Lab 3: Scikit Learn and Regression

Deadline Friday 4/1/22 11:59 pm

scikit-learn is a popular machine learning package that contains a variety of models and tools. In this lab and lab 4 we will work with different models provided by **scikit-learn** package and build several models.

All objects within scikitt-learn share a uniform common basic API consisting of 3 interfaces: an *estimator* interface for building and fitting models, a *predictor* interface for making predictions, and a *transformer* interface for converting data.

> The *estimator* interface defines object mechanism and a fit method for learning a model from training data. All supervised and unsupervised learning algorithms are offered as objects implementing this interface. Other machine learning tasks such as *feature extraction*, *feature selection*, and *dimensionality reduction* are provided as *estimators*.

For more information, check the scikit-learn API paper: [https://arxiv.org/pdf/1309.0238v1.pdf]

The general form of using models in scikit-learn:

```
clf = someModel( )
clf.fit(x_train , y_tain)
```

For Example:

```
clf = LinearSVC( )
clf.fit(x_train , y_tain)
```

> The *predictor* adds a predict method that takes an array x_test and produces predictions for x_test, based on the learned parameters of the *estimator*. In supervised learning, this method typically return predicted labels or values computed by the model. Some unsupervised learning estimators may also implement the predict interface, such as **k-means**, where the predicted values are the cluster labels.

```
clf.predict(x_test)
```

> *transform* method is used to modify or filter data before feeding it to a learning algorithm. It takes some new data as input and outputs a transformed version of that data. Preprocessing, feature selection, feature extraction and dimensionality reduction algorithms are all provided as *transformers* within the library.

This is usually done with **fit_transform** method. For example:

```
PCA = RandomizedPCA (n_components = 2)
x_train = PCA.fit_transform(x_train)
```

```
x_test = PCA.fit_transform(x_test)
```

In the example above, we first **fit** the training set to find the PC components, then they are transformed.

We can summarize the *estimator* as follows:

- In *all estimators*
    - `model.fit()` : fit training data. In supervised learning, fit will take two parameters: the data x and labels y. In unsupervised learning, fit will take a single parameter: the data x
- In *supervised estimators*
    - `model.predict()` : predict the label of new test data for the given model. Predict takes one parameter: the new test data and returns the learned label for each item in the test data
    - `model.score()` : Returns the score method for classification or regression methods.
- In *unsupervised estimators*
    - `model.transform()` : Tranform new data into new basis. Transform takes one parameter: new data and returns a new representation of that data based on the model

Linear Regression

Let's start with a simple linear regression. First we will see an example of a simple linear regression. A simple straight line that fits the data. The formula representing the model is

$$y = \beta_1 x + \beta_0$$

In [60]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

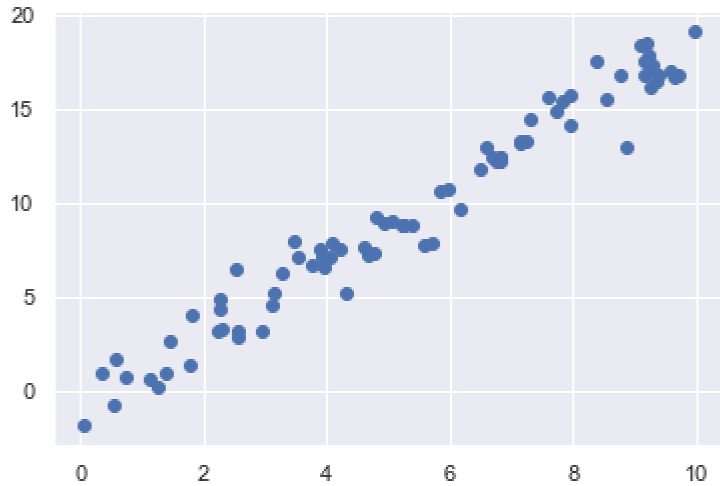
Let's start by using the following simple data for showing how linear regression works in scikit-learn. Then it will be your turn to build a regression model on a dataset

In [61]:

```
rng = np.random.RandomState(50)

x = 10 * rng.rand(80)
y = 2 * x - 1 + rng.randn(80)

plt.scatter(x,y)
```

Out[61]:

```
<matplotlib.collections.PathCollection at 0x167b05e8cd0>
```

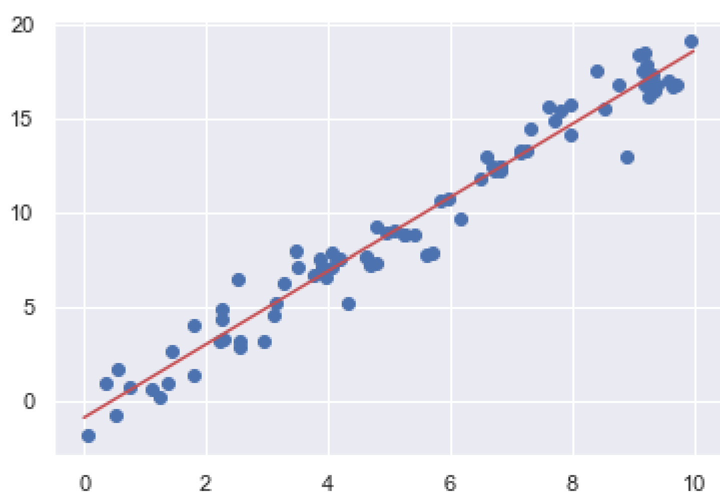


After processing your data, the first step is to choose a model. For the dataset above, we are going to pick "Linear Regression" model. Simply import your model:

In [62]:
```
from sklearn.linear_model import LinearRegression
```

Next, pick the model hyperparameters

In [63]:
```
model = LinearRegression(fit_intercept=True)

model.fit(x[:, np.newaxis], y)

xfit = np.linspace(0, 10, 1000)
yfit = model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit, 'r');
```



We can check the model settings:

In [64]:
```
print(model.coef_[0])
print(model.intercept_)
```

```
1.944535887214308
-0.8492545699739527
```

Linear regression on scikit-learn datasets

You can use datasets provided by scikit-learn as well. In the example below, we will apply linear regression to the **diabetes** dataset.

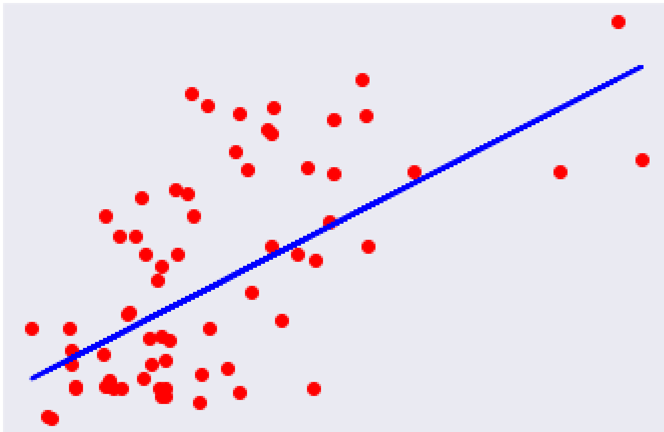
In the diabetes datasets, ten baseline variables; age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline.

In [65]:

```
# Importing diabetes dataset
from sklearn.datasets import load_diabetes

from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score


# Load the diabetes dataset
diabetes = load_diabetes()

# Use only one feature -- the following code creates a 1-dimensional
# array containing just the second feature
diabetes_X = diabetes.data[:, np.newaxis]
diabetes_X_data = diabetes_X[:,:,2]

# Split the data into training/testing sets
diabetes_X_train,diabetes_X_test , diabetes_y_train, diabetes_y_test = train_test_split
                                                     diabetes_X_data, diabetes.target, test_size = 0.15


# Create linear regression object
m1 = LinearRegression()


# Train the model with training data
m1.fit(diabetes_X_train, diabetes_y_train)

# Make predictions on test data
diabetes_y_pred = m1.predict(diabetes_X_test)

#print the coefficient
print('Coefficients: \n', m1.coef_)

#print the mean squared error
print('mean squared error: %.2f'% mean_squared_error(diabetes_y_test, diabetes_y_pred))


# print the r-squared
print('R-squared: %.2f' % r2_score(diabetes_y_test, diabetes_y_pred))


# Plot
plt.scatter(diabetes_X_test, diabetes_y_test,  color='red')
plt.plot(diabetes_X_test, diabetes_y_pred, color='blue', linewidth=2)

plt.xticks(())
plt.yticks(())
```

```
plt.show()
```

```
Coefficients:
 [944.91600363]
mean squared error: 3894.19
R-squared: 0.38
```



How to evaluate a regression model?

When it comes to regression model we have a variety of metrics that we can use to evaluate our model. The most common ones are:

- Mean Squared Error (MSE): the mean of squared errors:
  - $\frac{1}{n}\Sigma(y_i - \hat{y}_i)^2$
- Mean Absolute Error (MAE): the mean of the absolute value of the errors:
  - $\frac{1}{n}\Sigma|y_i - \hat{y}_i|$
- Root Mean Squared Error (RMSE): the square root of the mean of the squared errors:
  - $\sqrt{\frac{1}{n}\Sigma(y_i - \hat{y}_i)^2}$

You can check the full list of regression metrics here Scikit-Learn: Regression Metrics

Exercise 3.1 (20 pts)

First, perform the following tasks:

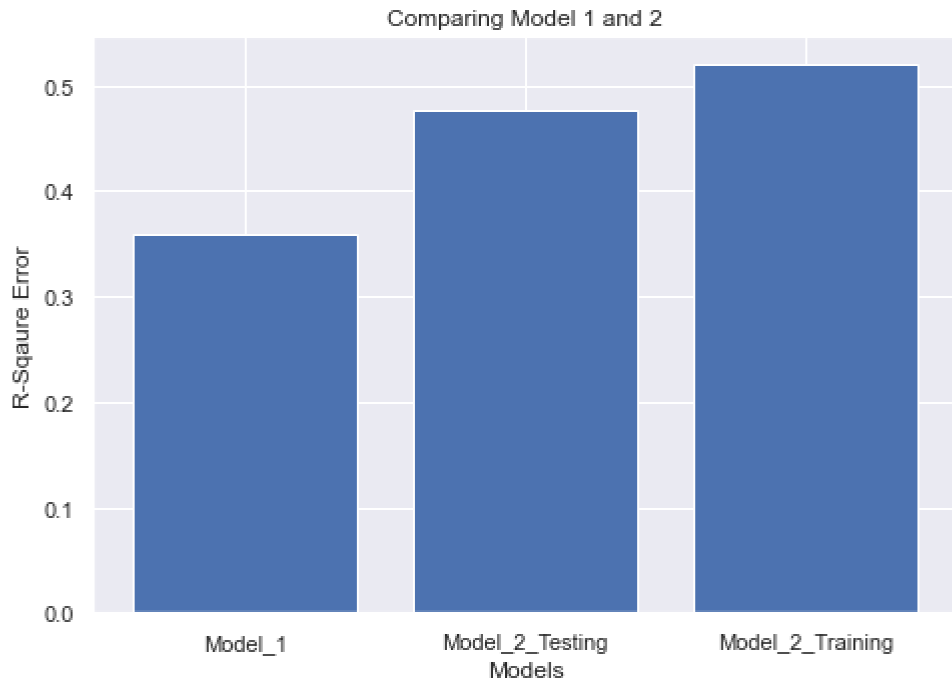
- Make a linear regression model with **all the features** in the dataset. Use `train_test_split` to keep 20% of the data for testing.
- Use your model to predict values for test set and print the predictions for the first 10 instances of the test data and compare them with actual values.
- Print the coefficient values and their corresponding feature name (e.g. age 43, bmi 200, ...)
  - Note that you can access `feature_names` from diabetes dataset directly
- Calculate training-MSE, testing-MSE, and R-squared value.

Compare the two models. Did using all available features improve the performance?

In [124…

```
# Your code goes here
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt2
diabetes1 = load_diabetes()
diabetes_X_train1,diabetes_X_test1, diabetes_y_train2, diabetes_y_test2 = train_test_sp
                                     diabetes1.data, diabetes1.target, test_size = 0.20

m1 = LinearRegression()


# Train the model with training data
m1.fit(diabetes_X_train1, diabetes_y_train2)

# Make predictions on test data
diabetes_y_pred3 = m1.predict(diabetes_X_test1)

#print the coefficient
print('Coefficients: \n', m1.coef_)

print(diabetes1['feature_names'])
print(diabetes1['data'][:10])

diabetes_y_pred4 = m1.predict(diabetes_X_train1)

#print the mean squared error
print('training mean squared error: %.2f'% mean_squared_error(diabetes_y_train2, diabet


# # print the r-squared
print('training R-squared: %.2f' % r2_score(diabetes_y_train2, diabetes_y_pred4))

#print the mean squared error
print('testing mean squared error: %.2f'% mean_squared_error(diabetes_y_test2, diabetes


# print the r-squared
print('testing R-squared: %.2f' % r2_score(diabetes_y_test2, diabetes_y_pred3))

models = ['Model_1','Model_2_Testing','Model_2_Training']
MSElist = [3654.04, mean_squared_error(diabetes_y_test2, diabetes_y_pred3), mean_square
Rlist = [0.36,r2_score(diabetes_y_test2, diabetes_y_pred3), r2_score(diabetes_y_train2,

# Plot
fig = plt.figure()
plt.bar(models,MSElist)
plt.xlabel("Models")
plt.ylabel("Mean Sqaure Error")
plt.title("Comparing Model 1 and 2")
plt.show()

fig = plt2.figure()
ax = fig.add_axes([0,0,1,1])
plt2.bar(models, Rlist)
plt2.xlabel("Models")
plt2.ylabel("R-Sqaure Error")
plt2.title("Comparing Model 1 and 2")
plt2.show()
```

```
Coefficients:
 [ -32.59372611 -257.78989843  485.13075339  308.91015825 -886.79653332
  603.67333532  185.54413185  262.13980737  801.25557898   33.58522923]
['age', 'sex', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6']
[[ 0.03807591  0.05068012  0.06169621  0.02187235 -0.0442235  -0.03482076
  -0.04340085 -0.00259226  0.01990842 -0.01764613]
 [-0.00188202 -0.04464164 -0.05147406 -0.02632783 -0.00844872 -0.01916334
   0.07441156 -0.03949338 -0.06832974 -0.09220405]
 [ 0.08529891  0.05068012  0.04445121 -0.00567061 -0.04559945 -0.03419447
  -0.03235593 -0.00259226  0.00286377 -0.02593034]
 [-0.08906294 -0.04464164 -0.01159501 -0.03665645  0.01219057  0.02499059
  -0.03603757  0.03430886  0.02269202 -0.00936191]
 [ 0.00538306 -0.04464164 -0.03638469  0.02187235  0.00393485  0.01559614
   0.00814208 -0.00259226 -0.03199144 -0.04664087]
 [-0.09269548 -0.04464164 -0.04069594 -0.01944209 -0.06899065 -0.07928784
   0.04127682 -0.0763945  -0.04118039 -0.09634616]
 [-0.04547248  0.05068012 -0.04716281 -0.01599922 -0.04009564 -0.02480001
   0.00077881 -0.03949338 -0.06291295 -0.03835666]
 [ 0.06350368  0.05068012 -0.00189471  0.06662967  0.09061988  0.10891438
   0.02286863  0.01770335 -0.03581673  0.00306441]
 [ 0.04170844  0.05068012  0.06169621 -0.04009932 -0.01395254  0.00620169
  -0.02867429 -0.00259226 -0.01495648  0.01134862]
 [-0.07090025 -0.04464164  0.03906215 -0.03321358 -0.01257658 -0.03450761
  -0.02499266 -0.00259226  0.06773633 -0.01350402]]
training mean squared error: 2786.56
training R-squared: 0.52
testing mean squared error: 3283.95
testing R-squared: 0.48
```



Feature selection allows your estimator to perform a better job by decreasing the model complexity and overfitting. scikit-learn provides several feature selection methods such as `SelectKBest` and `RFE`. Here is an example of using `RFE` or Recursive Feature Elimination on diabetes dataset:

In [125…

```
from sklearn.feature_selection import RFE

# Note that this piece of code works with training data and model from Exercise 3.1
# Either choose the same name for your variables or change the variable names below

rfe = RFE(estimator = m1 , n_features_to_select = 2 , step = 1)
rfe.fit(diabetes_X_train1, diabetes_y_train2)

print(rfe.ranking_)
```

```
[9 5 1 4 2 3 7 6 1 8]
```

The `RFE` is a popular feature selection algorithm. It is a feature ranking with recursive feature elimination.

```
Given an external estimator that assigns weights to features (e.g., the
coefficients of a linear model), the goal of recursive feature
elimination (RFE) is to select features by recursively considering
smaller and smaller sets of features. First, the estimator is trained on
the initial set of features and the importance of each feature is
obtained either through any specific attribute or callable. Then, the
least important features are pruned from current set of features. That
procedure is recursively repeated on the pruned set until the desired
number of features to select is reached.
```

`n_features_to_select` or number of features to select is an important hyperparameter for `RFE` algorithm. In the previous example, we set this parameter to 2 and the ranking shows the top two features that were selected.

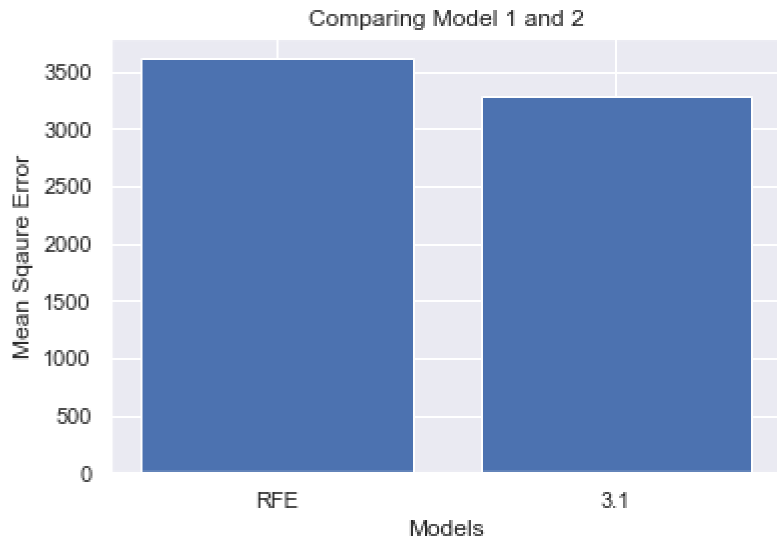
Exercise 3.2 (15 pts)

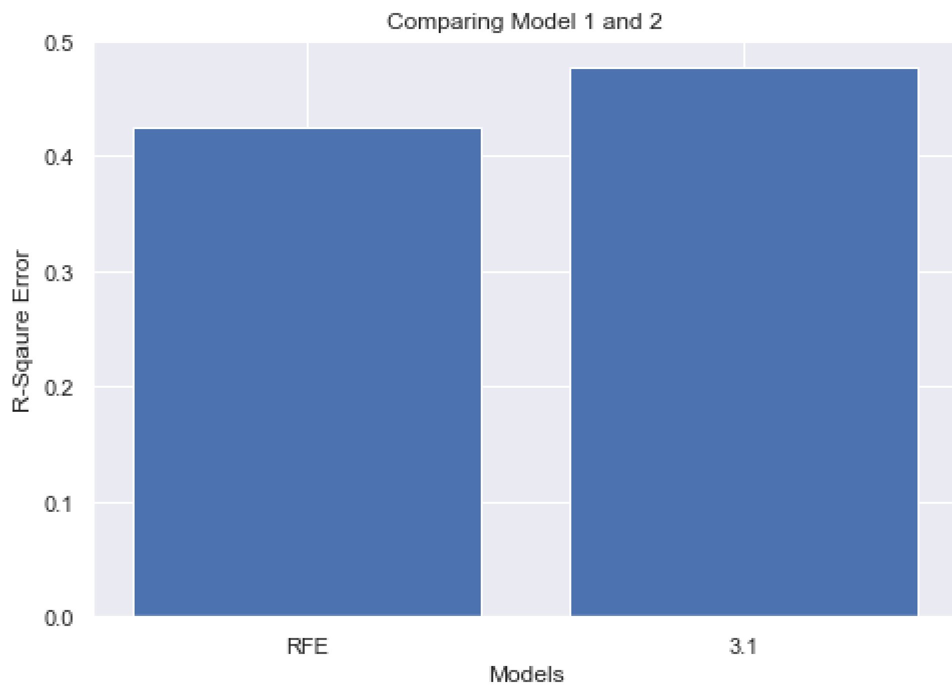
Make a Linear regression model using the two features that `RFE` selected. Calculate and print both train and test MSE and R-squared values, and compare this model with the model from 3.1. Explain your observation.

In [128…

```
# # Your code goes here
diabetes_y_pred6 = rfe.predict(diabetes_X_test1)

#print the mean squared error
print('testing rfe mean squared error: %.2f'% mean_squared_error(diabetes_y_test2, diab

# print the r-squared
print('testing rfe R-squared: %.2f' % r2_score(diabetes_y_test2, diabetes_y_pred6))

#print the mean squared error
print('testing 3.1 mean squared error: %.2f'% mean_squared_error(diabetes_y_test2, diab

# print the r-squared
print('testing 3.1 R-squared: %.2f' % r2_score(diabetes_y_test2, diabetes_y_pred3))

models = ['RFE','3.1']
MSE = [mean_squared_error(diabetes_y_test2, diabetes_y_pred6), mean_squared_error(diabe
R2 = [r2_score(diabetes_y_test2, diabetes_y_pred6), r2_score(diabetes_y_test2, diabetes_

# Plot
fig = plt.figure()
plt.bar(models,MSE)
plt.xlabel("Models")
plt.ylabel("Mean Sqaure Error")
plt.title("Comparing Model 1 and 2")
plt.show()

fig = plt2.figure()
ax = fig.add_axes([0,0,1,1])
plt2.bar(models, R2)
plt2.xlabel("Models")
plt2.ylabel("R-Sqaure Error")
plt2.title("Comparing Model 1 and 2")
plt2.show()
```

```
testing rfe mean squared error: 3614.41
testing rfe R-squared: 0.42
testing 3.1 mean squared error: 3283.95
testing 3.1 R-squared: 0.48
```





Linear regression on the Boston house price dataset

Now it's your turn to perform a linear regression on the **Boston housing** dataset.

Exercise 3.3 (10 pts)

Build a Linear Regression model for Boston house dataset using all available features. The first model you build can use 20% of the data for test (`test_size` = 0.2). Print the coefficients and their feature names. Calculate and print MSE and R-squred measures.

In [83]:
```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
boston = load_boston()


boston_x = boston.data[:]
```

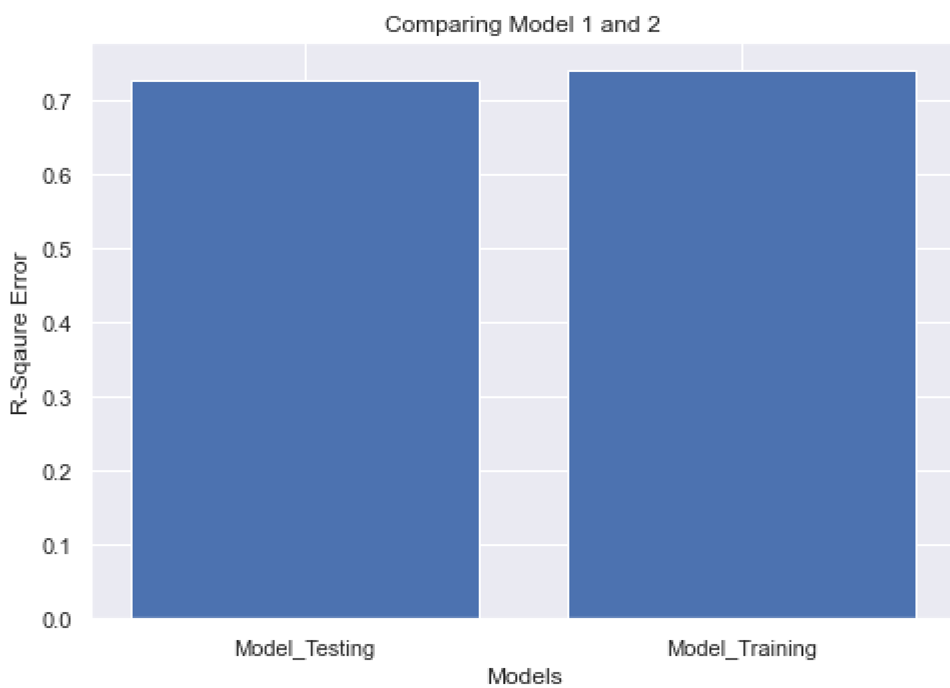
```
boston_y = boston.target


#Your code goes here
boston_X_train,boston_X_test, boston_y_train, boston_y_test = train_test_split(
                                     boston_x, boston_y, test_size = 0.20)

m1 = LinearRegression()


# Train the model with training data
m1.fit(boston_X_train, boston_y_train)

# Make predictions on test data
boston_y_pred1 = m1.predict(boston_X_test)
boston_y_pred2 = m1.predict(boston_X_train)

# print the coefficient
print('Coefficients: \n', m1.coef_)

print(boston['feature_names'])
print(boston['data'][:2])

# print the mean squared error
print('training mean squared error: %.2f'% mean_squared_error(boston_y_train, boston_y_

# print the r-squared
print('training R-squared: %.2f' % r2_score(boston_y_train, boston_y_pred2))

# print the mean squared error
print('testing mean squared error: %.2f'% mean_squared_error(boston_y_test, boston_y_pr

# print the r-squared
print('testing R-squared: %.2f' % r2_score(boston_y_test, boston_y_pred1))

models = ['Model_Testing','Model_Training']
MSElist = [mean_squared_error(boston_y_test, boston_y_pred1), mean_squared_error(boston
Rlist = [r2_score(boston_y_test, boston_y_pred1), r2_score(boston_y_train, boston_y_pre

# Plot
fig = plt.figure()
plt.bar(models,MSElist)
plt.xlabel("Models")
plt.ylabel("Mean Sqaure Error")
plt.title("Comparing Model 1 and 2")
plt.show()
```

```
Coefficients:
 [-1.03135168e-01  4.75015731e-02  1.26059484e-02  1.96287444e+00
 -1.50030032e+01  3.71856365e+00 -1.75613592e-03 -1.33762642e+00
  3.07788637e-01 -1.36230901e-02 -9.19483303e-01  1.01284220e-02
 -4.88786645e-01]
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
[[6.3200e-03 1.8000e+01 2.3100e+00 0.0000e+00 5.3800e-01 6.5750e+00
  6.5200e+01 4.0900e+00 1.0000e+00 2.9600e+02 1.5300e+01 3.9690e+02
  4.9800e+00]
 [2.7310e-02 0.0000e+00 7.0700e+00 0.0000e+00 4.6900e-01 6.4210e+00
  7.8900e+01 4.9671e+00 2.0000e+00 2.4200e+02 1.7800e+01 3.9690e+02
  9.1400e+00]]
```

```
training mean squared error: 20.44
training R-squared: 0.74
testing mean squared error: 28.54
testing R-squared: 0.73
```

In [84]:

```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
plt.bar(models, Rlist)
plt.xlabel("Models")
plt.ylabel("R-Sqaure Error")
plt.title("Comparing Model 1 and 2")
plt.show()
```



Exercise 3.4 (15 pts)

Let's build models with different training sizes to predict the house prices for boston house dataset. Each model should use all the available features. The goal is to train multiple linear regression models for the following train-test splits and calculate and visualize their MSE:

- a) 30% training, 70% testing
- b) 40% training, 60% testing
- c) 50% training, 50% testing
- d) 60% training, 40% testing
- e) 70% training, 30% testing

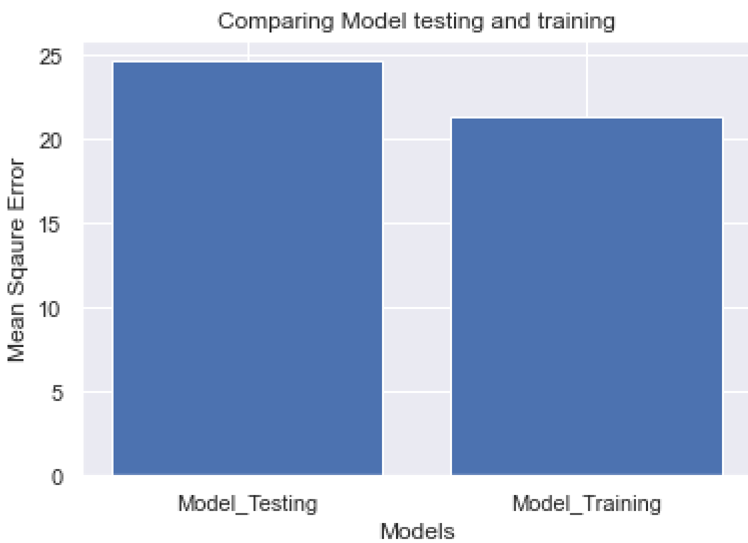
You also have the MSE value from previous step with 80% of data in training set. Plot the train and test MSE values for all models with respect to the training size.
(For simplicity, make a function that builds and fits the linear model and returns MSE)

In [158…

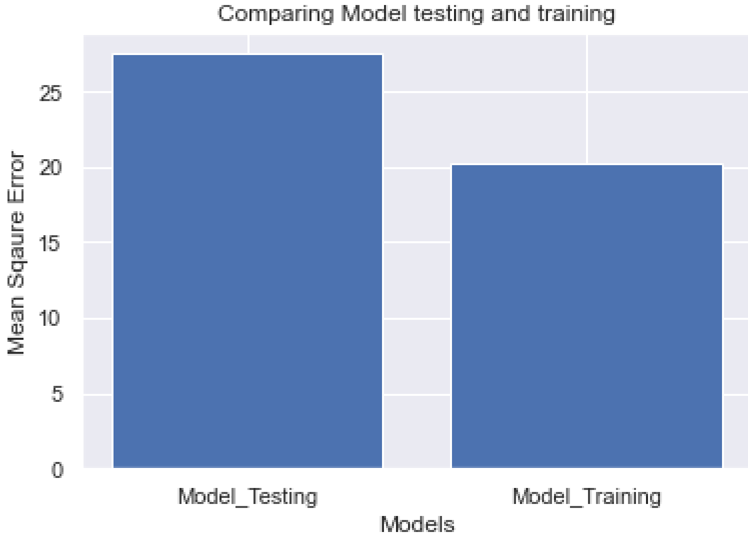
```
#Your code goes here

trainingMse = []
testingMse = []
l1 = []
l2 = []

for i in [0.20, 0.30, 0.40, 0.50, 0.60, 0.70]:
    boston_X_train,boston_X_test, boston_y_train, boston_y_test = train_test_split(
                                    boston_x, boston_y, test_size = i)
    m1 = LinearRegression()

    # Train the model with training data
    m1.fit(boston_X_train, boston_y_train)

    # Make predictions on test data
    boston_y_pred1 = m1.predict(boston_X_test)
    boston_y_pred2 = m1.predict(boston_X_train)

    # Print number
    print("The testing percent is", i*100)
    l1.append(i*100)
    l2.append(100-(i*100))

    # print the mean squared error
    print('training mean squared error: %.2f'% mean_squared_error(boston_y_train, bosto

    # print the r-squared
    print('training R-squared: %.2f' % r2_score(boston_y_train, boston_y_pred2))

    # print the mean squared error
    print('testing mean squared error: %.2f'% mean_squared_error(boston_y_test, boston_

    # print the r-squared
    print('testing R-squared: %.2f' % r2_score(boston_y_test, boston_y_pred1))

    models = ['Model_Testing','Model_Training']
    trainingMse.append(mean_squared_error(boston_y_train, boston_y_pred2))
    testingMse.append(mean_squared_error(boston_y_test, boston_y_pred1))
    MSElist = [mean_squared_error(boston_y_test, boston_y_pred1), mean_squared_error(bo
    Rlist = [r2_score(boston_y_test, boston_y_pred1), r2_score(boston_y_train, boston_y

    # Plot
    fig = plt.figure()
    plt.bar(models,MSElist)
    plt.xlabel("Models")
```

```
    plt.ylabel("Mean Sqaure Error")
    plt.title("Comparing Model testing and training")
    plt.show()

plt.bar(l1, testingMse, color ='maroon')
plt.xlabel("Testing MSE")
plt.ylabel("Mean Sqaure Error")
plt.title("Comparing Models")
plt.show()

plt2.bar(l2, trainingMse, color ='maroon')
plt2.xlabel("Training MSE")
plt2.ylabel("Mean Sqaure Error")
plt2.title("Comparing Models")
plt2.show()
```

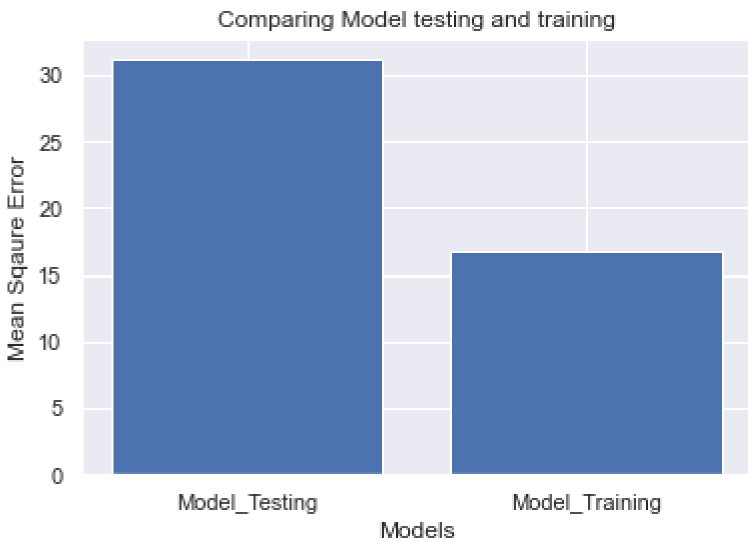
```
The testing percent is 20.0
training mean squared error: 21.41
training R-squared: 0.74
testing mean squared error: 24.68
testing R-squared: 0.72
```



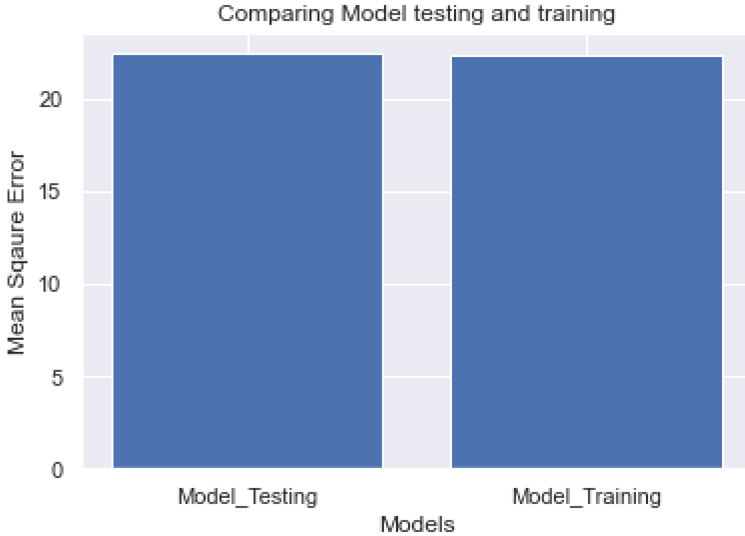

```
The testing percent is 30.0
training mean squared error: 20.20
training R-squared: 0.74
testing mean squared error: 27.51
testing R-squared: 0.72
```



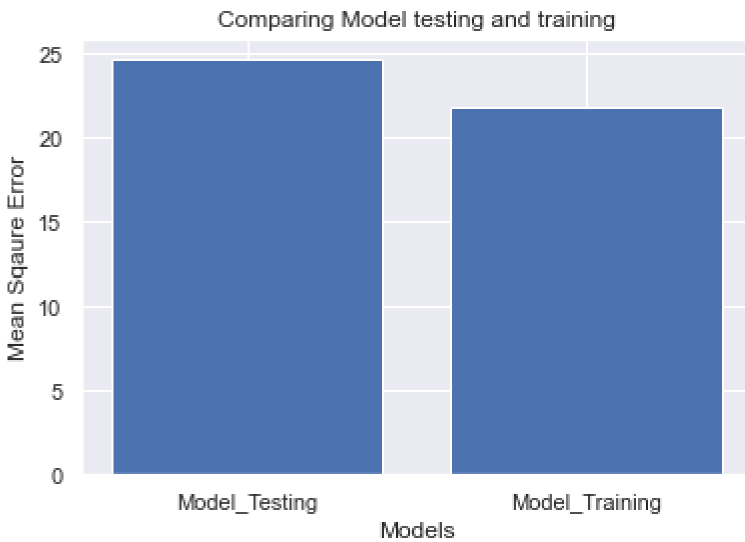

```
The testing percent is 40.0
training mean squared error: 16.69
training R-squared: 0.79
testing mean squared error: 31.18
testing R-squared: 0.65
```



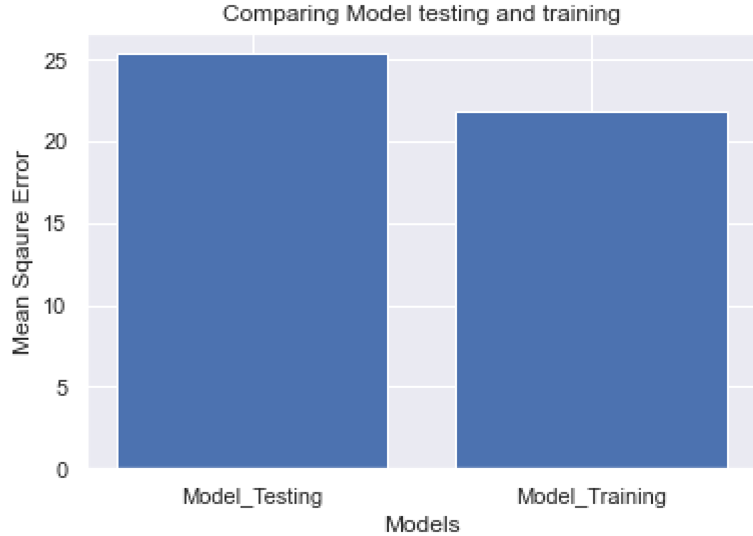

```
The testing percent is 50.0
training mean squared error: 22.41
training R-squared: 0.75
testing mean squared error: 22.46
testing R-squared: 0.71
```

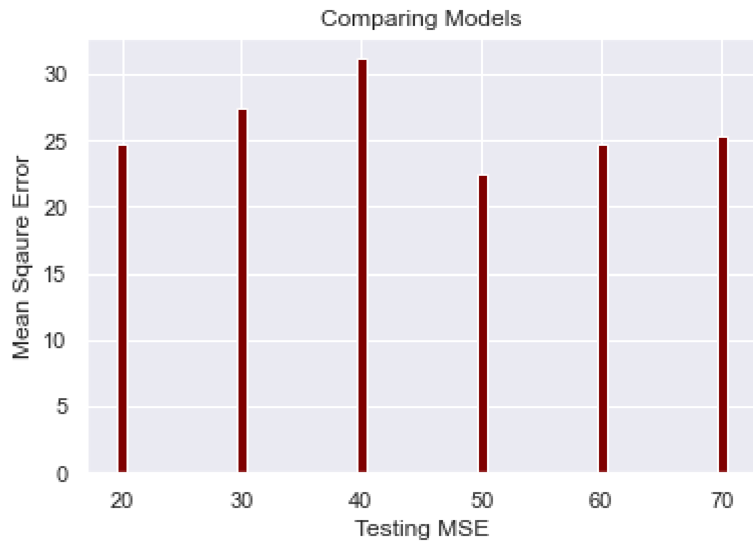


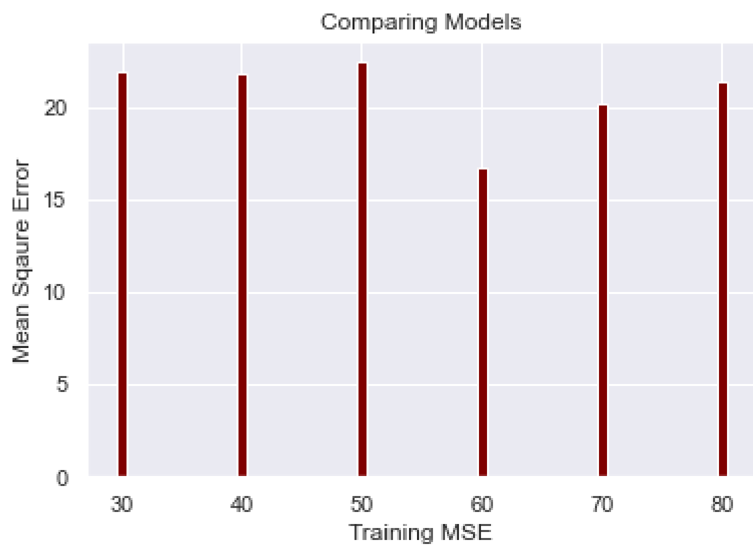

```
The testing percent is 60.0
training mean squared error: 21.80
training R-squared: 0.73
testing mean squared error: 24.67
testing R-squared: 0.72
```




```
The testing percent is 70.0
training mean squared error: 21.86
training R-squared: 0.80
testing mean squared error: 25.36
testing R-squared: 0.65
```







In [ ]:

```
#Your code goes here
```

Exercise 3.5 (30 pts)

Next, we want to use `RFE` to find the best set of features for prediction. When we used `RFE` in 3.2, we chose an arbitrary value for number of features. In practice, we don't know the best number of features to select for `RFE` , so we have to test different values. In this case, the number of features is the hyperparameter that we want to tune. Typically, we use cross validation for tuning hyperparameters.

Recall that cross validation is a technique where we split our data into equal folds and then use one fold for testing and the rest for training. We can use `KFold` Documentation form scikit learn `model_selection` to split our data into desired folds. We can also use `cross_val_score` Documentation to evaluate a score by cross validation.

For this exercise, use `RFE` with cross-validation (K = 5 aka 5fold) to find the best set of features for prediction. In order to do that, you need to consider all possible combination for number of features.

- Make an RFE model with i number of features
- Create a 5-fold CV on the data set
- Fit the model with Cross validation and store the MSE values
- Draw a box plot for MSE values of each model and pick the best number of features

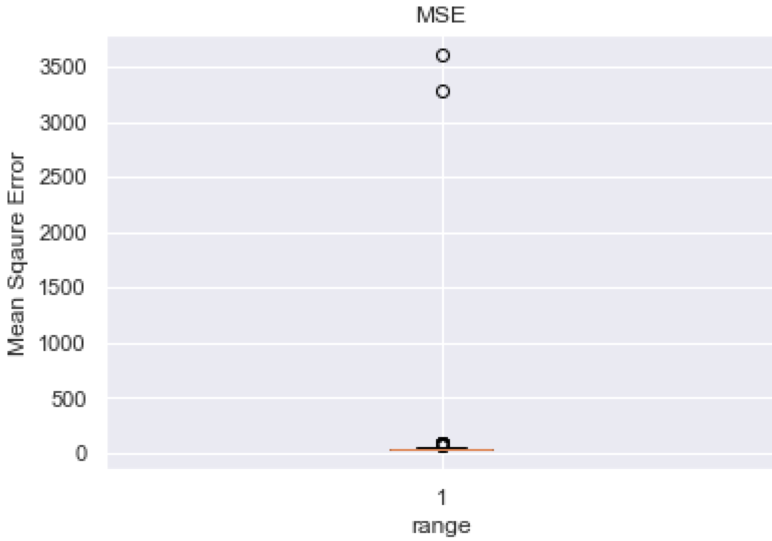
Note that the Boston housing data set contains 13 features, so you must create 13 models and use that model with 5-fold CV. At the end, you should have 13 models with 5 MSE values for each model (results of CV). Plot a side-by-side boxplot and compare the distrbution of MSE values for these 13 models. Then pick the model with lowest average MSE as the best result. What is the best number of features?

In [159…

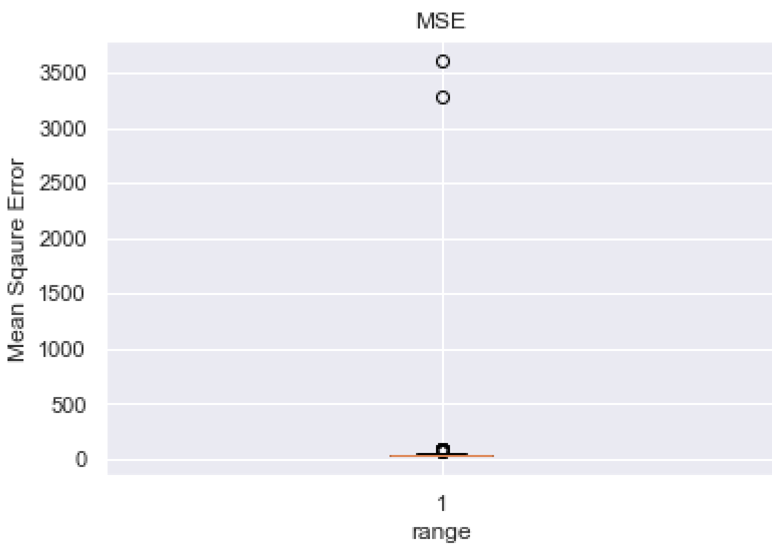
```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt2
import matplotlib.pyplot as plt
#Your code goes here
for i in range (1,14):
    print(i, "featureas")
    rfe1 = RFE(estimator = m1, n_features_to_select = i , step = 1)
    rfe1.fit(boston_X_train, boston_y_train)
    nike = KFold(n_splits = 5)
    cross_val_score(rfe1, boston_x, boston_y, cv=nike)
    boston_y_pred3 = rfe1.predict(boston_X_test)
    # print the mean squared error
    print('testing mean squared error: %.2f'% mean_squared_error(boston_y_test, boston_
    MSE.append(mean_squared_error(boston_y_test, boston_y_pred3))
    plt2.boxplot(MSE)
    plt2.xlabel("range")
    plt2.ylabel("Mean Sqaure Error")
    plt2.title("MSE")
    plt2.show()

for i in range (1,14):
    print(i, "featureas")
    rfe2 = RFE(estimator = m1, n_features_to_select = i , step = 1)
    rfe2.fit(boston_X_train, boston_y_train)
    nike = KFold(n_splits = 5)
```

```
    cross_val_score(rfe2, boston_x, boston_y, cv=nike)
    boston_y_pred4 = rfe2.predict(boston_X_test)
    # print the r-squared
    print('testing R-squared: %.2f' % r2_score(boston_y_test, boston_y_pred4))
    R2.append(r2_score(boston_y_test, boston_y_pred3))
    plt.boxplot(R2)
    plt.xlabel("range")
    plt.ylabel("R-square error")
    plt.title("R2")
    plt.show()
```

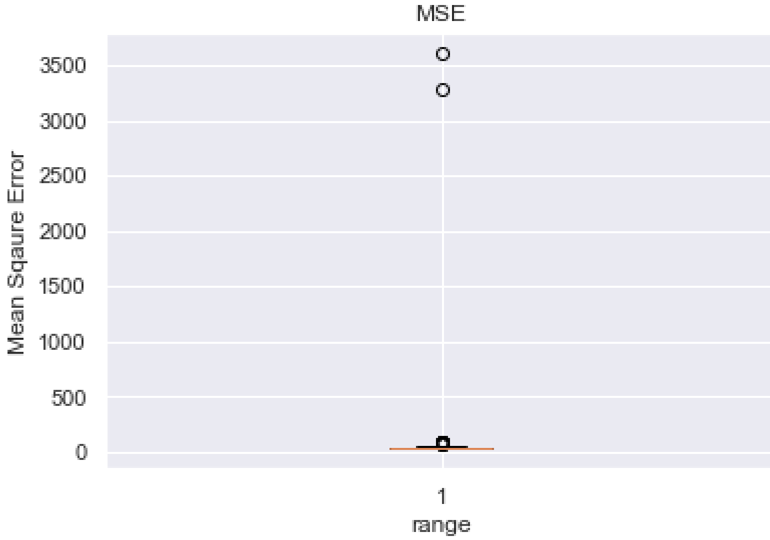
```
1 featureas
testing mean squared error: 62.12
```



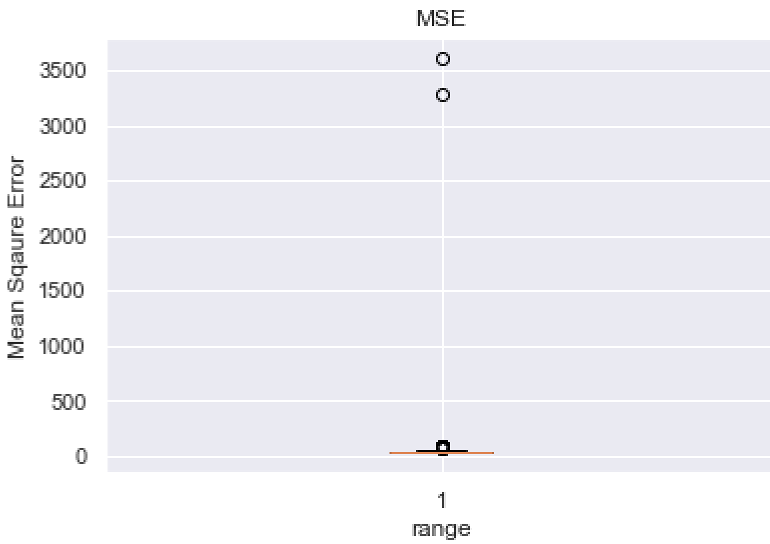

```
2 featureas
testing mean squared error: 40.68
```



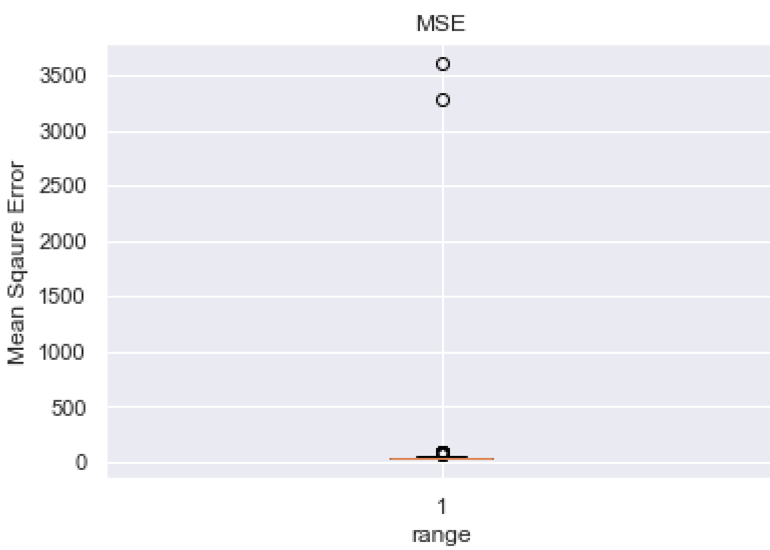

```
3 featureas
testing mean squared error: 39.70
```



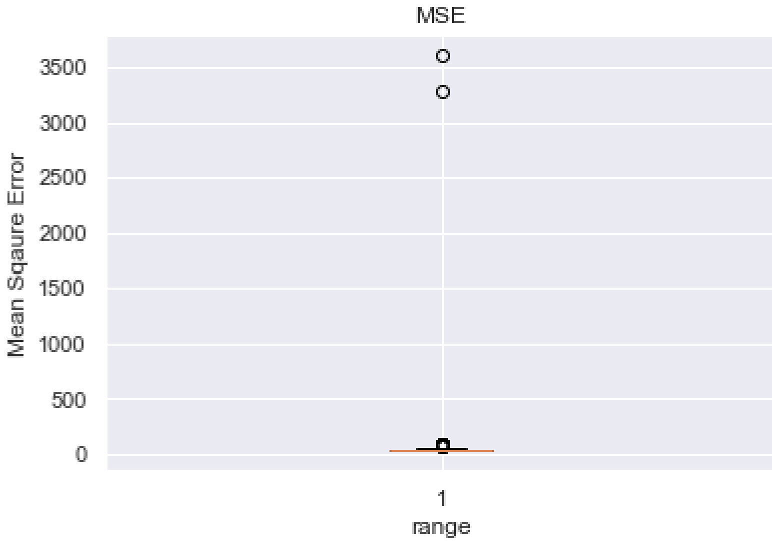

```
4 featureas
testing mean squared error: 34.33
```



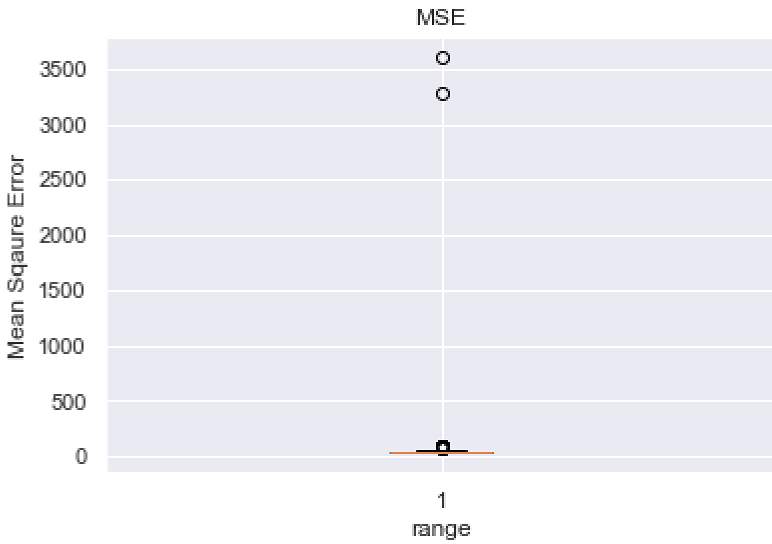

```
5 featureas
testing mean squared error: 32.95
```



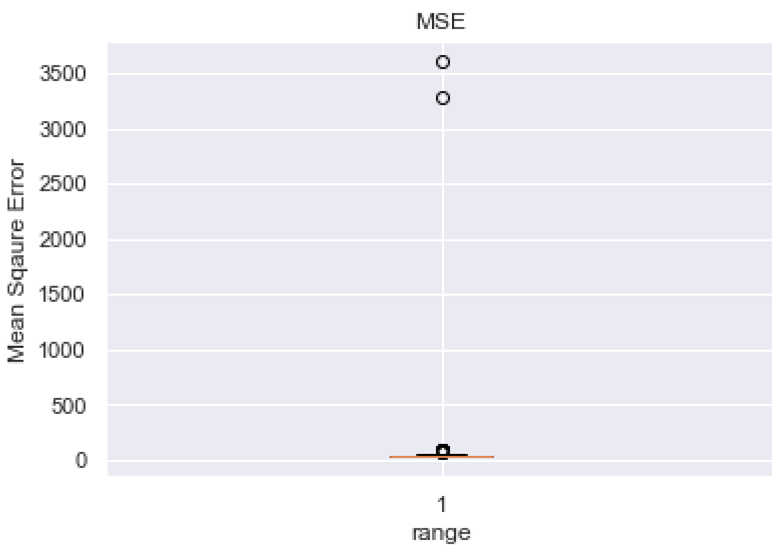

```
6 featureas
testing mean squared error: 26.82
```



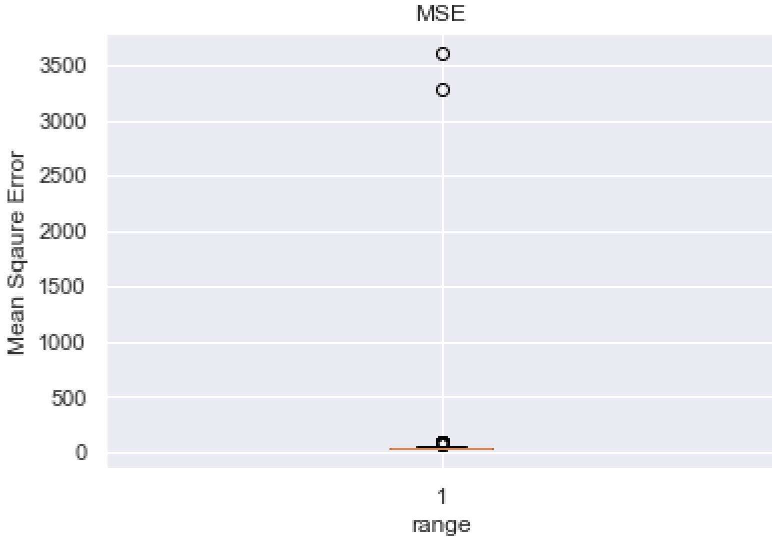

```
7 featureas
testing mean squared error: 27.00
```



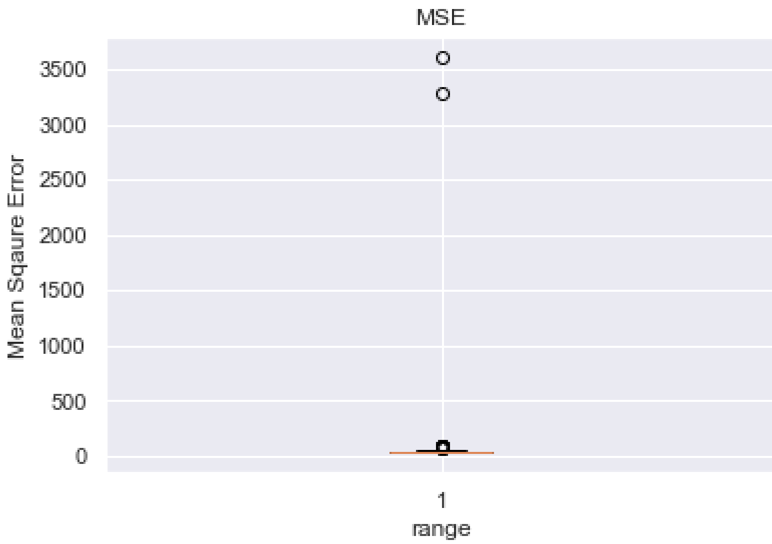

```
8 featureas
testing mean squared error: 26.51
```



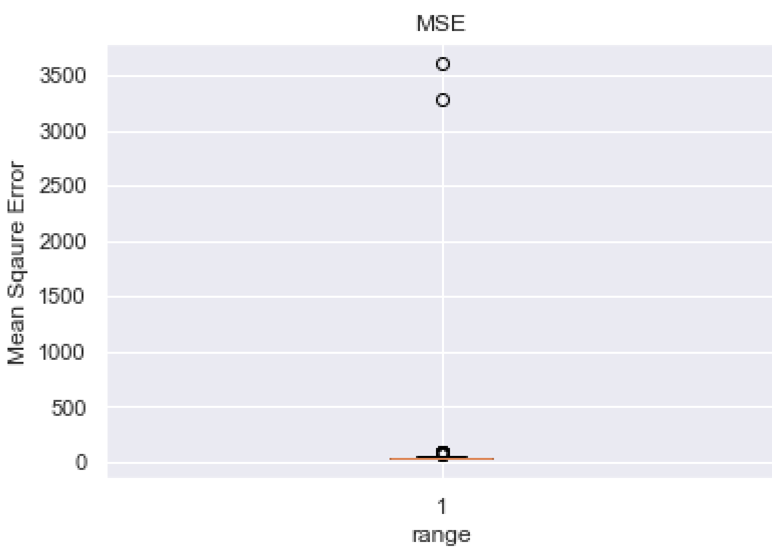

```
9 featureas
testing mean squared error: 26.46
```



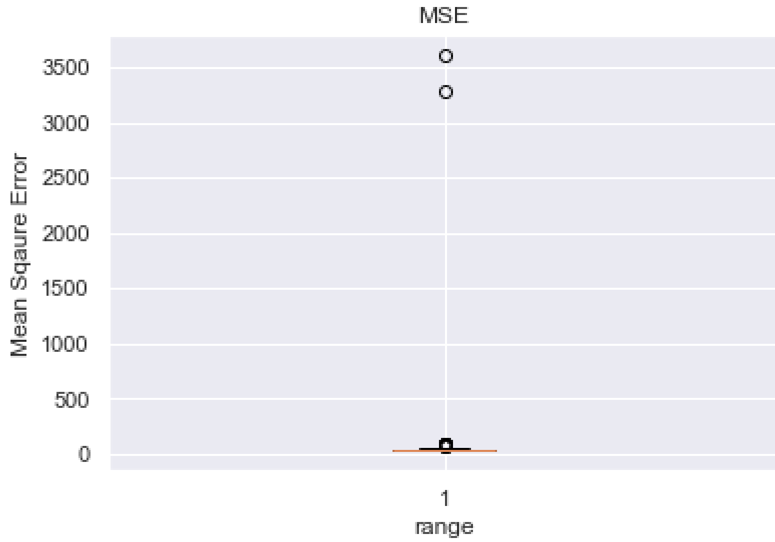

```
10 featureas
testing mean squared error: 26.26
```



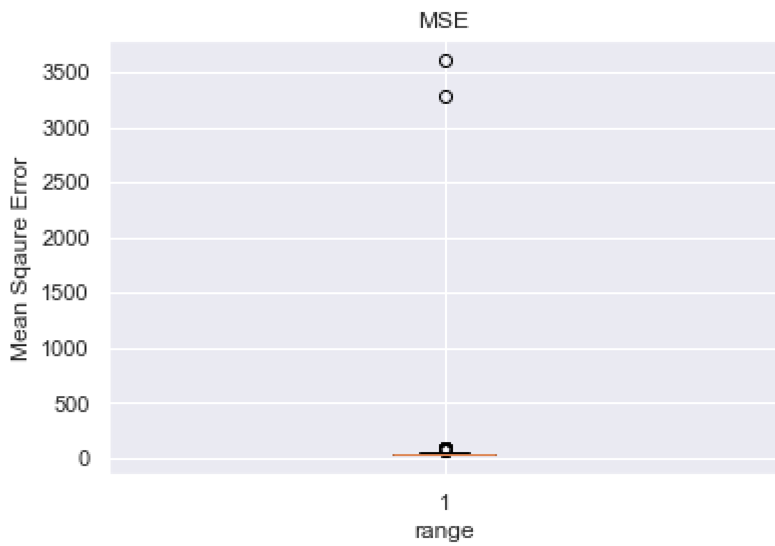

```
11 featureas
testing mean squared error: 26.13
```



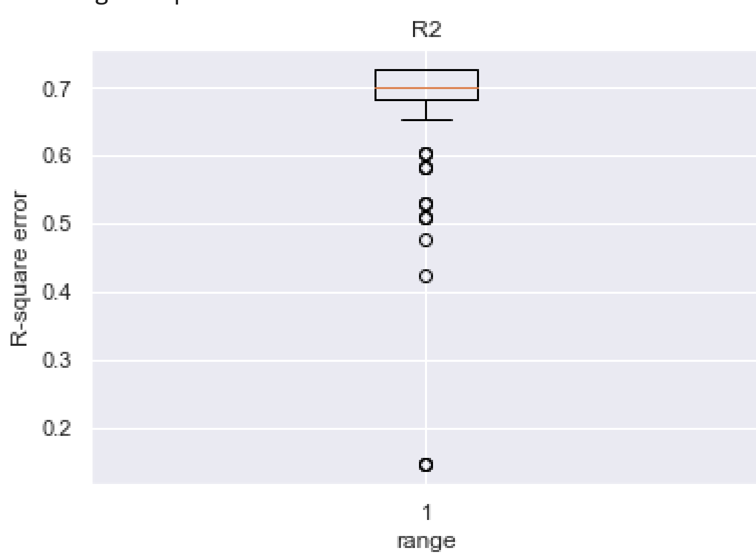

```
12 featureas
testing mean squared error: 26.24
```



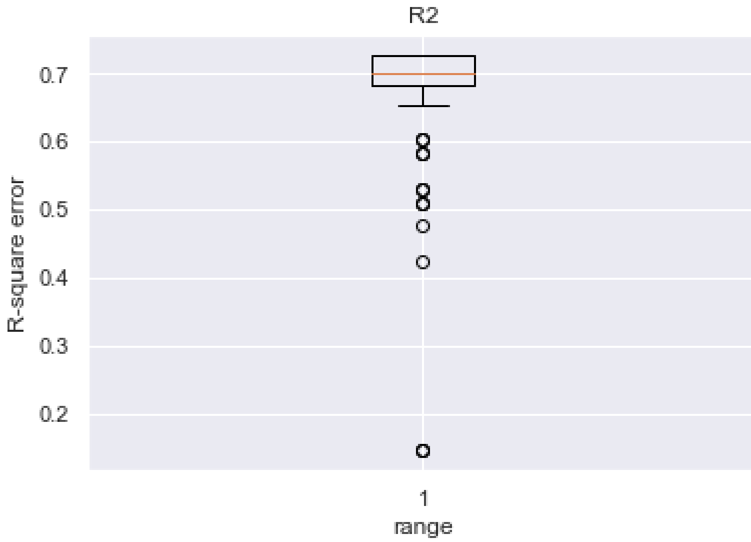

```
13 featureas
testing mean squared error: 25.36
```



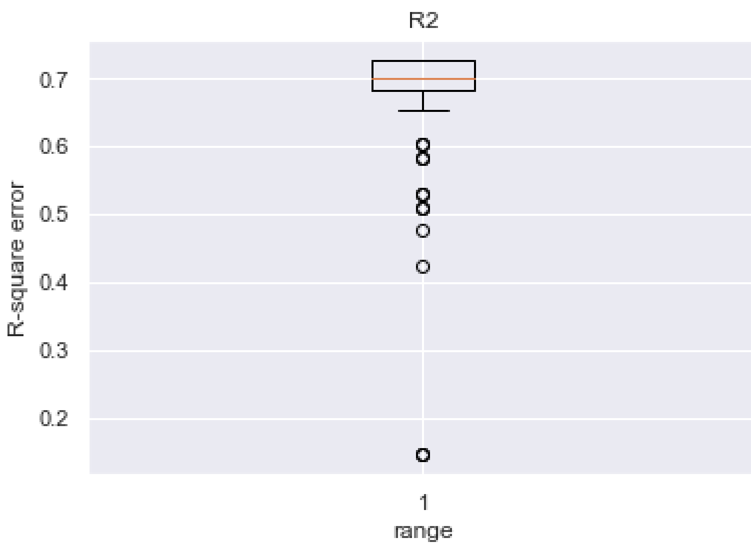

```
1 featureas
testing R-squared: 0.15
```



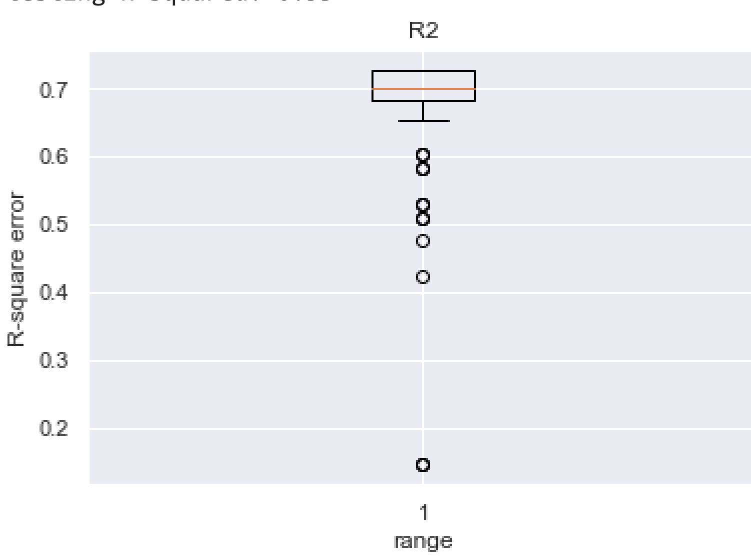

```
2 featureas
testing R-squared: 0.44
```



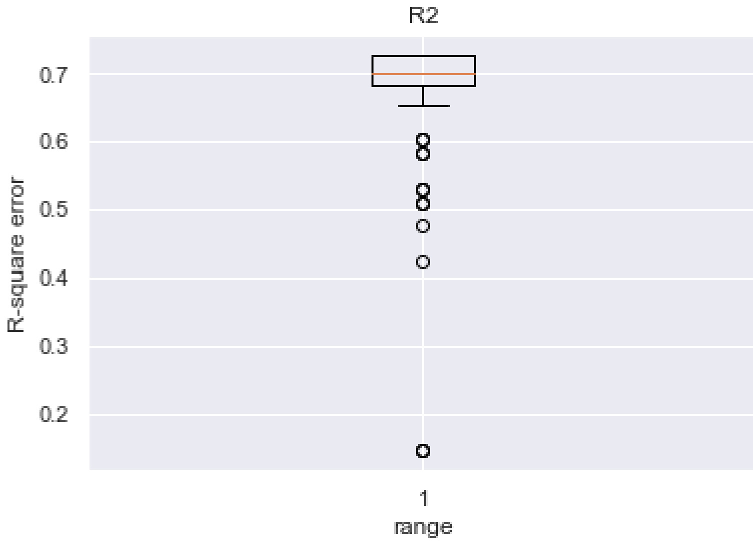

```
3 featureas
testing R-squared: 0.46
```



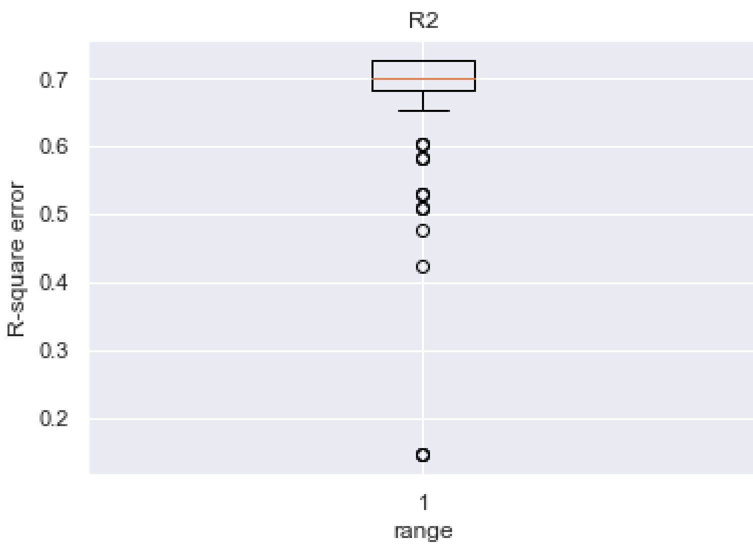

```
4 featureas
testing R-squared: 0.53
```



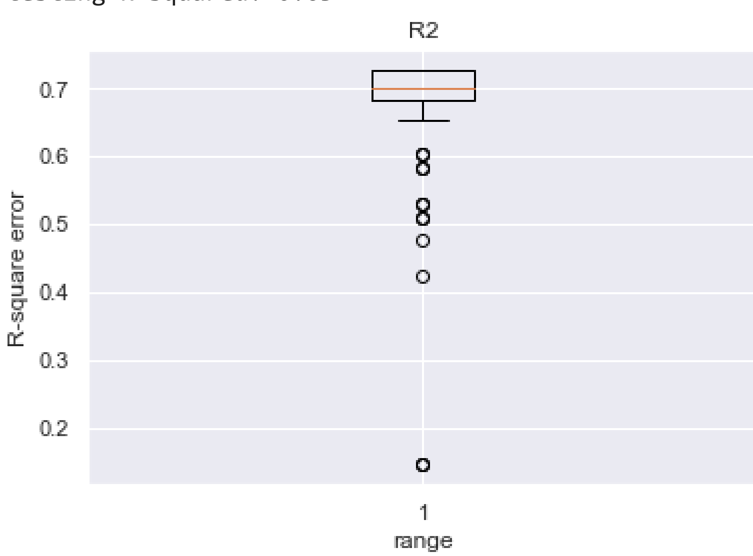

```
5 featureas
testing R-squared: 0.55
```



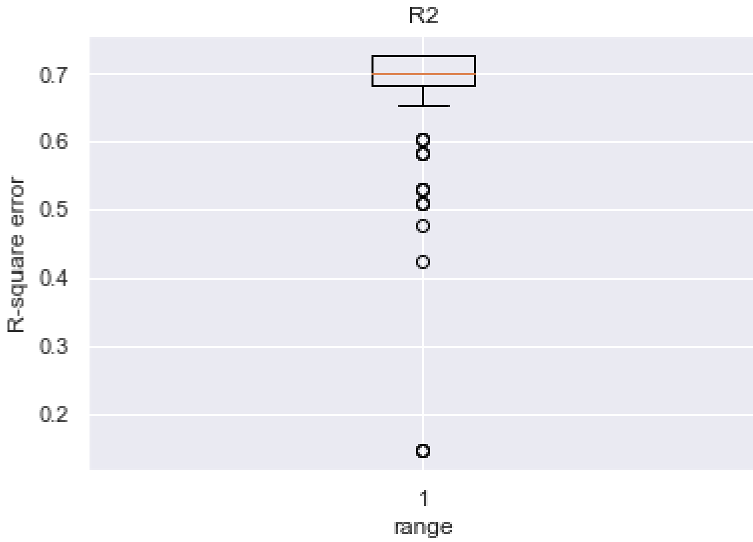

```
6 featureas
testing R-squared: 0.63
```



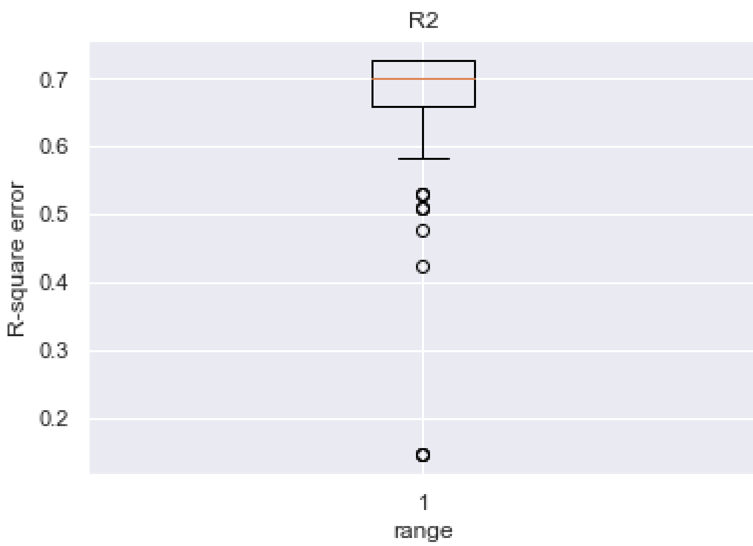

```
7 featureas
testing R-squared: 0.63
```



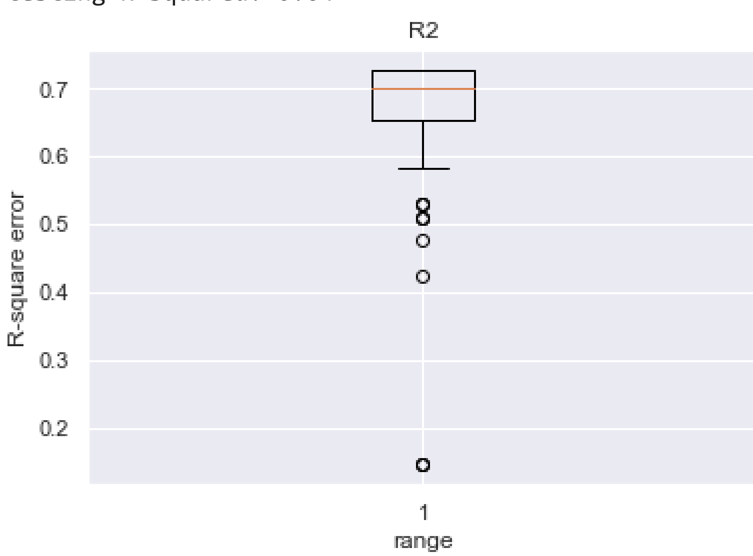

```
8 featureas
testing R-squared: 0.64
```



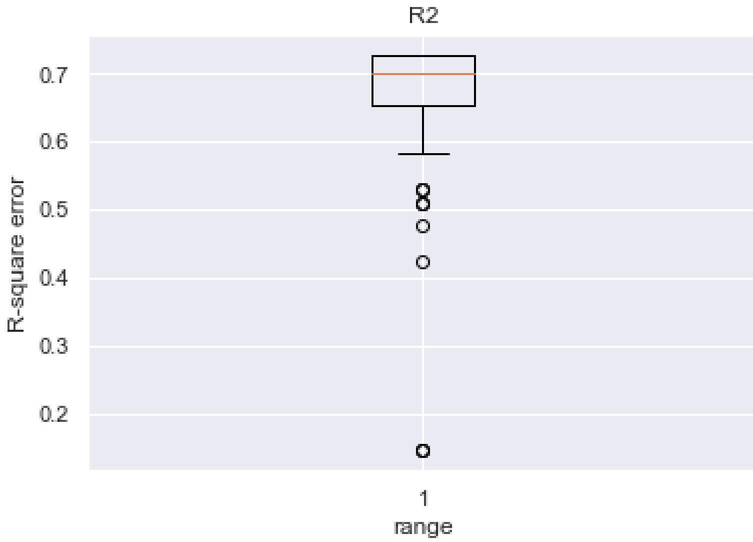

```
9 featureas
testing R-squared: 0.64
```



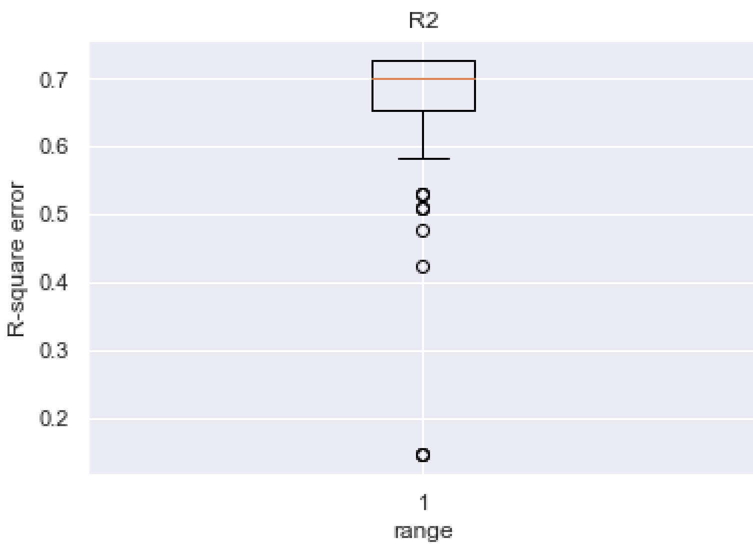

```
10 featureas
testing R-squared: 0.64
```



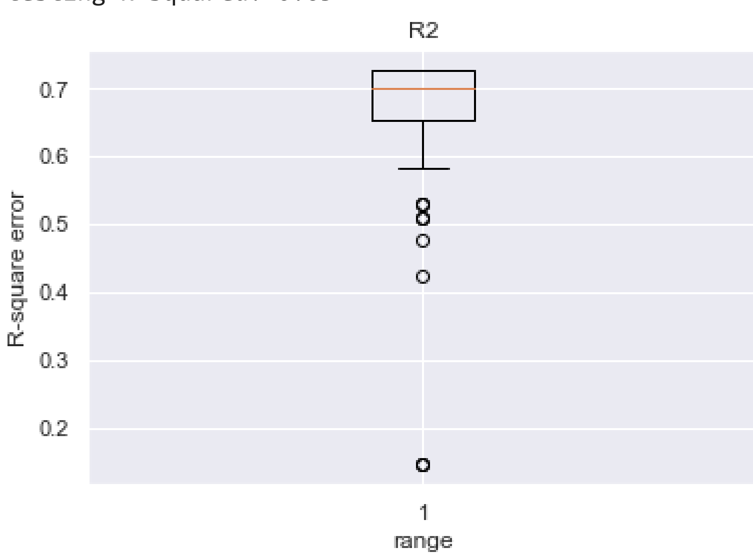

```
11 featureas
testing R-squared: 0.64
```




```
12 featureas
testing R-squared: 0.64
```




```
13 featureas
testing R-squared: 0.65
```



Exercise 3.5 (10 pts)

Another way of doing RFE with CV is using `RFECV` Documentation which performs the same task as `RFE` while performing a cross validation task. Use `RFECV` to find the best number of features for a regression model for Boston housing data set. Which features are selected? (you can use `ranking_` attribute)

In [156…

```
from sklearn.feature_selection import RFECV

#Your code goes here
selector = RFECV(estimator = m1, step=1, cv=5)
selector = selector.fit(boston_X_test, boston_y_test)
selector.ranking_
```

Out[156…

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

In [ ]:

Some Notes:

- For `scoring` parameter you can use `neg_mean_squared_error` to get MSE. You can read model evaluation documentation to learn what values you can pass for this parameter
- The default value for `cv` parameter in `RFECV` model is 5-fold cross-validation
- Your results for RFE may vary (values with different numerical precision) given the stochastic nature of the algorithm and evaluation process.
- Specifying random_state parameter ensures the same random partitions are used across different runs
- Read the documentation to learn more about parameters and attributes for each model we discussed.

In [ ]:

In [ ]: