Machine Learning Foundation

Regression Intro: Transforming Target

Learning objectives

By the end of this lesson, you will be able to:

- Apply transformations to make target variable more normally distributed for Regression
- Apply 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

# Surpress warnings:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

In the following cells we will load the data and define some useful plotting functions.

In [2]:
```
np.random.seed(72018)


def to_2d(array):
    return array.reshape(array.shape[0], -1)

def plot_exponential_data():
    data = np.exp(np.random.normal(size=1000))
    plt.hist(data)
    plt.show()
    return data

def plot_square_normal_data():
    data = np.square(np.random.normal(loc=5, size=1000))
    plt.hist(data)
    plt.show()
    return data
```

Loading the Boston Housing Data

In [3]:
```
file_name='https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDe
boston_data = pd.read_csv(file_name)
```

In [4]:
```
boston_data.head(15)
```

Out[4]:

	Unnamed: 0	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTR
0	0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	
1	1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	
2	2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	
3	3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	
4	4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	
5	5	0.02985	0.0	2.18	0.0	0.458	6.430	58.7	6.0622	3.0	222.0	
6	6	0.08829	12.5	7.87	0.0	0.524	6.012	66.6	5.5605	5.0	311.0	
7	7	0.14455	12.5	7.87	0.0	0.524	6.172	96.1	5.9505	5.0	311.0	
8	8	0.21124	12.5	7.87	0.0	0.524	5.631	100.0	6.0821	5.0	311.0	
9	9	0.17004	12.5	7.87	0.0	0.524	6.004	85.9	6.5921	5.0	311.0	
10	10	0.22489	12.5	7.87	0.0	0.524	6.377	94.3	6.3467	5.0	311.0	
11	11	0.11747	12.5	7.87	0.0	0.524	6.009	82.9	6.2267	5.0	311.0	
12	12	0.09378	12.5	7.87	0.0	0.524	5.889	39.0	5.4509	5.0	311.0	
13	13	0.62976	0.0	8.14	0.0	0.538	5.949	61.8	4.7075	4.0	307.0	
14	14	0.63796	0.0	8.14	0.0	0.538	6.096	84.5	4.4619	4.0	307.0	

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.

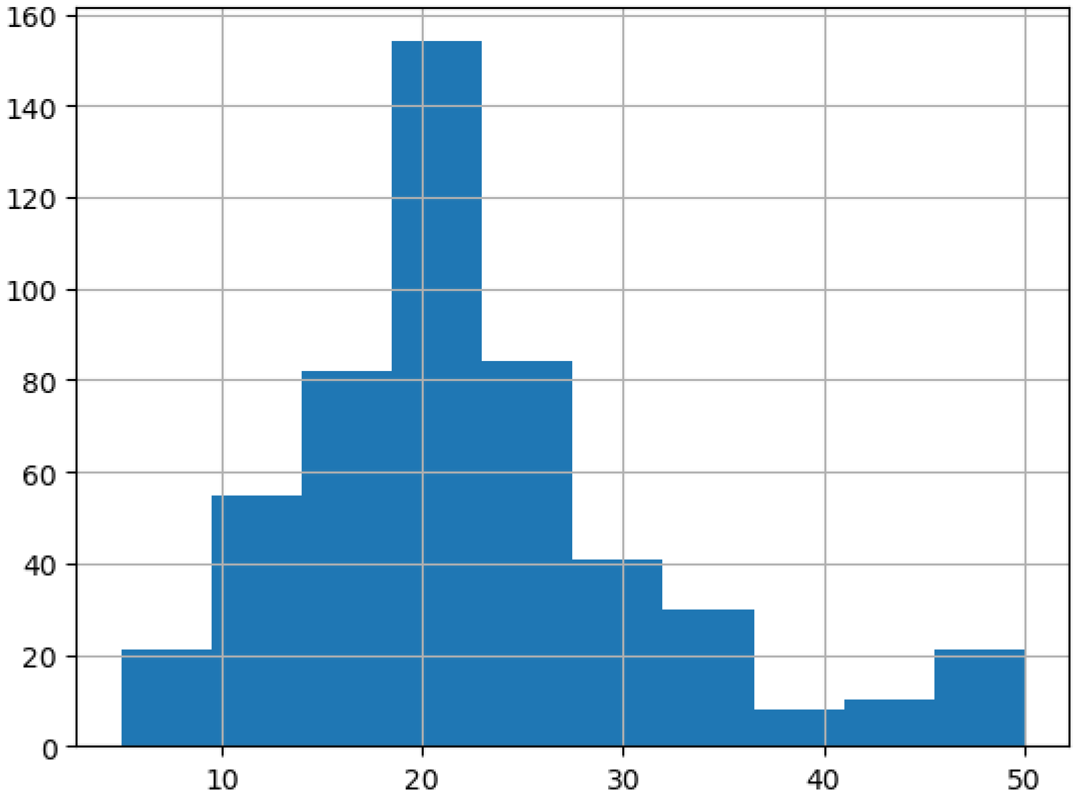
How can we tell if our target is normally distributed? There are two ways:

- Using a Visual approach
- Using a Statistical Test

Using a Visual approach

Plotting a histogram:

In [5]:
```
boston_data.MEDV.hist();
```



The histogram does not look normal due to its right tail.

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

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

Out[7]:
```
NormaltestResult(statistic=90.97468737009666, pvalue=1.7583188871696483e-20)
```

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

Apply transformations to make target variable more normally distributed for Regression

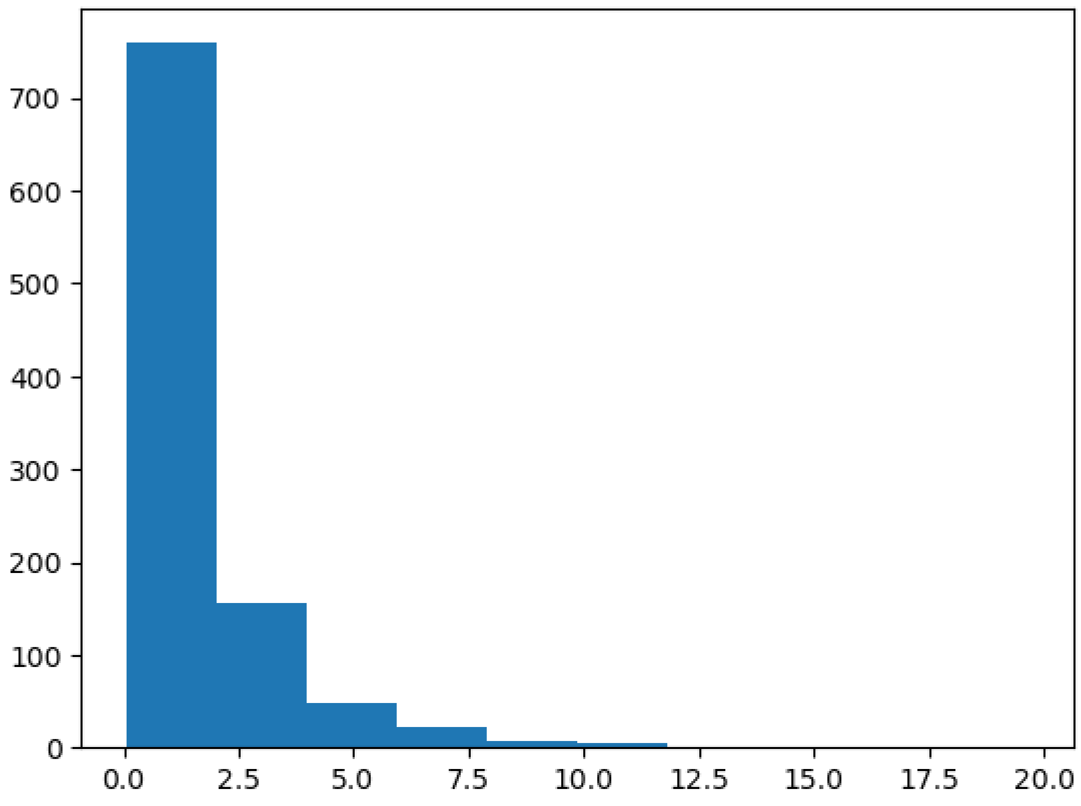
Linear Regression assumes a normally distributed residuals which can be aided by transforming **y** variable which is the target variable. Let's try some common transformations to try and get **y** to be normally distributed:

- Log Transformation
- Square root Transformation
- Box cox Transformation

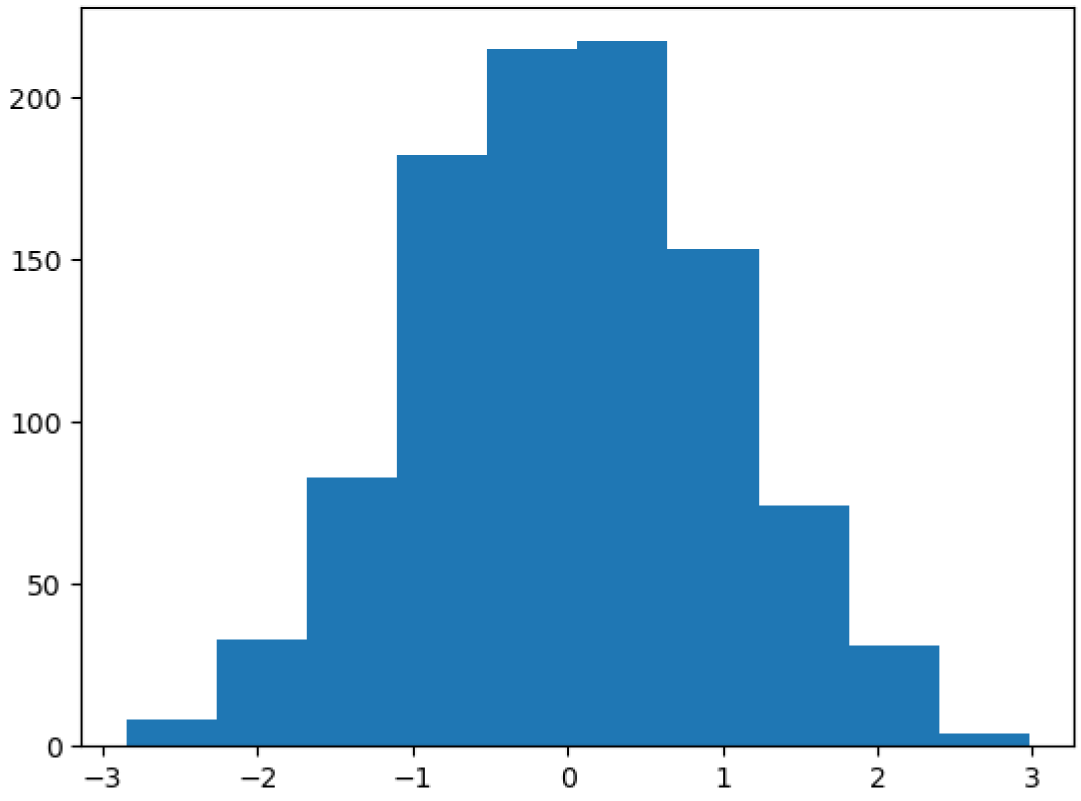
Log Transformation

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

In [8]:
```
data = plot_exponential_data()
```



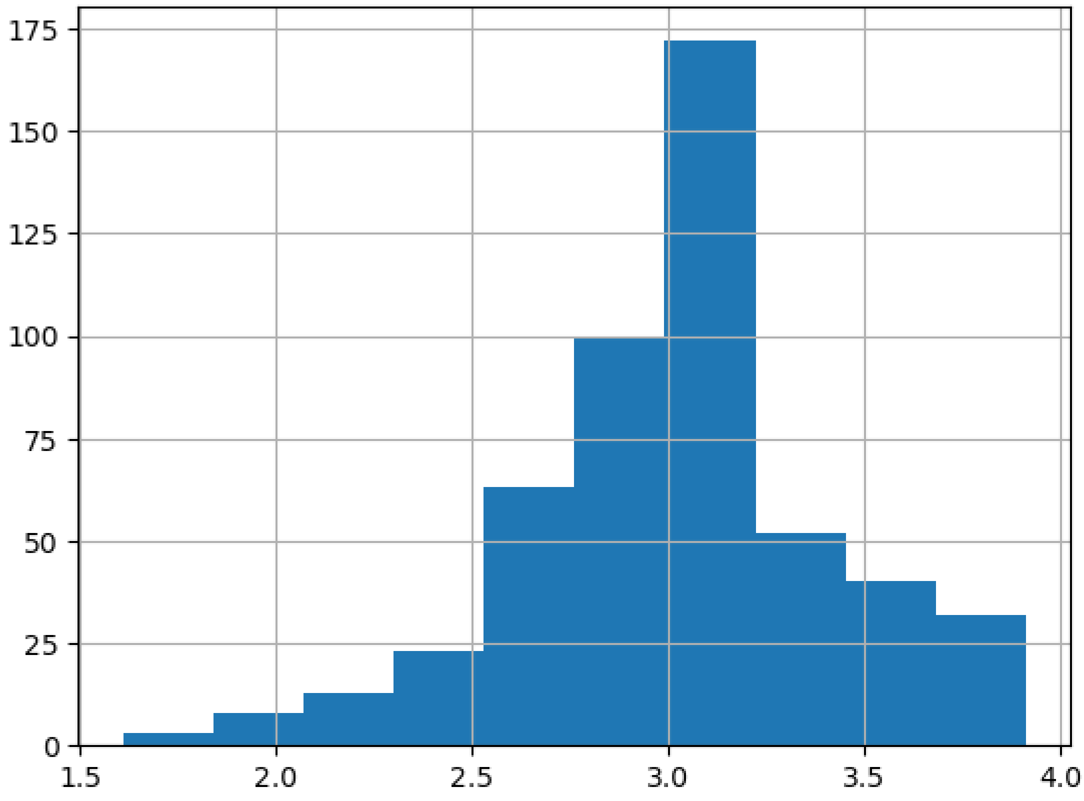
```
In [9]: plt.hist(np.log(data));
```



Apply transformation to Boston Housing data:

In [10]:
```
log_medv = np.log(boston_data.MEDV)
```

In [11]:
```
log_medv.hist();
```



In [12]:
```
normaltest(log_medv)
```

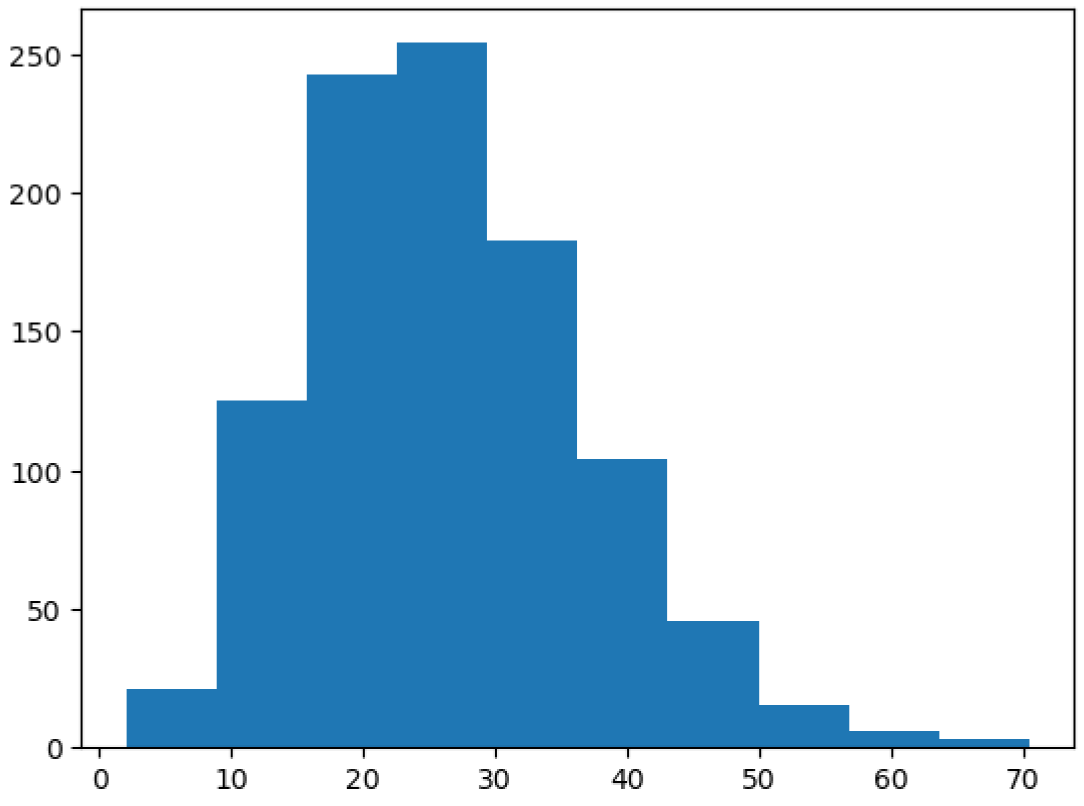
Out[12]:
```
NormaltestResult(statistic=17.218016966406978, pvalue=0.00018245472768345 23)
```

Conclusion: The output is closer to normal distribution, but still not completely normal.

Square root Transformation

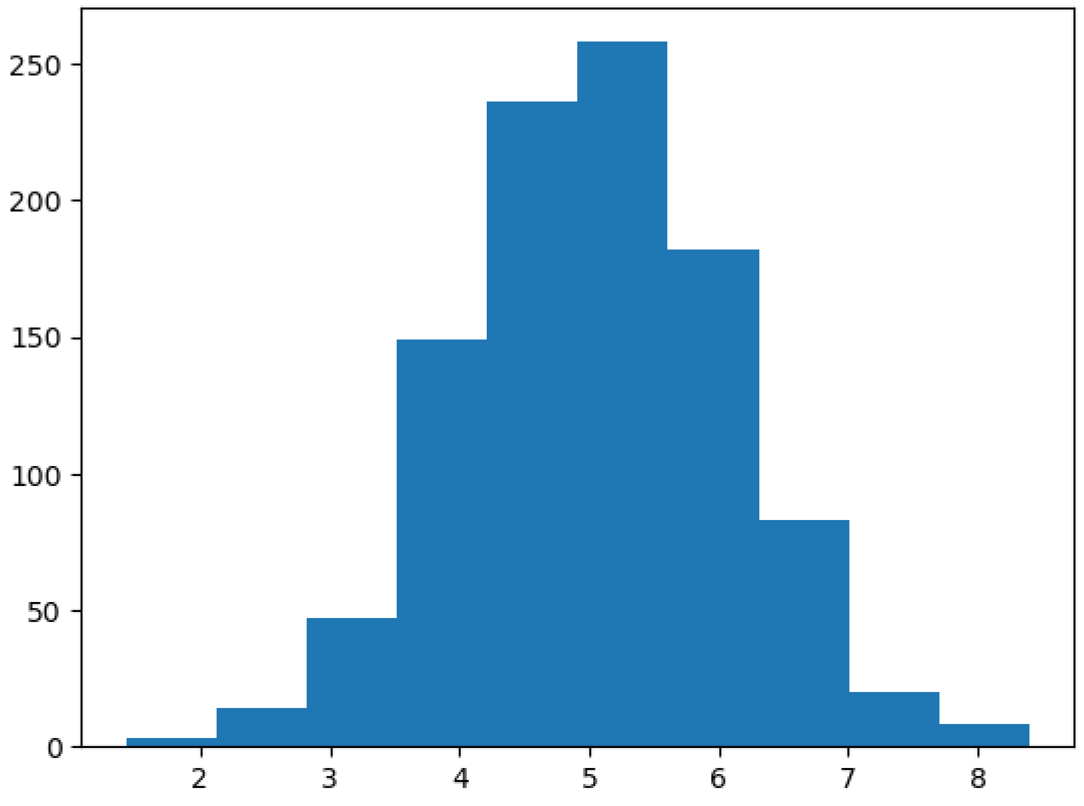
The square root transformation is another transformation that can transform non-normally distributed data into normally distributed data:

In [13]:
```
data = plot_square_normal_data()
```



You may notice that the output still exhibits a slight right skew.

```
In [14]: plt.hist(np.sqrt(data));
```

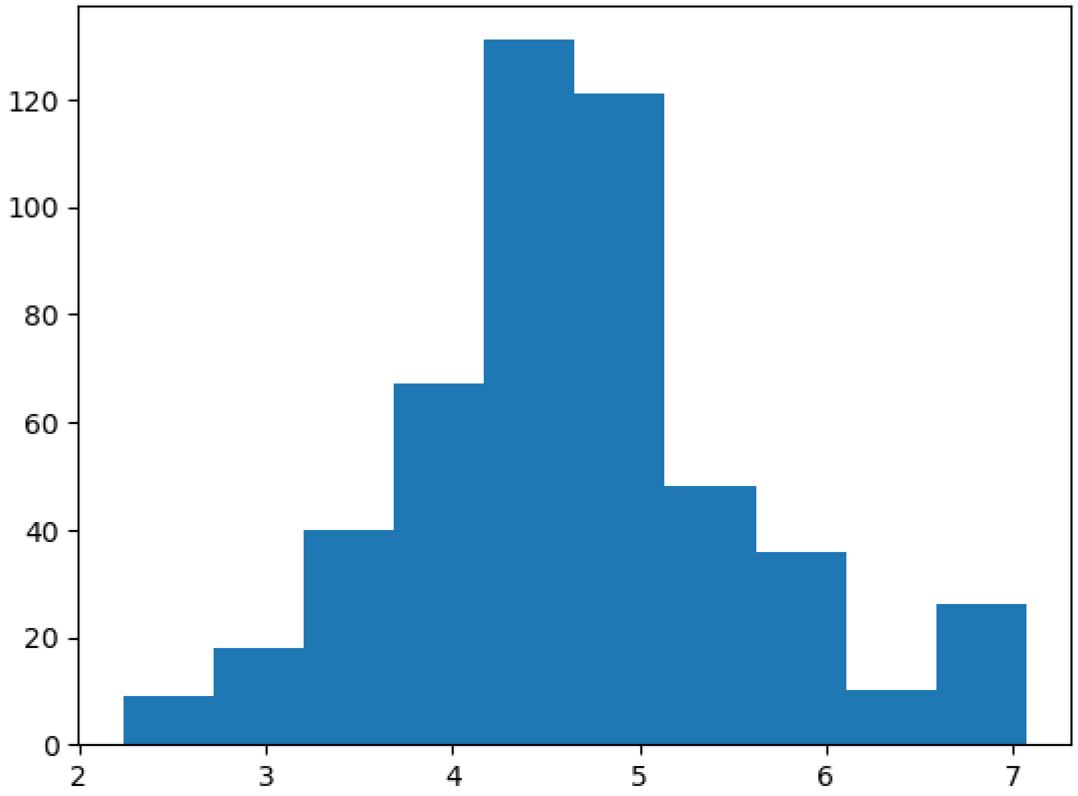


Exercise

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

In [24]:
```
## Enter your code here
sqrt_medv = np.sqrt(boston_data.MEDV)
plt.hist(sqrt_medv)
print(normaltest(sqrt_medv))
```

```
NormaltestResult(statistic=20.487090826863067, pvalue=3.558645701429252e-05)
```



▶ Click here for a sample python solution

Box cox Transformation

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 [25]:
```
from scipy.stats import boxcox
```

In [26]:
```
bc_result = boxcox(boston_data.MEDV)
boxcox_medv = bc_result[0]
```

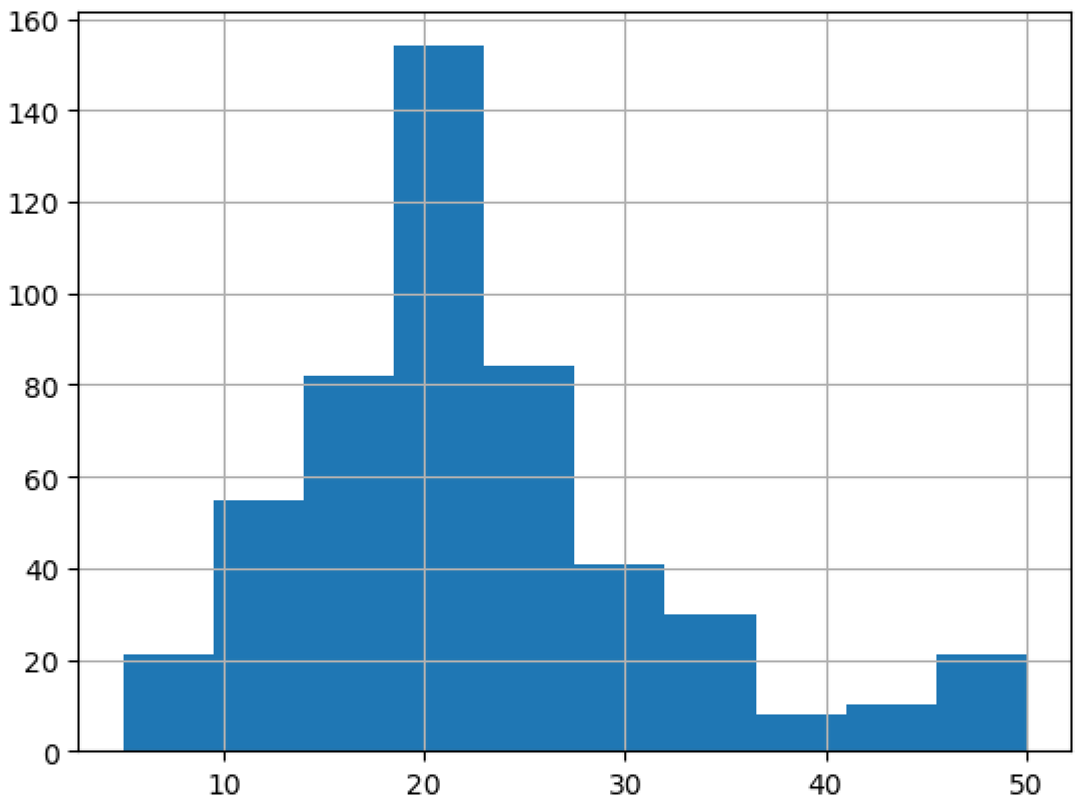
```
lam = bc_result[1]
```

1. `boston_data.MEDV` : You are accessing the "MEDV" column from the Boston housing dataset.
2. `boxcox(boston_data.MEDV)` : This is applying the Box-Cox transformation to the "MEDV" data.
3. `bc_result[0]` : This extracts the transformed data after applying the Box-Cox transformation. It seems you've assigned this to the variable `boxcox_medv` .
4. `bc_result[1]` : This extracts the lambda parameter, which is used in the Box-Cox transformation. The lambda parameter indicates the power to which the data is raised. It is often chosen to maximize the normality of the resulting transformed data. You've assigned this to the variable `lam` .

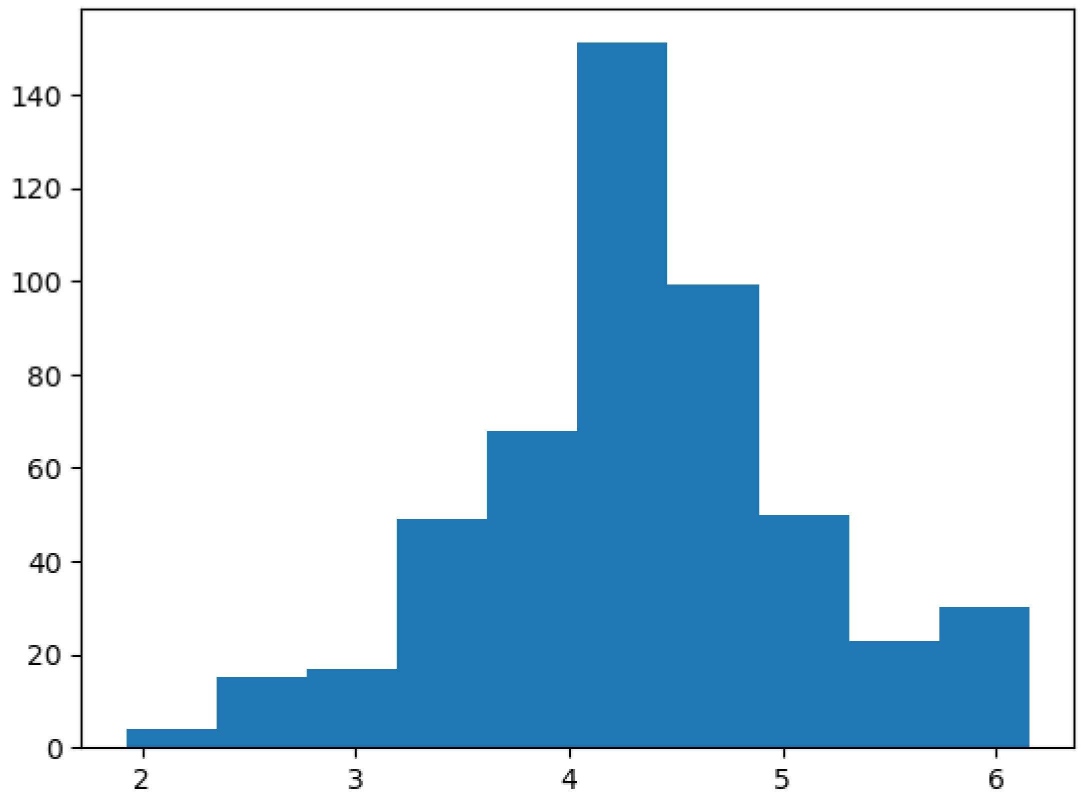
In [27]:
```
lam
```

Out[27]:
```
0.2166209218196593
```

In [28]:
```
boston_data['MEDV'].hist();
```



In [29]:
```
plt.hist(boxcox_medv);
```



In [30]:
```
normaltest(boxcox_medv)
```

Out[30]:
```
NormaltestResult(statistic=4.513528712300468, pvalue=0.1046886725916274)
```

We find that the box cox results in a graph which is significantly more normally distributed (according to p value) than the other two distributions.This can be even above 0.05.

Now that we have a normally distributed y-variable, let's test Regression using this transformed target variables.

Testing regression: (Steps required for Testing regression)

In [32]:
```
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 [33]:
```
lr = LinearRegression()
```

1. Define and load the predictor (X) and Target(y) variables

In [34]:
```
y_col = "MEDV"

X = boston_data.drop(y_col, axis=1) # df without the MEDV column
y = boston_data[y_col] # just the MEDV column
```

If `include_bias` is set to True, the first column of the transformed features will be a constant term (bias) equal to 1. This is useful when you want to fit a model that includes an intercept term. Including a bias term allows the polynomial features to account for the constant offset in the data.

If `include_bias` is set to False, the first column of the transformed features will not be a constant term, and the resulting polynomial features will not include an intercept term. This is appropriate when you don't want or need an intercept term in your model.

2. Create Polynomial Features

In [36]:
```
pf = PolynomialFeatures(degree=2, include_bias=False)
X_pf = pf.fit_transform(X)
```

3. Split the data into Training and Test Sets

The split ratio here is 0.7 and 0.3 which means we will assign **70%** data for training and **30%** data for testing

In [37]:
```
X_train, X_test, y_train, y_test = train_test_split(X_pf, y, test_size=0.3,
                                                    random_state=72018)
```

The `random_state` parameter in the `train_test_split` function is used to control the randomness of the data splitting process. When you set a specific value for random_state, it ensures that the data splitting is reproducible. In other words, if you use the same random_state value across multiple runs of your code, you'll get the same train and test sets.

4. Normalize the training data using `StandardScaler` on `X_train`. Use fit_transform() function

In [38]:
```
s = StandardScaler()
X_train_s = s.fit_transform(X_train)
```

5. Discuss: what transformation do we need to apply next?

Apply the appropriate transformation.

In [42]:
```
# Enter your code here
y_train_bc,lam2 = boxcox(y_train)
```

The function returns two values:

Transformed Data (`y_train_bc`): This is the array or list containing the transformed data after applying the Box-Cox transformation. The transformed data is adjusted to make it as close to a normal distribution as possible.

Lambda Value (`lam2`): This is the parameter used in the Box-Cox transformation. If you didn't provide a lambda manually, the function calculates and returns the optimal lambda for the transformation. This lambda is determined through maximum likelihood estimation.

▶ Click here for a sample python solution

As before, we'll now:

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

```
In [46]: y_train_bc.shape
```

```
Out[46]: (354,)
```

```
In [48]: """ The training process involves adjusting the model parameters to minimize the
difference between the predicted values and the actual target values."""

lr.fit(X_train_s, y_train_bc)

""" It's crucial to use the same preprocessing steps on both the training and test
the model is making predictions on data that is in a similar format as what it was
The s.transform method suggests that some form of scaling is being applied."""

X_test_s = s.transform(X_test)

"""Finally, this line uses the trained linear regression model to predict the targe
for the transformed test data (X_test_s). The predictions can then be compared with
target values (y_test_bc) to evaluate the performance of the model."""

y_pred_bc = lr.predict(X_test_s)
```

Discussion

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

The use of an **inverse transformation** after making predictions is often necessary when the target variable has undergone some form of transformation during preprocessing, and you want to bring the predicted values back to the original scale. This is especially common when dealing with transformations like the Box-Cox transformation or standardization (z-score scaling).

In our case, we did the Box-Cox transformation earlier. If our target variable (y) underwent a Box-Cox transformation during preprocessing, the predictions (y_pred_bc) would be on the transformed scale. To interpret and evaluate the predictions, you need to inverse-transform them to bring them back to the original scale.

In [63]:
```
print("We can see Shape of y_test is {} as same as of y_pred_bc {}".format(y_test.s
print("but they are on different scales. Below first 3 values of these two lists ar
print("Values of y_test")
print(y_test[:3])
print("Values of y_pred_bc")
print(y_pred_bc[:3])
```

```
We can see Shape of y_test is (152,) as same as of y_pred_bc (152,)
but they are on different scales. Below first 3 values of these two lists are displa
yed
Values of y_test
502    20.6
127    16.2
390    15.1
Name: MEDV, dtype: float64
Values of y_pred_bc
[4.35846763 4.06342569 4.11906424]
```

Apply inverse transformations to be able to use these in a Regression context

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 [64]:
```
from scipy.special import inv_boxcox
```

In [66]:
```
# code from above
```

In [67]:
```
inv_boxcox(boxcox_medv, lam)[:10]
```

Out[67]:
```
array([24. , 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9])
```

In [68]:
```
boston_data['MEDV'].values[:10]
```

Out[68]:
```
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: Use the **inv_boxcox()** function to get the transformed predicted values

In [78]:
```
#Enter your code here
y_pred_train = inv_boxcox(y_pred_bc,lam2)
r2_score(y_pred_train,y_test)
```

```
Out[78]: 0.8480525379812784
```

▶ Click here for a sample python solution

Practice Exercise:

Determine the R^2 of a LinearRegression without the box cox transformation.

```
In [79]: # Enter your code here
         lr = LinearRegression()
         lr.fit(X_train_s,y_train)
         lr_pred = lr.predict(X_test_s)
         r2_score(lr_pred,y_test)
```

```
Out[79]: 0.8667029116056741
```

▶ Click here for a sample python solution

The key difference between determining the R^2 of a Linear Regression model with and without the Box-Cox transformation lies in the nature of the target variable and the assumptions of linear regression.

1. **Box-Cox Transformation:**

   - When you apply the Box-Cox transformation to the target variable, you are potentially addressing issues related to heteroscedasticity(In simpler terms, it means that the spread of the residuals is not uniform throughout the range of the predictor variable(s)) and non-normality of residuals.
   - The Box-Cox transformation is useful when the variance of the target variable is not constant across different levels of the independent variables, and the residuals are not normally distributed.
   - Linear regression assumes constant variance and normally distributed residuals. Transforming the target variable can help in meeting these assumptions.

2. **Linear Regression without Box-Cox Transformation:**

   - Without the Box-Cox transformation, you are assuming that the relationship between the independent variables and the target variable is linear, and the residuals are normally distributed with constant variance.
   - If these assumptions are violated, the R^2 value might be less reliable as a measure of model performance. Residual analysis is crucial in such cases to check for violations of assumptions.

The Catch:

- The catch lies in whether the assumptions of linear regression are met or violated. If the assumptions are met, applying the Box-Cox transformation might not provide significant improvements, and the R^2 values with and without transformation might be similar.
- If the assumptions are violated (e.g., non-constant variance, non-normality of residuals), the Box-Cox transformation could lead to a more reliable model, and you might observe an improvement in R^2.

Steps to Consider:

1. **Check Assumptions:** Assess the assumptions of linear regression, including constant variance and normality of residuals. Diagnostic plots (e.g., residuals vs. predicted values) can be helpful.
2. **Compare R^2 Values:** Compare the R^2 values obtained with and without the Box-Cox transformation. If they are similar and assumptions are met, the transformation might not be necessary. If they differ, it suggests that the Box-Cox transformation has influenced the model performance.

In summary, the catch is in understanding whether the Box-Cox transformation is helping address violations of linear regression assumptions, leading to a more reliable model and potentially a different R^2 value. The decision to use the transformation should be based on a thorough understanding of the data and the assumptions of the regression model.