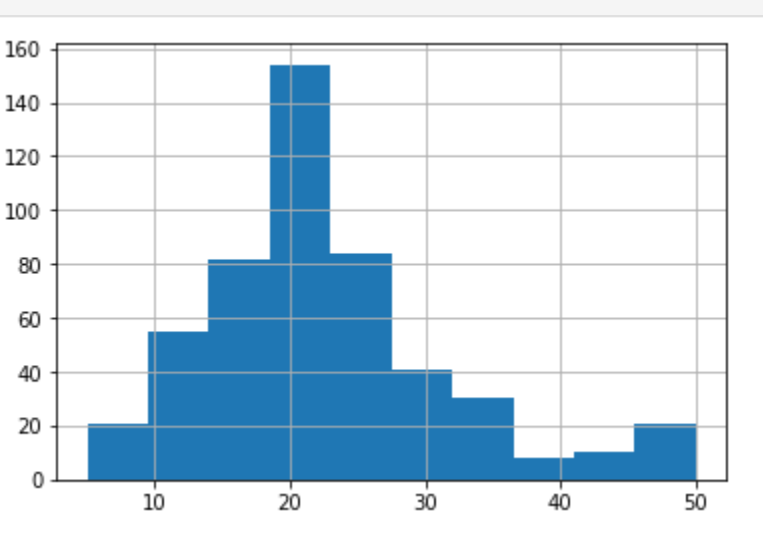
How can we tell if our target is normally distributed? There are two ways:

- Visually
- Using a statistical test

Visually

Plotting a histogram:

In [3]:
```
boston_data.MEDV.hist();
```



Does not look normal due to that right tail. Let's try to verify statistically:

In [6]:
```
from scipy.stats.mstats import normaltest # D'Agostino K^2 Test
```

Without getting into Bayesian vs. frequentist debates, for the purposes of this lesson, the following will suffice:

- This is a statistical test that tests whether a distribution is normally distributed or not. It isn't perfect, but suffice it to say:
  - This test outputs a "p-value". The *higher* this p-value is the *closer* the distribution is to normal.
  - Frequentist statisticians would say that you accept that the distribution is normal (more specifically: fail to reject the null hypothesis that it is normal) if p > 0.05.

In [7]:
```
normaltest(boston_data.MEDV.values)
```

Out[7]:
```
NormaltestResult(statistic=90.9746873700967, pvalue=1.7583188871696095e-20)
```

p-value *extremely* low. Our y variable we've been dealing with this whole time was not normally distributed!

Linear Regression assumes a normally distributed residuals which can be aided by transforming y variable. Let's try some common transformations to try and get y to be normally distributed:

- Log
- Square root
- Box cox

The log transform can transform data that is significantly skewed right to be more normally distributed:

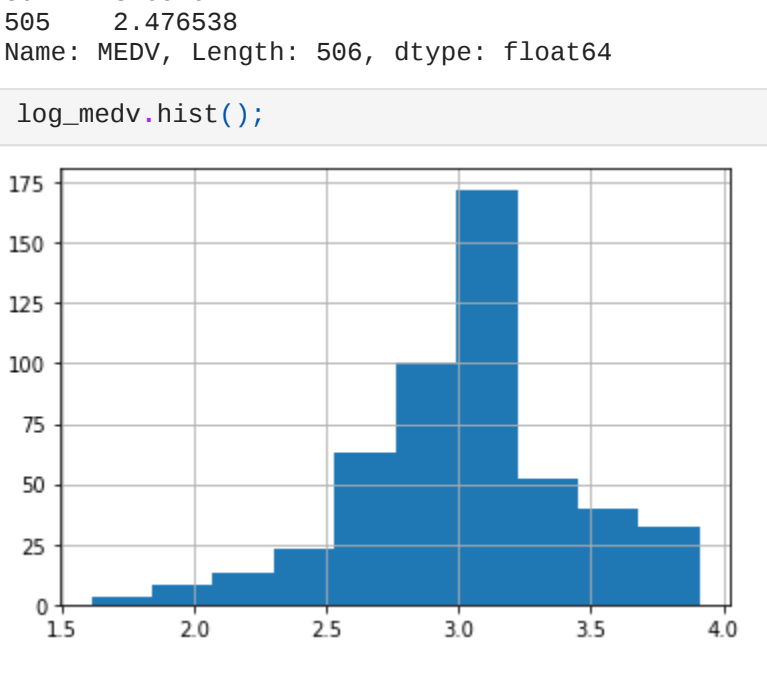
Apply transform to Boston data:

In [12]:
```
log_medv = np.log(boston_data.MEDV)
```

In [13]:
```
log_medv
```

Out[13]:
```
0      3.178054
1      3.072693
2      3.546740
3      3.508556
4      3.589059
         ...
501    3.109061
502    3.025291
503    3.173878
504    3.091042
505    2.476538
Name: MEDV, Length: 506, dtype: float64
```

In [14]:
```
log_medv.hist();
```



In [15]:
```
normaltest(log_medv)
```

Out[15]:
```
NormaltestResult(statistic=17.21801696640697, pvalue=0.00018245472768345307)
```

Conclusion: closer, but still not normal.

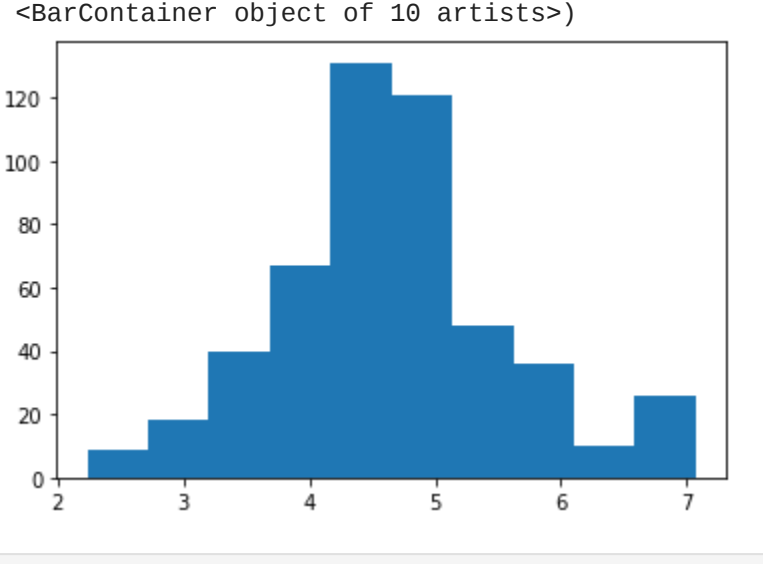
Slightly skewed right.

Apply the square root transformation to the Boston data target and test whether the result is normally distributed.

In [20]:
```
# Instructor Solution

sqrt_medv = np.sqrt(boston_data.MEDV)
plt.hist(sqrt_medv)
```

Out[20]:
```
(array([  9.,  18.,  40.,  67., 131., 121.,  48.,  36.,  10.,  26.]),
 array([2.23606798, 2.71956796, 3.20306794, 3.68656793, 4.17006791,
        4.65356789, 5.13706788, 5.62056786, 6.10406784, 6.58756783,
        7.07106781]),
 <BarContainer object of 10 artists>)
```



In [21]:
```
normaltest(sqrt_medv)
```

Out[21]:
```
NormaltestResult(statistic=20.487090826863067, pvalue=3.558645701429252e-05)
```

Box cox

The box cox transformation is a parametrized transformation that tries to get distributions "as close to a normal distribution as possible".

It is defined as:

$$\text{boxcox}(y_i) = \frac{y_i^\lambda - 1}{\lambda}$$

You can think of as a generalization of the square root function: the square root function uses the exponent of 0.5, but box cox lets its exponent vary so it can find the best one.

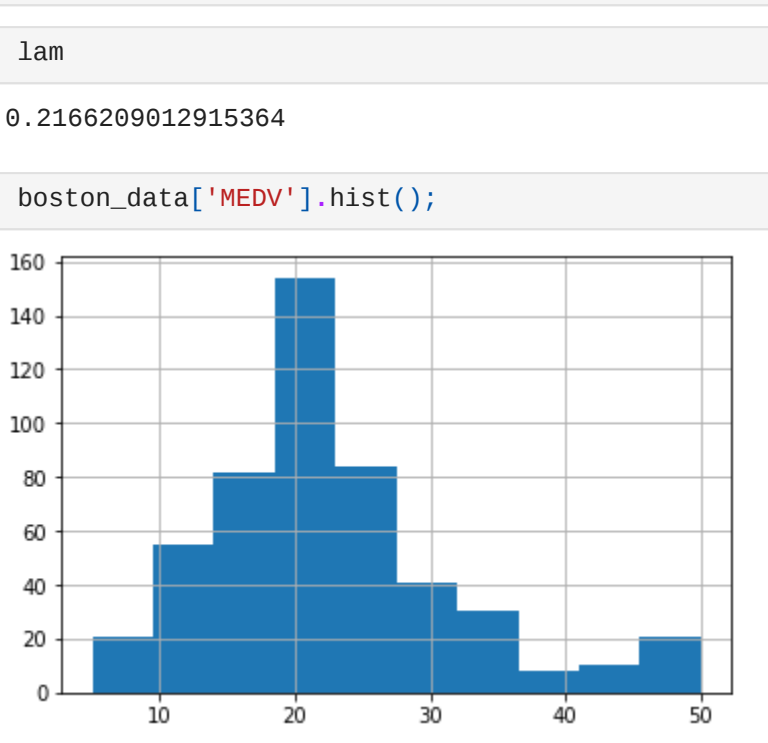
In [22]:
```
from scipy.stats import boxcox
```

In [25]:
```
bc_result = boxcox(boston_data.MEDV)
boxcox_medv = bc_result[0]
lam = bc_result[1]
```

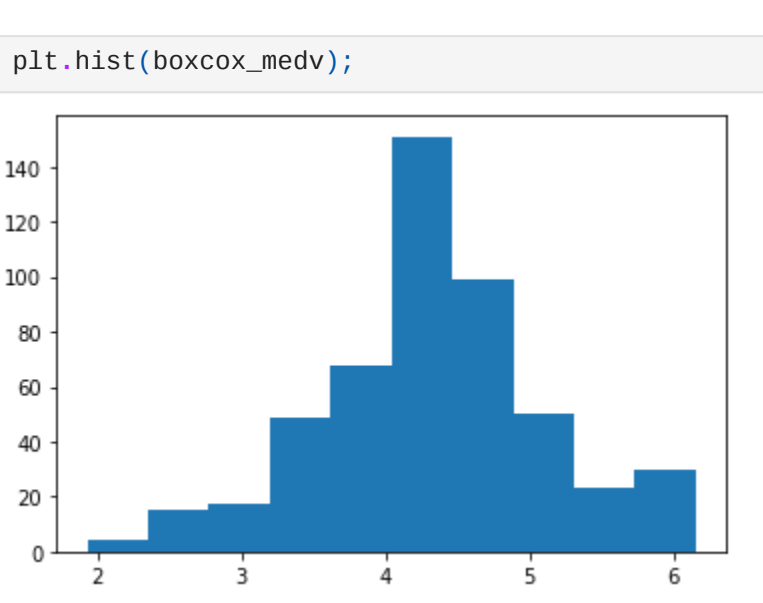
In [24]:
```
lam
```

Out[24]:
```
0.2166209012915364
```

In [30]:
```
boston_data['MEDV'].hist();
```



In [31]:
```
plt.hist(boxcox_medv);
```



In [32]:
```
normaltest(boxcox_medv)
```

Out[32]:
```
NormaltestResult(statistic=4.513528775533045, pvalue=0.1046886692817602)
```

Significantly more normally distributed (according to p value) than the other two distributions - above 0.05, even!

Now that we have a normally distributed y-variable, let's try a regression!

Testing regression:

In [33]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import (StandardScaler,
                                   PolynomialFeatures)
```

In [34]:
```
lr = LinearRegression( )
```

Reload clean version of `boston_data`:

In [37]:
```
from sklearn.datasets import load_boston
boston = load_boston()
boston_data = pd.DataFrame(boston.data, columns=boston.feature_names) # Creates dataframe with the feature names as columns
boston_data['MEDV'] = boston.target # Appends the MEDV or target onto the dataframe
boston_description = boston['DESCR'] # Description of the columns
```

Same steps as before.

Create X and y

In [38]:
```
y_col = "MEDV"

X = boston_data.drop(y_col, axis=1)
y = boston_data[y_col]
```

In [48]:
```
X
```

Out[48]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33
...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0.0	0.573	6.593	69.1	2.4786	1.0	273.0	21.0	391.99	9.67
502	0.04527	0.0	11.93	0.0	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	396.90	9.08
503	0.06076	0.0	11.93	0.0	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	396.90	5.64
504	0.10959	0.0	11.93	0.0	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	393.45	6.48
505	0.04741	0.0	11.93	0.0	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	396.90	7.88

506 rows × 13 columns

In [49]:
```
y
```

Out[49]:
```
0      24.0
1      21.6
2      34.7
3      33.4
4      36.2
       ...
501    22.4
502    20.6
503    23.9
504    22.0
505    11.9
Name: MEDV, Length: 506, dtype: float64
```

Create Polynomial Features

In [46]:
```
pf = PolynomialFeatures(degree=2, include_bias=False)
X_pf = pf.fit_transform(X)
```

In [47]:
```
X_pf
```

Out[47]:
```
array([[6.32000000e-03, 1.80000000e+01, 2.31000000e+00, ...,
         1.57529610e+05, 1.97656200e+03, 2.48004000e+01],
       [2.73100000e-02, 0.00000000e+00, 7.07000000e+00, ...,
         1.57529610e+05, 3.62766600e+03, 8.35396000e+01],
       [2.72900000e-02, 0.00000000e+00, 7.07000000e+00, ...,
         1.54315409e+05, 1.58310490e+03, 1.62409000e+01],
       ...,
       [6.07600000e-02, 0.00000000e+00, 1.19300000e+01, ...,
         1.57529610e+05, 2.23851600e+03, 3.18096000e+01],
       [1.09590000e-01, 0.00000000e+00, 1.19300000e+01, ...,
         1.54802902e+05, 2.54955600e+03, 4.19904000e+01],
       [4.74100000e-02, 0.00000000e+00, 1.19300000e+01, ...,
         1.57529610e+05, 3.12757200e+03, 6.20944000e+01]])
```

In [51]:
```
X.shape
```

Out[51]:
```
(506, 13)
```

In [52]:
```
X_pf.shape
```

Out[52]:
```
(506, 104)
```

Train test split

In [45]:
```
X_train, X_test, y_train, y_test = train_test_split(X_pf, y, test_size=0.3,
                                                    random_state=72018)
```

In [54]:
```
X_train.shape # X_train is 70% OF OUR x_pf
```

Out[54]:
```
(354, 104)
```

In [55]:
```
X_test.shape
```

Out[55]:
```
(152, 104)
```

Fit `StandardScaler` on `X_train` as before

we're going to use the StandardScaler, which is another transformer object. We set that equal to s. Again, we used the fit transform method and we pass in x_train, and that will calculate, given our x_train, those 354 rows and 104 columns. For each one of those columns, what is the mean value, what is the standard deviation, and subtract the mean and divide by that standard deviation. Then, we set the standardized version of x_train equal to x_train_s. That should have the same shape as before. Just now, the variables rather than being on different scales, they're now all on the same scale.

In [56]:
```
s = StandardScaler()
X_train_s = s.fit_transform(X_train)
```

In [57]:
```
X_train_s.shape
```

Out[57]:
```
(354, 104)
```

what transformation do we need to apply next?

Apply the appropriate transformation.

So we're going to use the boxcox that we defined earlier. We pass in y_train, which is the subset of just 354 rows, it's just one column because the y is just one column. We only want to test on our y_train, we're only learning our parameters from our x_train and y_train, so we're only going to do our transformations there first. From that, remember when we run boxcox, we get two outputs, both the actual transformed values, which is going to be index 0 of our output here, as well as the Lambda value that we have learned that parameter that we learned given our data in order to come up with the normal distribution. So we run this and we have our new transform version of the outcome variable, as well as lam2, which is just going to be for that trainingset what the Lambda value, you see it's slightly different than before because we're only working on a subset of the data in order to come up with a normal distribution.

In [59]:
```
# Instructor Solution
bc_result2 = boxcox(y_train)
y_train_bc = bc_result2[0]
lam2 = bc_result2[1]
```

In [61]:
```
y_train_bc
```

Out[61]:
```
array([4.8625686 , 6.0629177 , 4.51310411, 4.81957899, 4.95725428,
       5.77277497, 5.41197088, 5.75640624, 3.72665146, 4.94686892,
       5.75640624, 4.80874112, 3.92225296, 4.58384068, 4.78695477,
       6.64804017, 3.77300722, 2.91446761, 3.99405229, 5.37557157,
       2.91446761, 5.14858214, 3.37830364, 5.61413692, 4.46503222,
       6.62861276, 3.53307611, 3.78829063, 4.6643901 , 4.74293114,
       4.76501893, 4.58384068, 6.66093826, 4.90499278, 4.72068886,
       5.1583522 , 3.30653002, 3.15681299, 4.6070669 , 6.0021158 ,
       7.02952208, 4.6870301 , 3.78829063, 5.05933485, 4.19940562,
       4.93645035, 6.30368143, 7.02952208, 5.65657858, 4.54867373,
       4.59547537, 4.6070669 , 4.8411456 , 5.37557157, 3.93675676,
       4.22573738, 4.6070669 , 4.47712066, 4.54867373, 5.61413692,
       4.93645035, 2.39638821, 4.88385027, 3.6795137 , 5.08935167,
       4.78695477, 6.22513995, 4.92599831, 6.02503476, 4.10535832,
       4.4284788 , 4.59547537, 5.51012859, 3.51631095, 5.05933485,
       4.54867373, 5.90897289, 4.58384068, 4.05023615, 4.58384068,
       7.02952208, 5.45693428, 4.87322696, 4.47712066, 7.02952208,
       4.65300883, 5.01888279, 3.11799245, 7.02952208, 4.57216244,
       5.06937077, 5.10921274, 2.02481765, 4.54867373, 4.63012196,
       4.8625686 , 2.87170159, 4.96760668, 4.65300883, 4.98821357,
       5.89321422, 3.78829063, 4.69828951, 5.08935167, 4.6870301 ,
       5.17780698, 3.78829063, 2.97726946, 4.8411456 , 5.76459999,
       4.58384068, 4.75399418, 4.40386518, 3.69531512, 5.1583522 ,
       3.80349159, 4.77600569, 6.02503476, 4.39148356, 5.32023533,
       5.59701568, 2.82818197, 5.51891547, 2.02481765, 5.69851559,
       4.47712066, 4.41619677, 4.8625686 , 5.17780698, 4.02227997,
       5.50131941, 4.57216244, 4.31611532, 3.25138802, 5.465857  ,
       5.63966328, 5.55384267, 6.04781136, 4.61861563, 4.8411456 ,
       3.6795137 , 4.44071174, 4.59547537, 4.05023615, 6.1375352 ,
       4.05023615, 5.89321422, 4.70950898, 6.12273165, 2.69276611,
       4.83038039, 4.87322696, 6.19616649, 5.06937077, 5.03917094,
       4.44071174, 6.48309287, 4.89443878, 4.99846855, 5.51891547,
       4.93645035, 3.37830364, 4.98821357, 4.64158623, 5.07937631,
       5.19714927, 4.97792636, 5.12895568, 7.02952208, 3.48246652,
       7.02952208, 5.8133718 , 4.17283862, 6.12273165, 4.92599831,
       4.30336941, 3.74218874, 4.64158623, 5.31092456, 5.03917094,
       4.37905141, 4.59547537, 4.00820046, 5.13878337, 4.75399418,
       4.22573738, 4.6870301 , 5.54514366, 4.65300883, 6.20343095,
       6.0629177 , 5.33878073, 2.62209382, 5.04926831, 7.02952208,
       5.90897289, 4.96760668, 5.4924878 , 5.63966328, 5.41197088,
       4.56044025, 3.53307611, 4.63012196, 5.10921274, 5.13878337,
       4.6643901 , 4.52500607, 4.76501893, 5.06937077, 4.15946534,
       6.75005519, 4.97792636, 4.10535832, 4.31611532, 4.50115617,
       4.74293114, 6.71210964, 5.55384267, 4.95725428, 6.14491488,
       4.87322696, 5.02904251, 2.95651035, 3.89302335, 4.6643901 ,
       4.63012196, 3.95118773, 4.56044025, 4.67573041, 4.45289606,
       3.74218874, 4.83038039, 5.32952066, 4.80874112, 4.53686248,
       3.81861116, 4.80874112, 4.95725428, 3.6795137 , 5.69016788,
       4.25183879, 4.70950898, 4.70950898, 4.26480461, 3.66362184,
       5.04926831, 4.94686892, 3.90767545, 4.72068886, 4.10535832,
       3.44819273, 5.67341314, 7.02952208, 3.84861019, 4.17283862,
       6.38062104, 4.46503222, 4.85187491, 5.11909887, 7.02952208,
       6.83104027, 4.6070669 , 4.87322696, 4.45289606, 4.69828951,
       2.62209382, 5.44798843, 6.5829015 , 4.50115617, 4.53686248,
       4.99846855, 4.47712066, 4.87322696, 5.10921274, 4.06411435,
       4.72068886, 7.02952208, 5.14858214, 3.95118773, 4.67573041,
       2.62209382, 4.98821357, 4.02227997, 4.00820046, 4.29056935,
       3.6476383 , 4.57216244, 4.34144687, 3.84861019, 4.97792636,
       4.10535832, 4.46503222, 3.58276147, 6.54991774, 5.32952066,
       5.82948262, 4.6870301 , 3.84861019, 4.69828951, 4.77600569,
       4.48916183, 4.8411456 , 4.74293114, 5.00869155, 5.17780698,
       3.3605416 , 2.89317686, 4.87322696, 4.57216244, 3.28827772,
       4.17283862, 5.36641096, 4.97792636, 5.17780698, 4.72068886,
       3.96554675, 4.25183879, 4.98821357, 5.06937077, 5.11909887,
       5.29222605, 5.41197088, 3.93675676, 4.23881658, 5.05933485,
       4.50115617, 4.91551254, 4.91551254, 6.04023484, 5.71515227,
       4.8411456 , 4.26480461, 5.12895568, 4.34144687, 6.1078687 ,
       5.17780698, 3.81861116, 5.73171132, 5.86944667, 2.20407036,
       3.28827772, 4.89443878, 5.83751101, 5.17780698, 5.85351424,
       5.04926831, 4.78695477, 4.36656825, 4.39148356, 4.81957899,
       5.47475673, 5.20677872, 4.02227997, 4.34144687])
```

As before, we'll now:

1. Fit regression
2. Transform testing data
3. Predict on testing data

We should have 354 different values for our y that should match up with our x_train. Now, we have the x_train that we ultimately want as well as our transformed y-variable, and we are going to fit using our linear regression that we have defined before. If you recall all the way up here, we defined, look at the top of the screen, lr equals this linear regression object, which is a predictor. For our predictor, we need to pass in both the x and the y and that will fit our linear regression to that x and y so then it learns the parameters.

In [60]:
```
y_train_bc.shape
```

Out[60]:
```
(354,)
```

When you're doing your StandardScaler, you have to do that after doing your traintest split. If you imagine that you're trying to come up with the parameters for what's your StandardScaler would be and you just have your testset, imagine your testset was just one value, you wouldn't be able to come up with the mean and standard deviation in order to normalize that testset. So you have to use the mean and standard deviation that was originally defined on our trainingset.

In [62]:
```
lr.fit(X_train_s, y_train_bc)
X_test_s = s.transform(X_test)
y_pred_bc = lr.predict(X_test_s)
```

In [63]:
```
X_test_s.shape # 30 percent of our testset. So if you recall x_test_s.shape, we see that that is going to be 152 rows.
```

Out[63]:
```
(152, 104)
```

Discussion

- Are we done?
- What did we predict?
- How would you interpret these predictions?

. Let's look at our original y_test and we see they're probably not on the same scale and they seem pretty far off and we have to think why are they far off? It's because we transform our variable using the box cox transformation. So for us to realign and actually predict the median values moving forward, we have to do an inverse transformation to get them back on the same scale as our original numbers.

In [67]:
```
y_test
```

Out[67]:
```
502    20.6
127    16.2
390    15.1
303    33.1
277    33.1
       ...
433    14.3
505    11.9
53     23.4
82     24.8
377    13.3
Name: MEDV, Length: 152, dtype: float64
```

In [68]:
```
y_pred_bc
```

Out[68]:
```
array([4.43920316, 4.16230299, 4.14903773, 5.72770375, 5.70362608,
       6.59865445, 4.18842601, 4.77415222, 4.68282566, 4.85108878,
       6.82748198, 3.86236699, 5.0643167 , 4.31536775, 6.91107238,
       4.93538901, 4.64299542, 5.84297841, 5.24960436, 5.44823381,
       3.98464025, 4.88299011, 4.19919967, 3.59736013, 5.2309719 ,
       5.30478058, 5.85592444, 4.78820348, 4.78846728, 6.78622305,
       3.36299121, 4.39751616, 4.29381841, 6.9642437 , 4.49873355,
       5.51107039, 4.79092744, 5.61456675, 3.04824493, 5.35252088,
       3.34764987, 3.97589975, 5.14213658, 4.37539217, 5.07041151,
       6.31192606, 4.90476787, 5.25799412, 5.69133946, 4.21362253,
       5.2252357 , 5.49371642, 4.38650851, 3.95353734, 4.16951912,
       2.75465746, 4.06367514, 3.01030741, 4.5542641 , 4.8090537 ,
       5.74517281, 3.26233918, 3.5862807 , 6.17670312, 4.23677257,
       5.81985343, 3.41296646, 3.83284203, 4.98192021, 5.73417989,
       5.20709227, 5.2224684 , 5.25901124, 5.43662547, 3.8263134 ,
       3.54227896, 5.39503706, 4.46815491, 4.3752375 , 4.43397526,
       4.26857806, 4.86974534, 5.83795891, 4.04151355, 3.51858119,
       3.1923357 , 4.62263543, 6.02926386, 4.7009583 , 6.75024002,
       4.39142328, 7.09519245, 3.63633109, 6.84266201, 5.94655295,
       4.85824664, 3.31982381, 3.07049243, 4.97129982, 5.04765573,
       1.99045119, 3.36771013, 3.93454864, 5.25195111, 5.72084992,
       5.41600891, 4.6339003 , 4.06064081, 3.90652671, 5.47241751,
       6.34006908, 4.62295396, 5.78807659, 4.98261655, 4.79651198,
       5.74907439, 6.62391016, 3.79310717, 5.26601387, 5.05936907,
       5.27005544, 4.05674121, 4.90100131, 4.05584843, 5.22049234,
       5.40283716, 4.578356  , 4.22107366, 4.77645049, 4.8113669 ,
       5.2060228 , 4.23331845, 4.98686063, 4.95300337, 4.87192351,
       4.81043257, 3.21302163, 3.72381938, 5.6528538 , 4.83102518,
       3.61794231, 4.97683061, 4.81408485, 4.45751093, 5.63842481,
       6.18962525, 2.45986308, 4.31803429, 4.33344344, 4.90983086,
       5.18275029, 3.47798262])
```

Inverse transform

Every transformation has an inverse transformation. The inverse transformation of $f(x) = \sqrt{x}$ is $f^{-1}(x) = x^2$, for example. Box cox has an inverse transformation as well: notice that we have to pass in the lambda value that we found from before:

In [69]:
```
from scipy.special import inv_boxcox
```

we can bring in from scipy.special the inverse boxcox, that's just going to be the inverse of the function. If you look here, f of x equals square root of x, the inverse of that is x squared. You can think about something similar when you're working with the box cox, we're just having the inverse to get it back to the original form of x. So using our original box cox transformation on the entire dataset, we did bc_result equals boxcox, pulling in that function, passing it over the entire y value, and then we get both the boxcox array, which is the first output, as well as the Lambda value. I

In [70]:
```
# code from above
bc_result = boxcox(boston_data.MEDV)
boxcox_medv = bc_result[0]
lam = bc_result[1]
```

In order to get the inverse, we just need to pass in the array that we want to get the inverse of, which is this boxcox transformed version of the y variable as well as the Lambda that was used in the original boxcox, which was our second result

In [71]:
```
inv_boxcox(boxcox_medv, lam)[:10]
```

Out[71]:
```
array([24. , 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9])
```

In [72]:
```
boston_data['MEDV'].values[:10]
```

Out[72]:
```
array([24. , 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9])
```

Exactly the same, as we would hope!

Exercise:

1. Apply the appropriate inverse transformation to `y_pred_bc`.
2. Calculate the R^2 using the result of this inverse transformation and `y_test`.

Hint: Should be two lines of code.

In [73]:
```
# Instructor Solution
y_pred_tran = inv_boxcox(y_pred_bc,lam2)
r2_score(y_pred_tran,y_test)
```

Out[73]:
```
0.8794001851338578
```

Determine the R^2 of a LinearRegression without the box cox transformation. Is it higher or lower?

In [74]:
```
### BEGIN SOLUTION
lr = LinearRegression()
lr.fit(X_train_s,y_train)
lr_pred = lr.predict(X_test_s)
r2_score(lr_pred,y_test)
### END SOLUTION
```

Out[74]:
```
0.8555202098064165
```