Problem 1

2. B) Standardized and C) Normalized

Problem 2

3. Log Transformation

Problem 3

Mean for x_1 = 2
Mean for x_2 = -9

Deviations for x_1 = [8−2, 4−2, 0−2, −4−2] = [6 ,2 ,−2, −6]
Deviations for x_2 = [−16+9, −12+9, −10+9, 2+9] = [−7, −3, −1, 11]

Products of deviations: [-42, -6, 2, -66]

Sum of products = -112

Sum of squared deviations for x_1 = 36+4+4+36=80
Sum of squared deviations for x_2 = 49+9+1+121=180

Computing the Pearson correlation coefficient:
r = -112 / (sqrt(80 x 180))
r = -0.933

Problem 4

1. x_1 or 2. x_2 can be removed depending on the target index

M6-L1 Problem 1

MinMax Scaling

MinMax scaling scales the data such that the minimum in each column is transformed to 0, and the maximum in each column is transformed to 1, using the formula $X' = \frac{X - X_{min}}{X_{max} - X_{min}}$.

For example, for a training matrix X, MinMax scaling will give:

$$X = \begin{bmatrix} 10 & 2 \\ 9 & 0 \\ 3 & 8 \\ 4 & 9 \\ 0 & 8 \\ -3 & 2 \\ -4 & -2 \\ 5 & 6 \end{bmatrix} \xrightarrow{X} \begin{bmatrix} 1. & 0.364 \\ 0.929 & 0.182 \\ 0.5 & 0.909 \\ 0.571 & 1. \\ 0.286 & 0.909 \\ 0.071 & 0.364 \\ 0. & 0. \\ 0.643 & 0.727 \end{bmatrix}$$

Applying the same scaling to the given matrix of test data `A` should yield the following (notice we are scaling according to `X`, not `A`).

$$A = \begin{bmatrix} 2 & 1 \\ 5 & 5 \\ 11 & -1 \\ -3 & 2 \end{bmatrix} \xrightarrow{X} \begin{bmatrix} 0.429 & 0.273 \\ 0.643 & 0.636 \\ 1.071 & 0.091 \\ 0.071 & 0.364 \end{bmatrix}$$

Implementing MinMax Scaling

A function to compute the minimum and maximum of each column is provided. Complete the scaling function `MinMax_scaler(X, Min, Max)` below to implement $X' = \frac{X - X_{min}}{X_{max} - X_{min}}$.

Validate your results by comparing to the above.

In [8]:
```
import numpy as np
np.set_printoptions(precision=3)

def get_MinMax(X):
    Max = np.max(X,axis=0).reshape(1,-1)
    Min = np.min(X,axis=0).reshape(1,-1)
    return Min, Max

def MinMax_scaler(X, Min, Max):
    # YOUR CODE GOES HERE
    # Scale values such that Max --> 1 and Min --> 0
    sig = Max - Min
    sig[sig == 0] = 1

    X_ = np.divide(X - Min, sig)
    return X_

# Loading train data X and test data A:
X = np.array([[10,9,3,4,0,-3,-4,5],[2,0,8,9,8,2,-2,6]]).T
A = np.array([[2,5,11,-3],[1,5,-1,2]]).T

# Getting the scaling constants for each column of X:
Xmin, Xmax = get_MinMax(X)

# Scaling X and A, and printing the results
print("X =\n", X, " -->\n", MinMax_scaler(X,Xmin,Xmax))
print("A =\n", A, " -->\n", MinMax_scaler(A,Xmin,Xmax))
```

```
X =
 [[10  2]
 [ 9  0]
 [ 3  8]
 [ 4  9]
 [ 0  8]
 [-3  2]
 [-4 -2]
 [ 5  6]]  -->
 [[1.    0.364]
 [0.929 0.182]
 [0.5   0.909]
 [0.571 1.   ]
 [0.286 0.909]
 [0.071 0.364]
 [0.    0.   ]
 [0.643 0.727]]
A =
 [[ 2  1]
 [ 5  5]
 [11 -1]
 [-3  2]]  -->
 [[0.429 0.273]
 [0.643 0.636]
 [1.071 0.091]
 [0.071 0.364]]
```

Standard Scaling

Standard scaling scales the data according to the mean (μ) and standard deviation (σ) of the training data. Scaling uses the formula $X' = \frac{X-\mu}{\sigma}$.

For example, for a training matrix X, Standard scaling will give:

$$X = \begin{bmatrix} 10 & 2 \\ 9 & 0 \\ 3 & 8 \\ 4 & 9 \\ 0 & 8 \\ -3 & 2 \\ -4 & -2 \\ 5 & 6 \end{bmatrix} \xrightarrow{X} \begin{bmatrix} 1.46 & -0.547 \\ 1.251 & -1.061 \\ 0. & 0.997 \\ 0.209 & 1.254 \\ -0.626 & 0.997 \\ -1.251 & -0.547 \\ -1.46 & -1.576 \\ 0.417 & 0.482 \end{bmatrix}$$

Applying the same scaling to the given matrix of test data `A` should yield the following (notice we are scaling according to `X`, not `A`).

$$A = \begin{bmatrix} 2 & 1 \\ 5 & 5 \\ 11 & -1 \\ -3 & 2 \end{bmatrix} \xrightarrow{X} \begin{bmatrix} -0.209 & -0.804 \\ 0.417 & 0.225 \\ 1.668 & -1.318 \\ -1.251 & -0.547 \end{bmatrix}$$

Implementing Standard Scaling

A function to compute the `mu` and `sigma` of each column is provided. Complete the scaling function `Standard_scaler(X, Min, Max)` below to implement $X' = \frac{X-\mu}{\sigma}$.

Validate your results by comparing to the above.

In [9]:
```python
import numpy as np
np.set_printoptions(precision=3)

def get_MuSigma(X):
    mu = np.mean(X,axis=0).reshape(1,-1)
    sigma = np.std(X,axis=0).reshape(1,-1)
    return mu, sigma

def Standard_scaler(X, mu, sigma):
    # YOUR CODE GOES HERE
    # Scale values such that mu --> 0 and sigma --> 1
    sigma[sigma == 0] = 1

    X_ = np.divide(X - mu, sigma)
```

```
    return X_

# Loading train data X and test data A:
X = np.array([[10,9,3,4,0,-3,-4,5],[2,0,8,9,8,2,-2,6]]).T
A = np.array([[2,5,11,-3],[1,5,-1,2]]).T

# Getting the scaling constants for each column of X:
Xmu, Xsigma = get_MuSigma(X)

# Scaling X and A, and printing the results
print("X =\n", X, " -->\n", Standard_scaler(X,Xmu,Xsigma))
print("A =\n", A, " -->\n", Standard_scaler(A,Xmu,Xsigma))
```

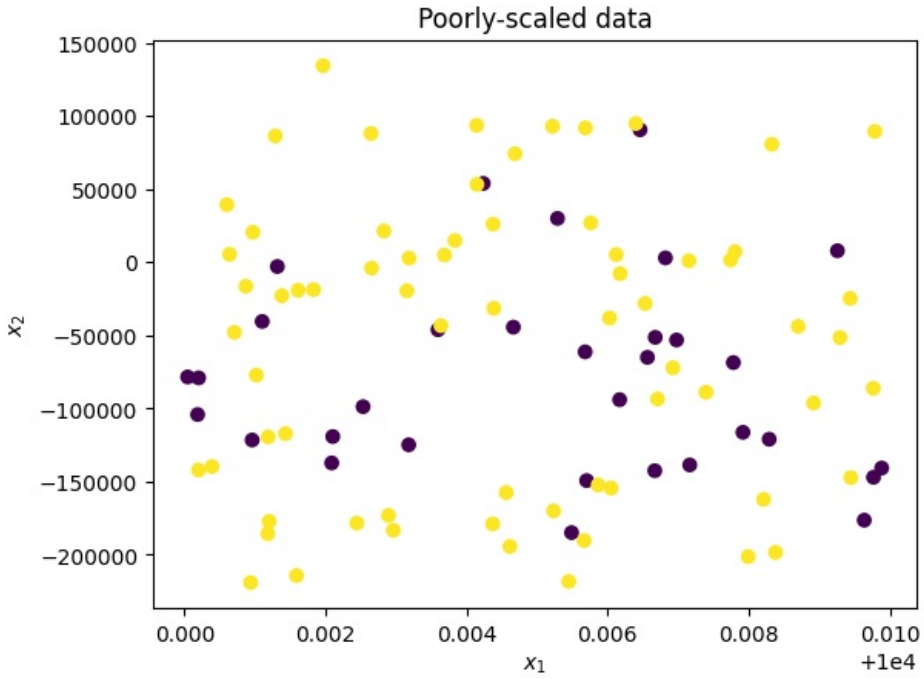
```
X =
 [[10  2]
 [ 9  0]
 [ 3  8]
 [ 4  9]
 [ 0  8]
 [-3  2]
 [-4 -2]
 [ 5  6]]  -->
 [[ 1.46  -0.547]
 [ 1.251 -1.061]
 [ 0.     0.997]
 [ 0.209  1.254]
 [-0.626  0.997]
 [-1.251 -0.547]
 [-1.46  -1.576]
 [ 0.417  0.482]]
A =
 [[ 2  1]
 [ 5  5]
 [11 -1]
 [-3  2]]  -->
 [[-0.209 -0.804]
 [ 0.417  0.225]
 [ 1.668 -1.318]
 [-1.251 -0.547]]
```

M6-L1 Problem 2

In this problem you'll learn how to make a 'pipeline' in SciKit-Learn. A pipeline chains together multiple sklearn modules and runs them in series. For example, you can create a pipeline to perform feature scaling and then regression. For more information see https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/

First, run the cell below to import modules and load data. Note the data axis scaling.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

x1 = np.array([10000.00548814, 10000.00715189, 10000.00602763, 10000.00544883, 10000.00423655, 10000.00645894,
x2 = np.array([-184863.4856705 ,    1074.38382588,  -38090.38042426, -218261.93176495,   53942.6974416 ,   9063
X = np.vstack([x1,x2]).T
y = np.array([0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0, train_size=0.8)

plt.figure()
plt.scatter(x1,x2,c=y,cmap="viridis")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.title("Poorly-scaled data")
plt.show()
```



Creating a pipeline

In this section, code to set up a pipeline has been given. Make note of how each step works:

1. Create a scaler and classifier
2. Put the scaler and classifier into a new pipeline
3. Fit the pipeline to the training data
4. Make predictions with the pipeline

In [2]:
```
# Create a scaler and a classifier
scaler = MinMaxScaler()
model = KNeighborsClassifier()

# Put the scaler and classifier into a new pipeline
pipeline = Pipeline([("MinMax Scaler", scaler), ("KNN Classifier", model)])
```

```
# Fit the pipeline to the training data
pipeline.fit(X_train, y_train)

# Make predictions with the pipeline
pred_train = pipeline.predict(X_train)
pred_test = pipeline.predict(X_test)
print("Training accuracy:", accuracy_score(y_train, pred_train), "   Testing accuracy:", accuracy_score(y_test,
```

```
Training accuracy: 0.825    Testing accuracy: 0.6
```

Testing several pipelines

Now, complete the code to create a new pipeline for every combination of scalers and models below:

Scalers:

- None
- MinMax
- Standard

Classifiers:

- Logistic Regression
- Support Vector Machine
- KNN Classifier, 1 neighbor

Within the loop, a scaler and model are created. You will create a pipeline, fit it to the training data, and make predictions on testing and training data.

In [3]:
```
def get_scaler(i):
    if i == 0:
        return ("No Scaler", None)
    elif i == 1:
        return ("MinMax Scaler", MinMaxScaler())
    elif i == 2:
        return ("Standard Scaler", StandardScaler())

def get_model(i):
    if i == 0:
        return ("Logistic Regression", LogisticRegression())
    elif i == 1:
        return ("Support Vector Classifier", SVC())
    elif i == 2:
        return ("1-NN Classifier", KNeighborsClassifier(n_neighbors=1))

for scaler_index in range(3):
    for model_index in range(3):
        scaler = get_scaler(scaler_index)
        model = get_model(model_index)

        # YOUR CODE GOES HERE
        # Create a pipeline
        pipeline = Pipeline([get_scaler(scaler_index), get_model(model_index)])
        # Fit the pipeline on X_train, y_train
        pipeline.fit(X_train, y_train)
        # Calculate acc_train and acc_test for the pipeline
        pred_train = pipeline.predict(X_train)
        acc_train = accuracy_score(y_train, pred_train)

        pred_test = pipeline.predict(X_test)
        acc_test = accuracy_score(y_test, pred_test)

        print(f"{scaler[0]:>15},{model[0]:>26}:    Train Acc. = {100*acc_train:5.1f}%    Test Acc. = {100*acc_t
```

```
     No Scaler,       Logistic Regression:    Train Acc. =  67.5%    Test Acc. =  70.0%
     No Scaler, Support Vector Classifier:    Train Acc. =  78.8%    Test Acc. =  65.0%
     No Scaler,           1-NN Classifier:    Train Acc. = 100.0%    Test Acc. =  50.0%
  MinMax Scaler,       Logistic Regression:    Train Acc. =  67.5%    Test Acc. =  70.0%
  MinMax Scaler, Support Vector Classifier:    Train Acc. =  67.5%    Test Acc. =  70.0%
  MinMax Scaler,           1-NN Classifier:    Train Acc. = 100.0%    Test Acc. =  85.0%
Standard Scaler,       Logistic Regression:    Train Acc. =  67.5%    Test Acc. =  70.0%
Standard Scaler, Support Vector Classifier:    Train Acc. =  68.8%    Test Acc. =  70.0%
Standard Scaler,           1-NN Classifier:    Train Acc. = 100.0%    Test Acc. =  85.0%
```

Questions

Answer the following questions:

1. Which model's testing accuracy was improved the most by scaling data?

2. Which performs better on this data: MinMax scaler, Standard scaler, or neither?

Answers

1. The 1-NN Classifier had a test accuracy of 50% without feature scaling, but with feature scaling, the test accuracy jumped to 85% with both MinMax scaling and Standard scaling.

2. From the output, both scaling methods have similar effects on the model accuracy for all three models. Data scaling does not seem to increase the train accuracy by much for the Logistic Regression model. On the other hand, both scaling methods cause the train accuracy to drop for the Support Vector Classifier! There is a significant improvement in the test accuracy for the 1-NN classifier model, but overall the performance improvement on the test data due to both scaling methods is very similar.

M6-L1 Problem 3

SciKit-Learn only offers a few built-in preprocessors, such as MinMax and Standard scaling. However, it also offers the ability to create custom data transformation functions, which can be integrated into your pipeline. In this problem, you will implement a log transform and observe how using it changes a regression result.

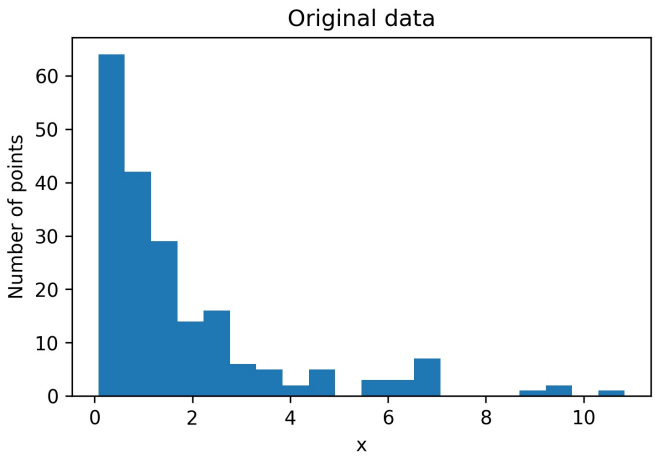
Start by running this cell to import modules and load data:

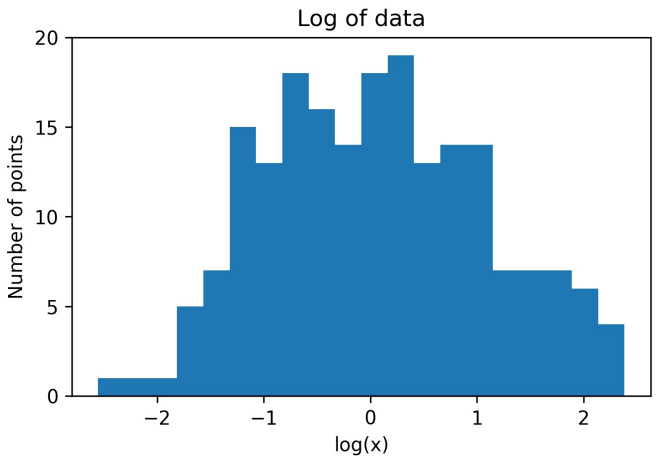
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer
from sklearn.svm import SVR

def plot(X_train, X_test, y_train, y_test, model=None,log = False):
    plt.figure(figsize=(5,5),dpi=200)
    if model is not None:
        X_fit = np.linspace(min(X_train)-0.15,max(X_train)+0.2)
        y_fit = model.predict(X_fit)
        plt.plot(np.log(X_fit+1) if log else X_fit, y_fit,c="red",label="Prediction")
    if log:
        X_train = np.log(X_train+1)
        X_test = np.log(X_test+1)

    plt.scatter(X_train,y_train,s=30,c="powderblue",edgecolors="navy",linewidths=.5,label="Train")
    plt.scatter(X_test,y_test,s=30,c="orange",edgecolors="red",linewidths=.5,label="Test")
    plt.legend()
    plt.xlabel("log(x+1)" if log else "x")
    plt.ylabel("y")
    plt.show()

x = np.array([ 5.83603919,  1.49205924,  2.66109578,  9.40172515,  6.47247125, 0.37633413,  2.58593829,  0.8595
X = x.reshape(-1,1)
y = np.array([ 4.32538472e+00, -5.59312420e+00, -4.57455876e+00,  4.23667057e+01, 1.04907251e+01, -4.16547735e+(

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1, train_size=0.8)
```

Distribution of x data

Let's visualize how the original input feature is distributed, alongside the log of the data -- notice that performing this log transformation makes the data much closer to normally distributed.

In [2]:
```
plt.figure(figsize=(12,3.4),dpi=300)
plt.subplot(1,2,1)
plt.hist(x,bins=20)
plt.xlabel("x")
plt.ylabel("Number of points")
plt.title("Original data")

plt.subplot(1,2,2)
plt.hist(np.log(x),bins=20)
plt.xlabel("log(x)")
plt.ylabel("Number of points")
plt.title("Log of data")
plt.ylim(0,20)
plt.yticks([0,5,10,15,20])

plt.show()
```



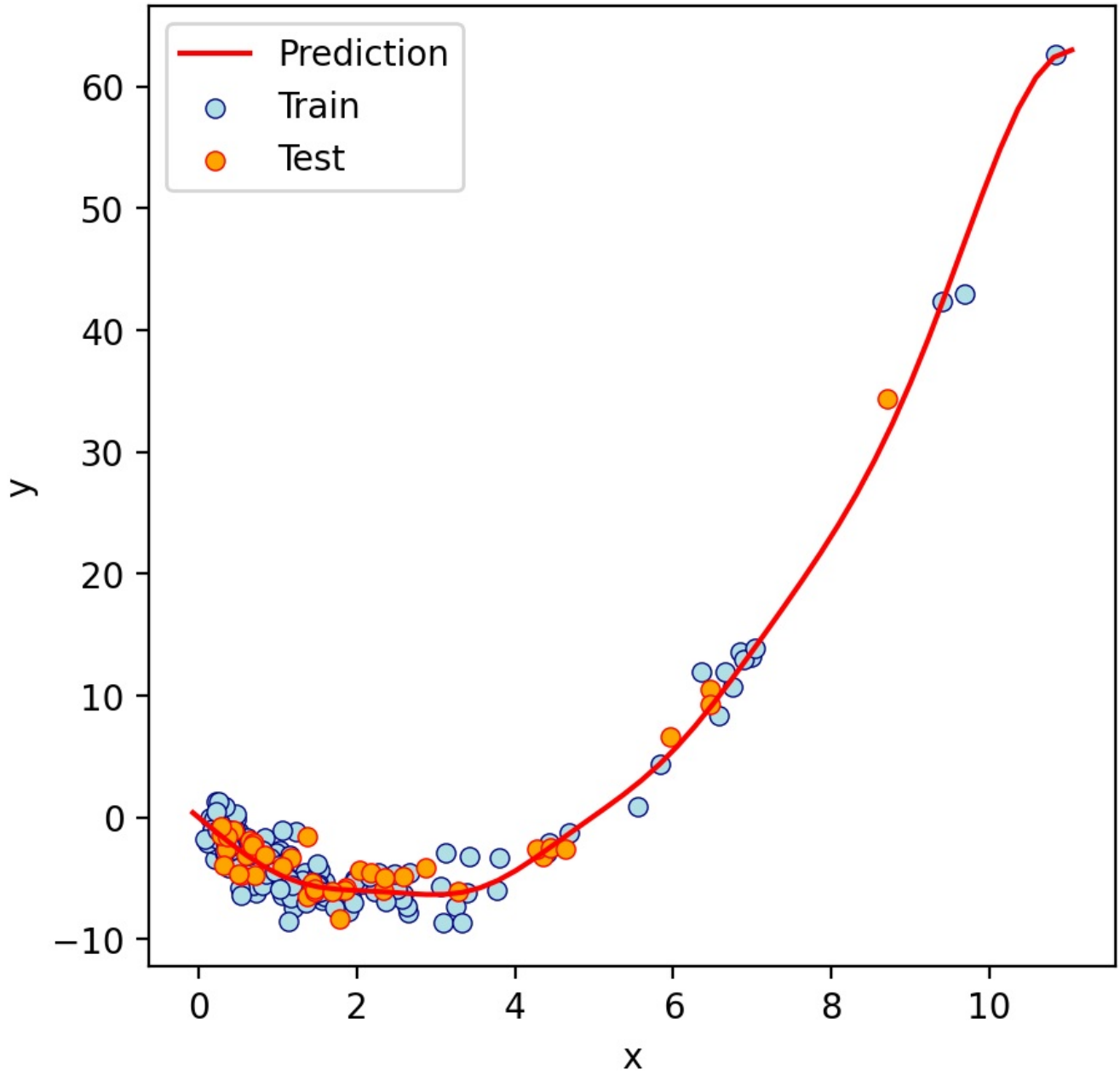


No log transform

First, we do support vector regression on the untransformed inputs. The code to do this has been provided below.

In [3]:
```
model = SVR(C=100)

pipeline = Pipeline([("SVR", model)])

# Fit the pipeline to the training data
pipeline.fit(X_train, y_train)

# Make predictions with the pipeline
pred_train = pipeline.predict(X_train)
pred_test = pipeline.predict(X_test)
print("Training MSE:", mean_squared_error(y_train, pred_train), "   Testing MSE:", mean_squared_error(y_test, p

# Plot the predictions
plot(X_train, X_test, y_train, y_test, pipeline)
```

```
Training MSE: 2.071006155233616    Testing MSE: 1.9453578716771704
```



With log transform

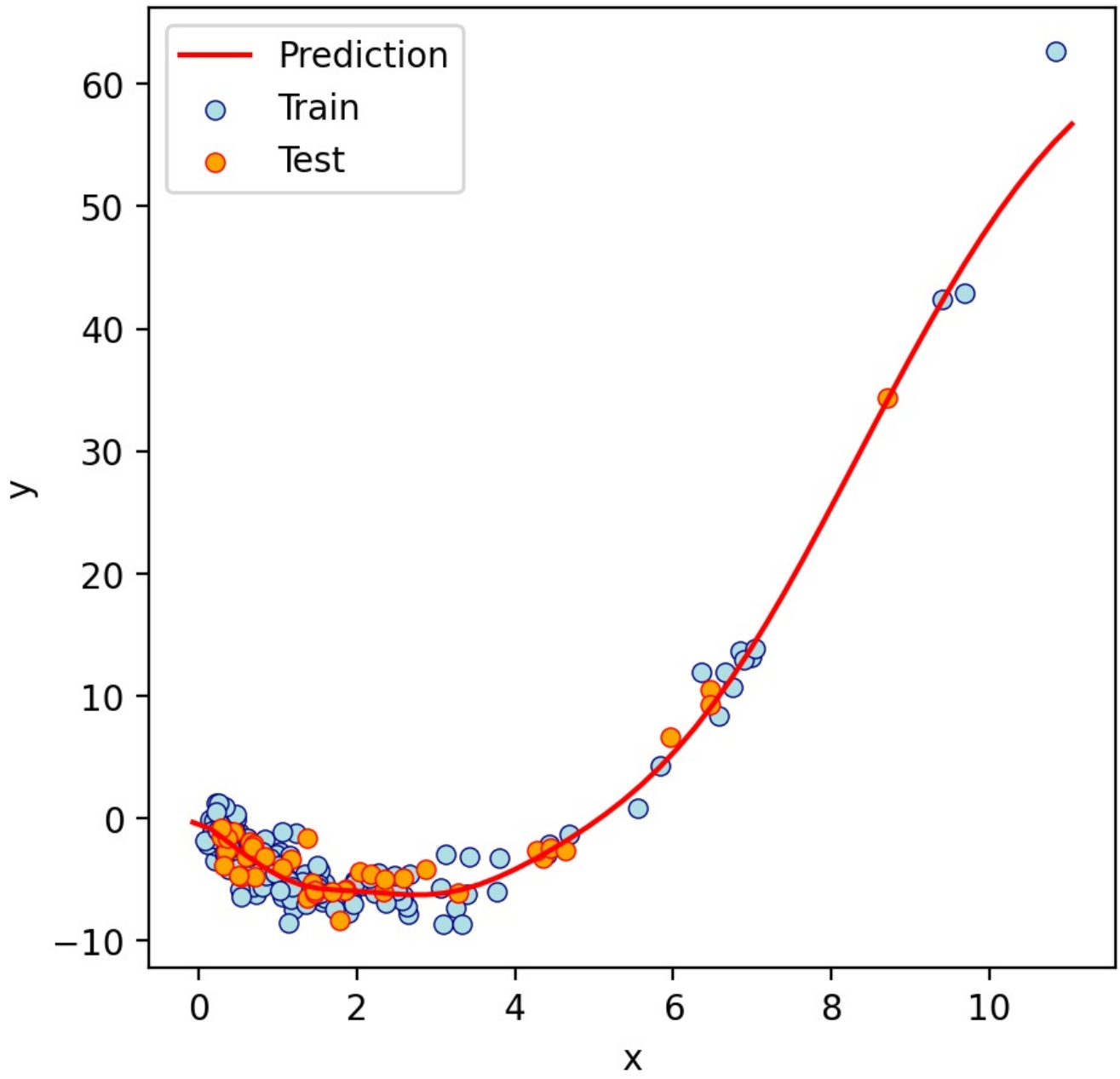
Notice that the data are not spread uniformly across the x axis. Instead, most input data points have low values -- this is a roughly "log normal" distribution. If we take the log of the input, we saw it was more normally distributed, which can improve machine learning model results in some cases. The transform function has been given below. Add this to a new pipeline, train the pipeline, and compute the train MSE and test MSE. Show a plot as above. Note the subtle change in behavior of the fitting curve.

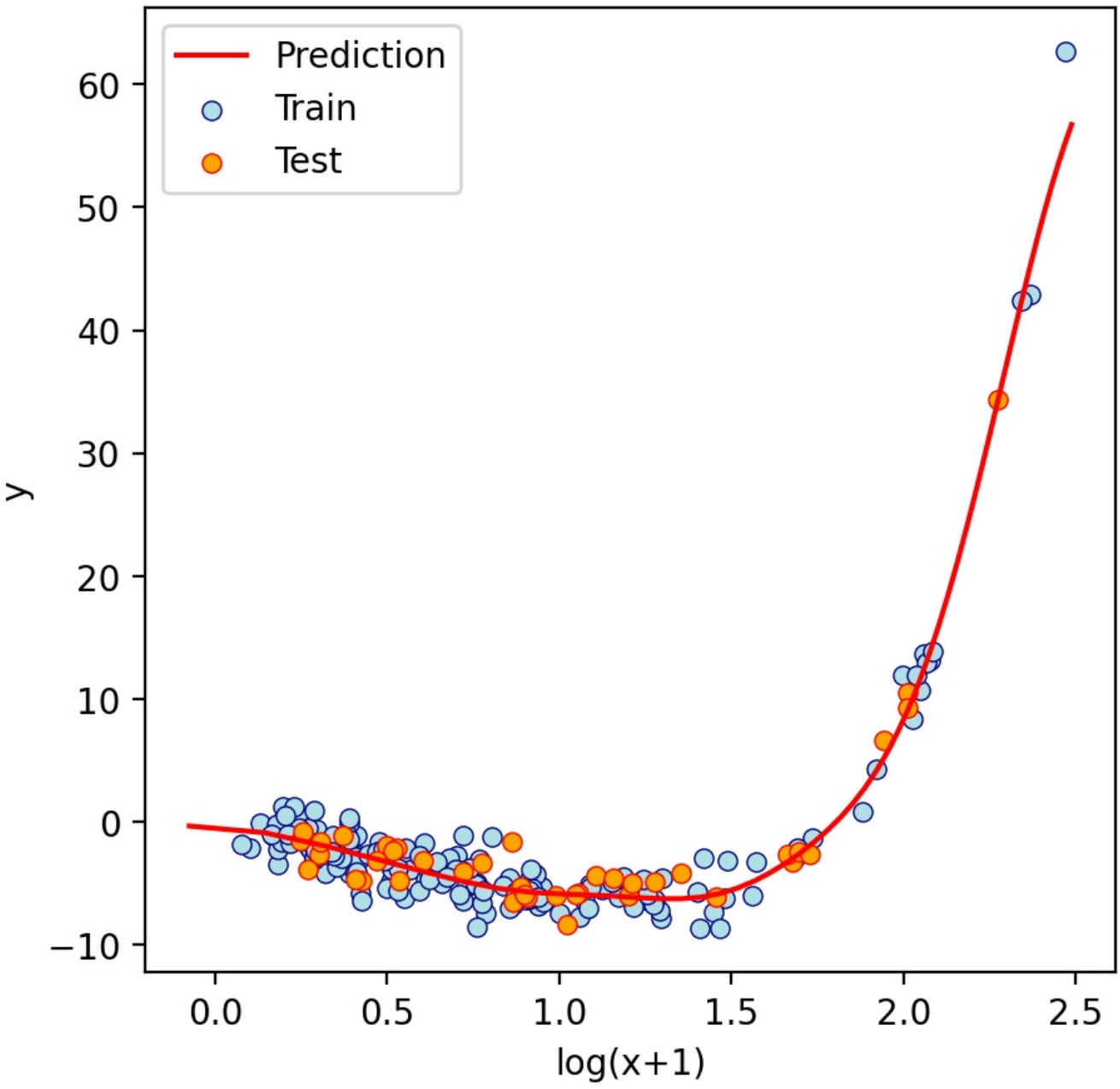
Also, make another plot setting the `log` argument to `True`. This will show the scaling of the x-axis used by the model.

In [6]:
```
def log_transform(x):
    return np.log(x + 1.)

transform = FunctionTransformer(log_transform)
model = SVR(C=100)


# YOUR CODE GOES HERE
pipeline = Pipeline([("Log Scaler", transform), ("SVR", model)])

# Fit the pipeline to the training data
pipeline.fit(X_train, y_train)

# Make predictions with the pipeline
pred_train = pipeline.predict(X_train)

pred_test = pipeline.predict(X_test)
print("Training MSE:", mean_squared_error(y_train, pred_train), "   Testing MSE:", mean_squared_error(y_test, p

# Plot the predictions
```

```
plot(X_train, X_test, y_train, y_test, pipeline)

plot(X_train, X_test, y_train, y_test, pipeline, log = True)
```

```
Training MSE: 2.3139214731606486    Testing MSE: 1.7200875366333022
```





M6-L2 Problem 1

In this problem you will code a function to perform feature filtering using the Pearson's Correlation Coefficient method.

To start, run the following cell to load the mtcars dataset. Feature names are stored in `feature_names`, while the data is in `data`.

In [51]:
```
import numpy as np

feature_names = ["mpg","cyl","disp","hp","drat","wt","qsec","vs","am","gear","carb"]
data = np.array([[21,6,160,110,3.9,2.62,16.46,0,1,4,4], [21,6,160,110,3.9,2.875,17.02,0,1,4,4], [22.8,4,108,93,
                 [18.1,6,225,105,2.76,3.46,20.22,1,0,3,1], [14.3,8,360,245,3.21,3.57,15.84,0,0,3,4], [24.4,4,146
                 [17.8,6,167.6,123,3.92,3.44,18.9,1,0,4,4],[16.4,8,275.8,180,3.07,4.07,17.4,0,0,3,3],[17.3,8,275
                 [10.4,8,460,215,3,5.424,17.82,0,0,3,4],[14.7,8,440,230,3.23,5.345,17.42,0,0,3,4],[32.4,4,78.7,6
                 [21.5,4,120.1,97,3.7,2.465,20.01,1,0,3,1],[15.5,8,318,150,2.76,3.52,16.87,0,0,3,2],[15.2,8,304,
                 [27.3,4,79,66,4.08,1.935,18.9,1,1,4,1],[26,4,120.3,91,4.43,2.14,16.7,0,1,5,2],[30.4,4,95.1,113,
                 [15,8,301,335,3.54,3.57,14.6,0,1,5,8],[21.4,4,121,109,4.11,2.78,18.6,1,1,4,2]])
```

Filtering

Now define a function `find_redundant_features(data, target_index, threshold)`. Inputs:

- data: input feature matrix
- target_index: index of column in `data` to treat as the target feature
- threshold: eliminate indices with pearson correlation coefficients greater than `threshold`

Return:

- Array of the indices of features to remove.

Procedure:

1. Compute correlation coefficients with `np.corrcoeff(data.T)`, and take the absolute value.
2. Find off-diagonal entries greater than `threshold` which are not in the target_index row/column.
3. For each of these entries above `threshold`, determine which has a lower correlation with the target feature -- add this index to the list of indices to filter out/remove.
4. Remove possible duplicate entries in the list of indices to remove.

In [52]:
```
def find_redundant_features(data, target_index, threshold):
    # YOUR CODE GOES HERE
    ind = []
    corr_coeff = abs(np.corrcoef(data.T))


    for i in range(len(corr_coeff)):
        for j in range(i):
            if corr_coeff[i][j] > threshold and corr_coeff[i][target_index] > corr_coeff[j][target_index] and i
                ind.append(j)
            elif corr_coeff[i][j] > threshold and corr_coeff[i][target_index] < corr_coeff[j][target_index] and
                ind.append(i)


    ind = list(set(ind))
    return ind
```

Testing your function

The following test cases should give the following results:

target_index	threshold	Indices to remove
0	0.9	[2]
2	0.7	[0, 3, 4, 5, 6, 7, 8, 9, 10]
10	0.8	[1, 2, 5]

Try these out in the cell below and print the indices you get.

In [53]:
```
# YOUR CODE GOES HERE
ind1 = find_redundant_features(data, 0, 0.9)
ind2 = find_redundant_features(data, 2, 0.7)
ind3 = find_redundant_features(data, 10, 0.8)
```

```
print("Test case 1 indices: ", ind1)
print("Test case 2 indices: ", ind2)
print("Test case 3 indices: ", ind3)
```

```
Test case 1 indices:  [2]
Test case 2 indices:  [0, 3, 4, 5, 6, 7, 8, 9, 10]
Test case 3 indices:  [1, 2, 5]
```

Using your function

Run these additional cases and print the results:

target_index	threshold	Indices to remove
4	0.9	?
5	0.8	?
6	0.95	?

In [55]:
```
# YOUR CODE GOES HERE
ind1 = find_redundant_features(data, 4, 0.9)
ind2 = find_redundant_features(data, 5, 0.8)
ind3 = find_redundant_features(data, 6, 0.95)

print("Test case 1 indices: ", ind1)
print("Test case 2 indices: ", ind2)
print("Test case 3 indices: ", ind3)
```

```
Test case 1 indices:  [1]
Test case 2 indices:  [0, 1, 3, 7]
Test case 3 indices:  []
```

M6-L2 Problem 2

Now you will implement a wrapper method. This will iteratively determine which features should be most beneficial for predicting the output. Once more, we will use the MTCars dataset predicting `mpg`.

In [1]:
```
import numpy as np
np.set_printoptions(precision=3)
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import itertools

feature_names = ["mpg","cyl","disp","hp","drat","wt","qsec","vs","am","gear","carb"]
data = np.array([[21,6,160,110,3.9,2.62,16.46,0,1,4,4], [21,6,160,110,3.9,2.875,17.02,0,1,4,4], [22.8,4,108,93,
                 [18.1,6,225,105,2.76,3.46,20.22,1,0,3,1], [14.3,8,360,245,3.21,3.57,15.84,0,0,3,4], [24.4,4,146
                 [17.8,6,167.6,123,3.92,3.44,18.9,1,0,4,4],[16.4,8,275.8,180,3.07,4.07,17.4,0,0,3,3],[17.3,8,275
                 [10.4,8,460,215,3,5.424,17.82,0,0,3,4],[14.7,8,440,230,3.23,5.345,17.42,0,0,3,4],[32.4,4,78.7,6
                 [21.5,4,120.1,97,3.7,2.465,20.01,1,0,3,1],[15.5,8,318,150,2.76,3.52,16.87,0,0,3,2],[15.2,8,304,
                 [27.3,4,79,66,4.08,1.935,18.9,1,1,4,1],[26,4,120.3,91,4.43,2.14,16.7,0,1,5,2],[30.4,4,95.1,113,
                 [15,8,301,335,3.54,3.57,14.6,0,1,5,8],[21.4,4,121,109,4.11,2.78,18.6,1,1,4,2]])
target_idx = 0
y = data[:,target_idx]
X = np.delete(data,target_idx,1)
```

Fitting a model

The following function is provided: `get_train_test_mse(X,y,feature_indices)`. This will train a model to fit the data, using only the features specified in `feature_indices`. A train and test MSE are computed and returned.

In [2]:
```
def get_train_test_mse(X, y, feature_indices=None):
    if feature_indices is not None:
        X = X[:,feature_indices]
    X_tr, X_te, y_tr, y_te = train_test_split(X,y,random_state=12,train_size=int(len(y)*.8))
    model = SVR()
    model.fit(X_tr,y_tr)
    mse_train = mean_squared_error(y_tr,model.predict(X_tr))
    mse_test = mean_squared_error(y_te,model.predict(X_te))
    return mse_train, mse_test

mse_train, mse_test = get_train_test_mse(X, y, None)
print(f"Model using all features:    Train MSE={mse_train:.1f},  Test MSE={mse_test:.1f}")
```

```
Model using all features:    Train MSE=16.1,  Test MSE=18.3
```

Wrapper method

Now your job is to write a function `get_next_pair(X, y, current_indices)` that considers all pairs of features to add to the model.
`X` and `y` contain the full input and output arrays. `current_indices` lists the indices currently used by your model and you want to determine the indices of the 2 features that best improve the model (gives the lowest test MSE). Return the indices as an array.

If you want to avoid a double for-loop, `itertools.combinations()` can help generate all pairs of indices from a given array.

In [59]:
```
def get_next_pair(X, y, current_indices):
    # YOUR CODE GOES HERE
    best_mse_test = float('inf')
    best_pair = None

    indices = np.arange(X.shape[1])

    avail_ind = [i for i in indices if i not in current_indices]
    mse_test = []

    for pair in itertools.combinations(avail_ind, 2):
        new_ind = list(current_indices) + list(pair)
        _, mse_test = get_train_test_mse(X, y, new_ind)

        if mse_test < best_mse_test:
            best_mse_test = mse_test
            best_pair = pair

    return best_pair
```

Trying out the wrapper method

Now, let's start with an empty array of indices and add 2 features at a time to the model.
Repeat this until there are 8 features considered. Each pair is printed as it is added.

The first few pairs should be:

- (2, 5)
- (0, 8)

In [62]:
```
indices = np.array([])
while len(indices) < 8:
    pair = get_next_pair(X, y, indices)
    print(f"Adding pair {tuple(map(int, pair))}")
    indices = np.union1d(indices, pair).astype(int)
```

```
Adding pair (2, 5)
Adding pair (0, 8)
Adding pair (6, 7)
Adding pair (4, 9)
```

Question

Which 2 feature indices were deemed "least important" by this wrapper method?

Index 1 and 3 were deemed least important, as they caused the test MSE values to increases and hence were not added to the list of indices

Problem 1

During the lecture you worked with pipelines in SciKit-Learn to perform feature transformation before classification/regression using a pipeline. In this problem, you will look at another scaling method in a 2D regression context.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

Sklearn Models (no scaling): Print Train and Test MSE

- Linear Regression (input degree 8 features)
- SVR, C = 1000
- KNN, K = 4
- Random Forest, 100 estimators of max depth 10

Sklearn Pipeline (scaling + model): Print Train and Test MSE

- Linear Regression (input degree 8 features)
- SVR, C = 1000
- KNN, K = 4
- Random Forest, 100 estimators of max depth 10

Plots

- 1x5 subplot showing model predictions on unscaled features, next to ground truth
- 1x5 subplot showing pipeline predictions with features scaled, next to ground truth

Questions

- Respond to the prompts at the end

In [30]:
```
import numpy as np
import matplotlib.pyplot as plt


from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import PolynomialFeatures, QuantileTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot(X, y, title=""):
    plt.scatter(X[:,0],X[:,1],c=y,cmap="jet")
    plt.colorbar(orientation="horizontal")
    plt.xlabel("$x_1$")
    plt.ylabel("$x_2$")
    plt.title(title)
```

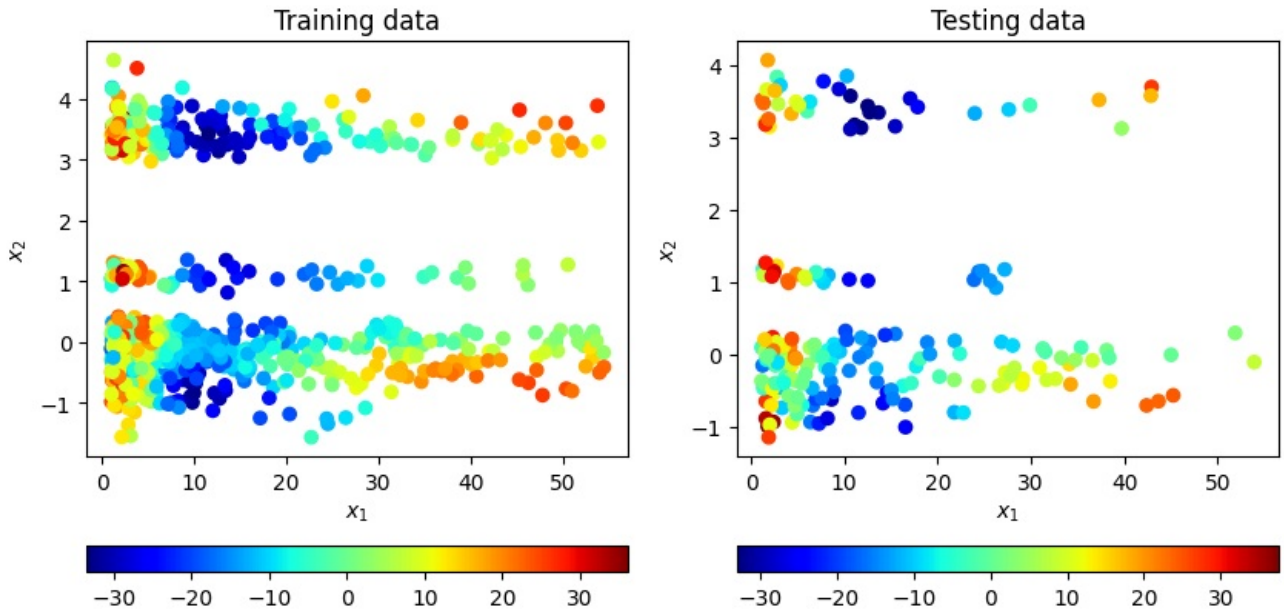
Load the data

Complete the loading process below by inputting the path to the data file "w6-p1-data.npy"

Training data is in `X_train` and `y_train`. Testing data is in `X_test` and `y_test`.

In [31]:
```
# YOUR CODE GOES HERE
# Define path
data = np.load("data\w6-p1-data.npy")
X, y = data[:,:2], data[:,2]
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=int(0.8*len(y)),random_state=0)

plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plot(X_train, y_train, "Training data")
plt.subplot(1,2,2)
plot(X_test, y_test, "Testing data")
plt.show()
```

```
<>:3: SyntaxWarning: invalid escape sequence '\w'
<>:3: SyntaxWarning: invalid escape sequence '\w'
C:\Users\barat\AppData\Local\Temp\ipykernel_2244\3237680281.py:3: SyntaxWarning: invalid escape sequence '\w'
  data = np.load("data\w6-p1-data.npy")
```



Models (no input scaling)

Fit 4 models to the training data:

- `LinearRegression()`. This should be a pipeline whose first step is `PolynomialFeatures()` with degree 7.
- `SVR()` with C = 1000 and "rbf" kernel
- `KNeighborsRegressor()` using 4 nearest neighbors
- `RandomForestRegressor()` with 100 estimators of max depth 10

Print the Train and Test MSE for each

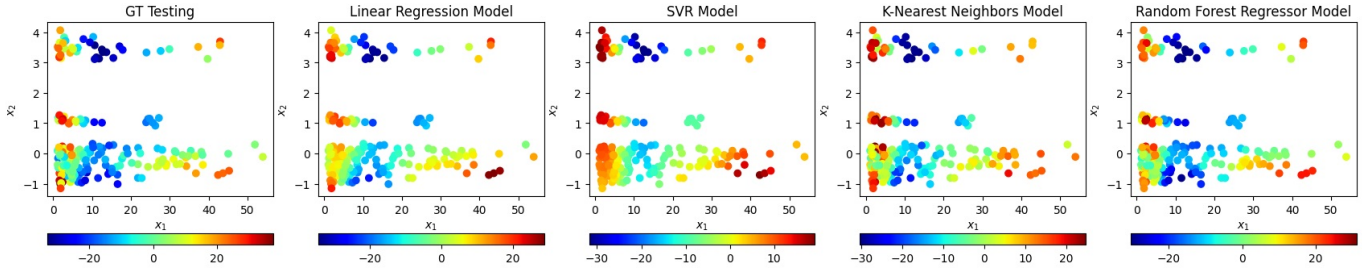
In [32]:
```
model_names = ["LSR", "SVR", "KNN", "RF"]

# YOUR CODE GOES HERE
LR_ = Pipeline([("poly", PolynomialFeatures(degree = 7)), ("LSR", LinearRegression())])
SVR_ = Pipeline([("SVR", SVR(C = 1000, kernel = 'rbf'))])
KNN_ = Pipeline([("KNN", KNeighborsRegressor(n_neighbors = 4))])
RF_ = Pipeline([("RF", RandomForestRegressor(n_estimators = 100, max_depth = 10))])

LR_.fit(X_train, y_train)
SVR_.fit(X_train, y_train)
KNN_.fit(X_train, y_train)
RF_.fit(X_train, y_train)

mse_train_LR_ = mean_squared_error(y_train, LR_.predict(X_train))
mse_train_SVR_ = mean_squared_error(y_train, SVR_.predict(X_train))
mse_train_KNN_ = mean_squared_error(y_train, KNN_.predict(X_train))
mse_train_RF_ = mean_squared_error(y_train, RF_.predict(X_train))

mse_test_LR_ = mean_squared_error(y_test, LR_.predict(X_test))
mse_test_SVR_ = mean_squared_error(y_test, SVR_.predict(X_test))
mse_test_KNN_ = mean_squared_error(y_test, KNN_.predict(X_test))
mse_test_RF_ = mean_squared_error(y_test, RF_.predict(X_test))

print("Train MSE for Linear Regression Model: ", mse_train_LR_)
print("Test MSE for Linear Regression Model: ", mse_test_LR_)

print("\nTrain MSE for SVR Model: ", mse_train_SVR_)
print("Test MSE for SVR Model: ", mse_test_SVR_)

print("\nTrain MSE for KNN Model: ", mse_train_KNN_)
print("Test MSE for KNN Model: ", mse_test_KNN_)

print("\nTrain MSE for RF Model: ", mse_train_RF_)
print("Test MSE for RF Model: ", mse_test_RF_)
```

```
Train MSE for Linear Regression Model:  50.866389902820316
Test MSE for Linear Regression Model:  57.28650739095314

Train MSE for SVR Model:  82.04352603565992
Test MSE for SVR Model:  98.63319719407623

Train MSE for KNN Model:  26.856498566141628
Test MSE for KNN Model:  47.63617328402055

Train MSE for RF Model:  5.874835115214487
Test MSE for RF Model:  25.26604134921764
```

Visualizing the predictions

Plot the predictions of each method on the testing data in a 1x5 subplot structure, with the ground truth values as the leftmost subplot.

In [33]:
```
plt.figure(figsize=(21,4))
plt.subplot(1,5,1)
plot(X_test, y_test, "GT Testing")

# YOUR CODE GOES HERE
plt.subplot(1,5,2)
plot(X_test, LR_.predict(X_test), "Linear Regression Model")

plt.subplot(1,5,3)
plot(X_test, SVR_.predict(X_test), "SVR Model")

plt.subplot(1,5,4)
plot(X_test, KNN_.predict(X_test), "K-Nearest Neighbors Model")

plt.subplot(1,5,5)
plot(X_test, RF_.predict(X_test), "Random Forest Regressor Model")

plt.show()
```



Quantile Scaling

A `QuantileTransformer()` can transform the input data in a way that attempts to match a given distribution (uniform distribution by default).

- Create a quantile scaler with `n_quantiles = 800`.
- Then, create a pipeline for each of the 4 types of models used earlier
- Fit each pipeline to the training data, and again print the train and test MSE

In [34]:
```
pipeline_names = ["LSR, scaled", "SVR, scaled", "KNN, scaled", "RF, scaled"]

# YOUR CODE GOES HERE
QF = QuantileTransformer(n_quantiles = 800)

LR_scaled_ = Pipeline([('scaler', QF), ("poly", PolynomialFeatures(degree = 7)), ("LSR, scaled", LinearRegressi
SVR_scaled_ = Pipeline([('scaler', QF), ("SVR, scaled", SVR(C = 1000, kernel = 'rbf'))])
KNN_scaled_ = Pipeline([('scaler', QF), ("KNN, scaled", KNeighborsRegressor(n_neighbors = 4))])
RF_scaled_ = Pipeline([('scaler', QF), ("RF, scaled", RandomForestRegressor(n_estimators = 100, max_depth = 10)

LR_scaled_.fit(X_train, y_train)
SVR_scaled_.fit(X_train, y_train)
KNN_scaled_.fit(X_train, y_train)
RF_scaled_.fit(X_train, y_train)

mse_train_LR_scaled_ = mean_squared_error(y_train, LR_scaled_.predict(X_train))
mse_train_SVR_scaled_ = mean_squared_error(y_train, SVR_scaled_.predict(X_train))
mse_train_KNN_scaled_ = mean_squared_error(y_train, KNN_scaled_.predict(X_train))
mse_train_RF_scaled_ = mean_squared_error(y_train, RF_scaled_.predict(X_train))

mse_test_LR_scaled_ = mean_squared_error(y_test, LR_scaled_.predict(X_test))
mse_test_SVR_scaled_ = mean_squared_error(y_test, SVR_scaled_.predict(X_test))
mse_test_KNN_scaled_ = mean_squared_error(y_test, KNN_scaled_.predict(X_test))
mse_test_RF_scaled_ = mean_squared_error(y_test, RF_scaled_.predict(X_test))
```

```
print("Train MSE for scaled Linear Regression Model: ", mse_train_LR_scaled_)
print("Test MSE for scaled Linear Regression Model: ", mse_test_LR_scaled_)

print("\nTrain MSE for scaled SVR Model: ", mse_train_SVR_scaled_)
print("Test MSE for scaled SVR Model: ", mse_test_SVR_scaled_)

print("\nTrain MSE for scaled KNN Model: ", mse_train_KNN_scaled_)
print("Test MSE for scaled KNN Model: ", mse_test_KNN_scaled_)

print("\nTrain MSE for scaled RF Model: ", mse_train_RF_scaled_)
print("Test MSE for scaled RF Model: ", mse_test_RF_scaled_)
```

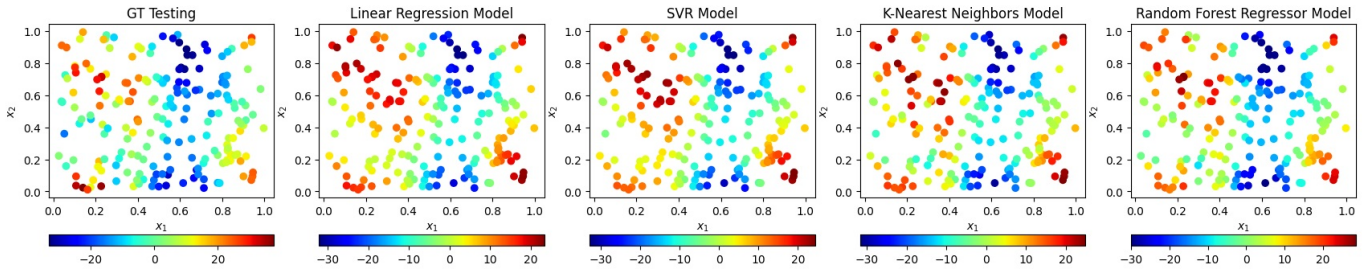
```
Train MSE for scaled Linear Regression Model:  39.52893428670224
Test MSE for scaled Linear Regression Model:  43.20363492250541

Train MSE for scaled SVR Model:  41.03425800596035
Test MSE for scaled SVR Model:  43.017915737899095

Train MSE for scaled KNN Model:  19.687691313922564
Test MSE for scaled KNN Model:  36.397038931930005

Train MSE for scaled RF Model:  6.105820991617033
Test MSE for scaled RF Model:  25.546583759880477
```

Visualization with scaled input

As before, plot the predictions of each *scaled* method on the testing data in a 1x5 subplot structure, with the ground truth values as the leftmost subplot.

This time, for each plot, show the scaled data points instead of the original data. You can do this by calling `.transform()` on your quantile scaler. The scaled points should appear to follow a uniform distribution.

In [35]:
```
# YOUR CODE GOES HERE
plt.figure(figsize=(21,4))
plt.subplot(1,5,1)
plot(QF.transform(X_test), y_test, "GT Testing")

# YOUR CODE GOES HERE
plt.subplot(1,5,2)
plot(QF.transform(X_test), LR_scaled_.predict(X_test), "Linear Regression Model")

plt.subplot(1,5,3)
plot(QF.transform(X_test), SVR_scaled_.predict(X_test), "SVR Model")

plt.subplot(1,5,4)
plot(QF.transform(X_test), KNN_scaled_.predict(X_test), "K-Nearest Neighbors Model")

plt.subplot(1,5,5)
plot(QF.transform(X_test), RF_scaled_.predict(X_test), "Random Forest Regressor Model")

plt.show()
```



Questions

1. Without transforming the input data, which model performed the best on test data? What about after scaling?
2. For each method, say whether scaling the input improved or worsened, how extreme the change was, and why you think this is.

Answers

1. Before scaling the data, the Random Forest Regressor model performed the best on the test data, with a test MSE of 25.3. After scaling as well, the Random Forest Regressor model performed the best on the test data.
2. The test MSE decreased significantly after scaling for the LR, SVR and KNN models. The decrease was around 20% for the LR and KNN models, but for the SVR model, the test MSE decreased to less than half of the original value. For the RF model, the test MSE

increases very slightly.This happens because LR, SVR and KNN depends on distance based metrics for optimization, whereas RF depends on thresholds for feature values. Due to this, RF models are usually insensitive to feature scaling

Problem 2

Data-driven field prediction models can be used as a substitute for performing expensive calculations/simulations in design loops. For example, after being trained on finite element solutions for many parts, they can be used to predict nodal von Mises stress for a new part by taking in a mesh representation of the part geometry.

Consider the plane-strain compression problem shown in "data/plane-strain.png".

In this problem you are given node features for 100 parts. These node features have been extracted by processing each part shape using a neural network. You will perform feature selection to determine which of these features are most relevant using feature selection tools in sklearn.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

SciKit-Learn Models: Print Train and Test MSE

- `LinearRegression()` with all features
- `DecisionTreeRegressor()` with all features
- `LinearRegression()` with features selected by `RFE()`
- `DecisionTreeRegressor()` with features selected by `RFE()`

Feature Importance/Coefficient Visualizations

- Feature importance plot for Decision Tree using all features
- Feature coefficient plot for Linear Regression using all features
- Feature importance plot for DT showing which features RFE selected
- Feature coefficient plot for LR showing which features RFE selected

Stress Field Visualizations: Ground Truth vs. Prediction

- Test dataset shape index 8 for decision tree and linear regression with all features
- Test dataset shape index 16 for decision tree and linear regression with RFE features

Questions

- Respond to the 5 prompts at the end

Imports and Utility Functions:

In [15]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.feature_selection import RFE

def plot_shape(dataset, index, model=None, lims=None):
    x = dataset["coordinates"][index][:,0]
    y = dataset["coordinates"][index][:,1]

    if model is None:
        c = dataset["stress"][index]
    else:
        c = model.predict(dataset["features"][index])

    if lims is None:
        lims = [min(c),max(c)]

    plt.scatter(x,y,s=5,c=c,cmap="jet",vmin=lims[0],vmax=lims[1])
    plt.colorbar(orientation="horizontal", shrink=.75, pad=0,ticks=lims)
    plt.axis("off")
    plt.axis("equal")

def plot_shape_comparison(dataset, index, model, title=""):
    plt.figure(figsize=[6,3.2], dpi=120)
    plt.subplot(1,2,1)
    plot_shape(dataset,index)
    plt.title("Ground Truth",fontsize=9,y=.96)
    plt.subplot(1,2,2)
    c = dataset["stress"][index]
    plot_shape(dataset, index, model, lims = [min(c), max(c)])
    plt.title("Prediction",fontsize=9,y=.96)
```

```
    plt.suptitle(title)
    plt.show()

def load_dataset(path):
    dataset = np.load(path)
    coordinates = []
    features = []
    stress = []
    N = np.max(dataset[:,0].astype(int)) + 1
    split = int(N*.8)
    for i in range(N):
        idx = dataset[:,0].astype(int) == i
        data = dataset[idx,:]
        coordinates.append(data[:,1:3])
        features.append(data[:,3:-1])
        stress.append(data[:,-1])
    dataset_train = dict(coordinates=coordinates[:split], features=features[:split], stress=stress[:split])
    dataset_test = dict(coordinates=coordinates[split:], features=features[split:], stress=stress[split:])
    X_train, X_test = np.concatenate(features[:split], axis=0), np.concatenate(features[split:], axis=0)
    y_train, y_test = np.concatenate(stress[:split], axis=0), np.concatenate(stress[split:], axis=0)
    return dataset_train, dataset_test, X_train, X_test, y_train, y_test

def get_shape(dataset,index):
    X = dataset["features"][index]
    y = dataset["stress"][index]
    return X, y

def plot_importances(model, selected = None, coef=False, title=""):
    plt.figure(figsize=(6,2),dpi=150)
    y = model.coef_ if coef else model.feature_importances_
    N = 1+len(y)
    x = np.arange(1,N)

    plt.bar(x,y)

    if selected is not None:
        plt.bar(x[selected],y[selected],color="red",label="Selected Features")
        plt.legend()

    plt.xlabel("Feature")

    plt.ylabel("Coefficient" if coef else "Importance")
    plt.xlim(0,N)
    plt.title(title)
    plt.show()
```

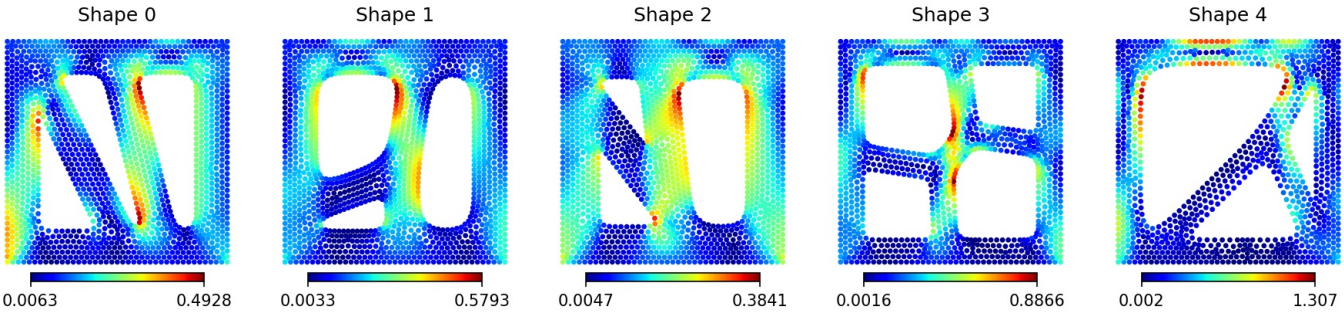
Loading the data

First, complete the code below to load the data and plot the von Mises stress fields for a few shapes.
You'll need to input the path of the data file, the rest is done for you.

All training node features and outputs are in `X_train` and `y_train`, respectively. Testing nodes are in `X_test`, `y_test`.

`dataset_train` and `dataset_test` contain more detailed information such as node coordinates, and they are separated by shape.
Get features and outputs for a shape by calling `get_shape(dataset,index)`. `N_train` and `N_test` are the number of training and testing shapes in each of these datasets.

In [16]:
```
# YOUR CODE GOES HERE
# Define data_path
data_path = "data/stress_nodal_features.npy"

dataset_train, dataset_test, X_train, X_test, y_train, y_test = load_dataset(data_path)
N_train = len(dataset_train["stress"])
N_test = len(dataset_test["stress"])

plt.figure(figsize=[15,3.2], dpi=150)
for i in range(5):
    plt.subplot(1,5,i+1)
    plot_shape(dataset_train,i)
    plt.title(f"Shape {i}")
plt.show()
```



Fitting models with all features

Create two models to fit the training data `X_train`, `y_train`:

1. A `LinearRegression()` model
2. A `DecisionTreeRegressor()` model with a `max_depth` of 20

Print the training and testing MSE for each.

In [42]:
```
# YOUR CODE GOES HERE
LR_ = LinearRegression()
DTR_ = DecisionTreeRegressor(max_depth = 20)

LR_.fit(X_train, y_train)
DTR_.fit(X_train, y_train)

mse_train_LR_ = mean_squared_error(y_train, LR_.predict(X_train))
mse_train_DTR_ = mean_squared_error(y_train, DTR_.predict(X_train))

mse_test_LR_ = mean_squared_error(y_test, LR_.predict(X_test))
mse_test_DTR_ = mean_squared_error(y_test, DTR_.predict(X_test))

print("Train MSE for Linear Regression Model: ", mse_train_LR_)
print("Test MSE for Linear Regression Model: ", mse_test_LR_)

print("\nTrain MSE for Decision Tree Regressor Model: ", mse_train_DTR_)
print("Test MSE for Decision Tree Regressor Model: ", mse_test_DTR_)
```

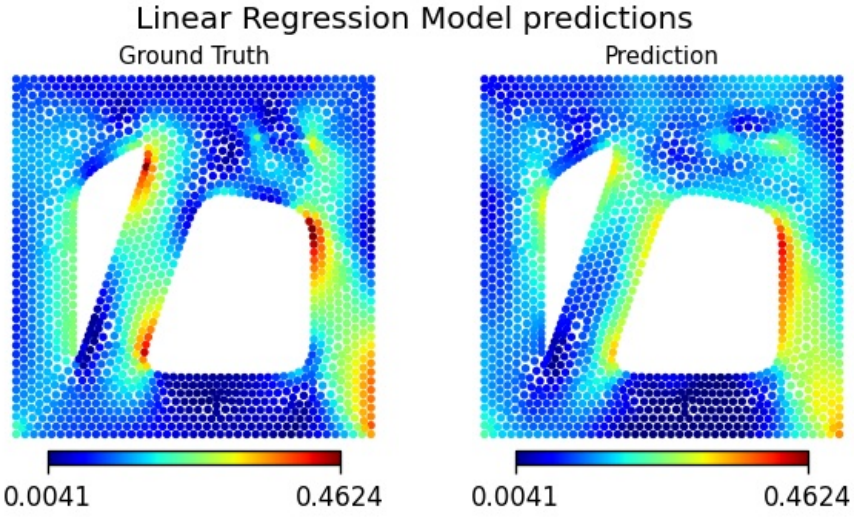
```
Train MSE for Linear Regression Model:  0.008110601
Test MSE for Linear Regression Model:  0.009779483

Train MSE for Decision Tree Regressor Model:  0.0004944875978805109
Test MSE for Decision Tree Regressor Model:  0.008307369037199367
```

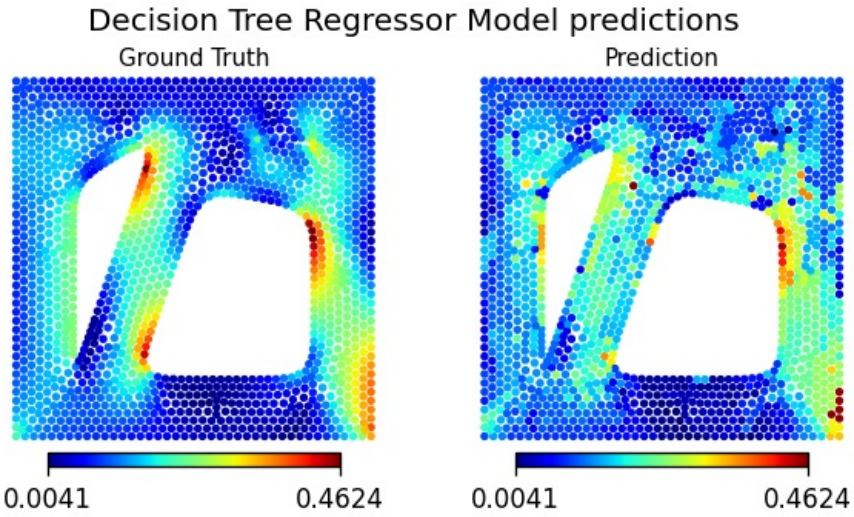
Visualization

Use the `plot_shape_comparison()` function to plot the index 8 shape results in `dataset_test` for each model.

Include titles to indicate which plot is which, using the `title` argument.

In [43]:
```
test_idx = 8

# YOUR CODE GOES HERE
plot_shape_comparison(dataset_test, test_idx, LR_, title = "Linear Regression Model predictions")
plot_shape_comparison(dataset_test, test_idx, DTR_, title = "Decision Tree Regressor Model predictions")
```



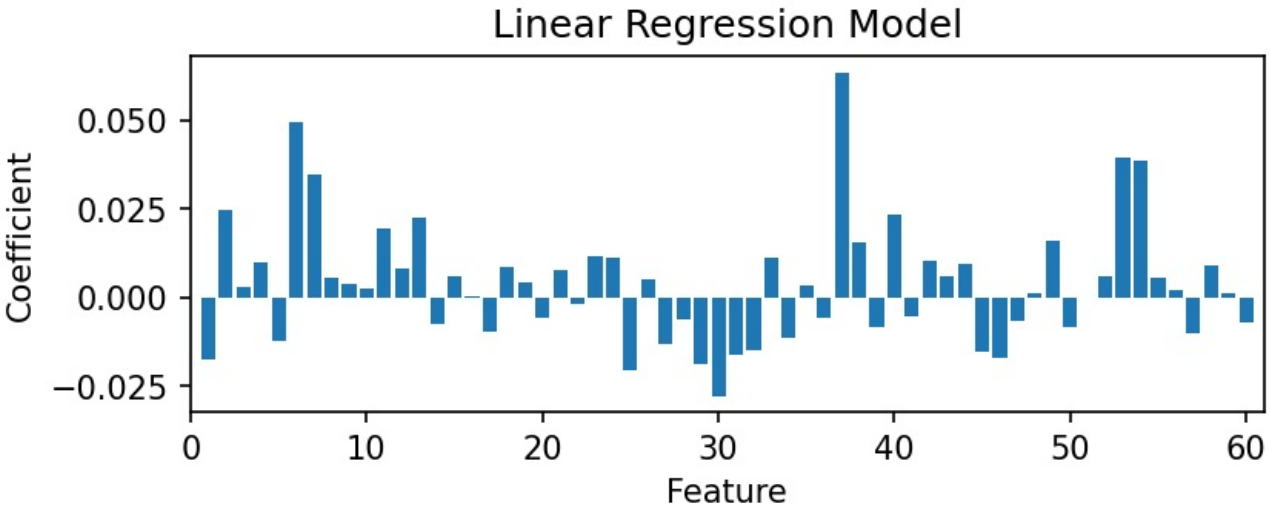


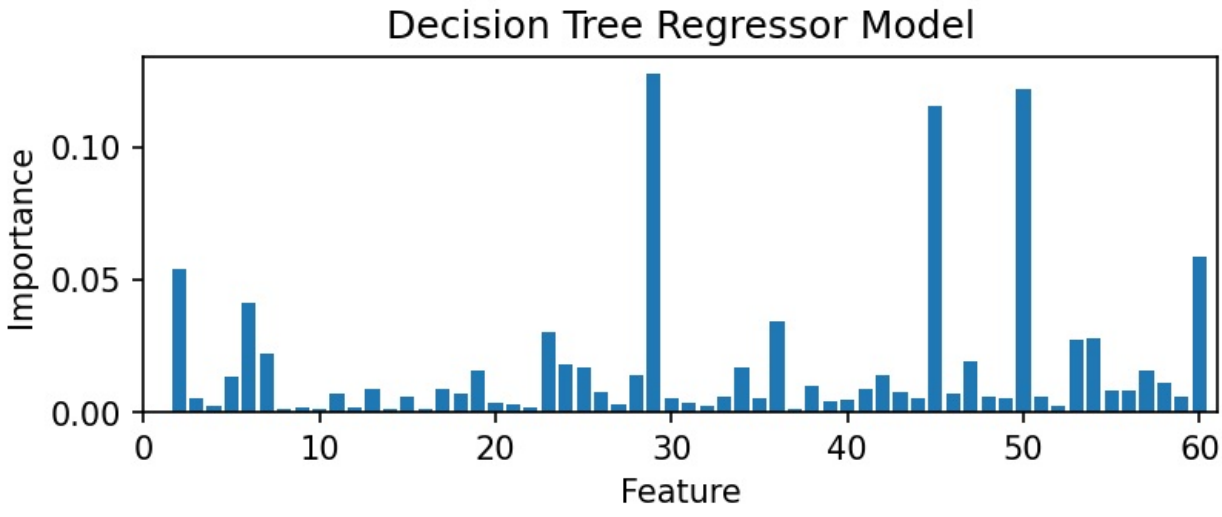
Feature importance

For a tree methods, "feature importance" can be computed, which can be done for an sklearn model using `.feature_importances_`.

Use the provided function `plot_importances()` to visualize which features are most important to the final decision tree prediction. Then create another plot using the same function to visualize the linear regression coefficients by setting the "coef" argument to `True`.

In [44]:
```
# YOUR CODE GOES HERE
plot_importances(LR_, coef = True, title = "Linear Regression Model")
plot_importances(DTR_, title = "Decision Tree Regressor Model")
```





Feature Selection by Recursive Feature Elimination

Using `RFE()` in sklearn, you can iteratively select a subset of only the most important features.

For both linear regression and decision tree (depth 20) models:

1. Create a new model.
2. Create an instance of `RFE()` with `n_features_to_select` set to 30.
3. Fit the RFE model as you would a normal sklearn model.
4. Report the train and test MSE.

Note that the decision tree RFE model may take a few minutes to train.

Visit https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html for more information.

In [45]:
```
# YOUR CODE GOES HERE
RFE_LR_ = RFE(LR_, n_features_to_select = 30)
RFE_DTR_ = RFE(DTR_, n_features_to_select = 30)

RFE_LR_.fit(X_train, y_train)
RFE_DTR_.fit(X_train, y_train)

mse_train_rfe_lr = mean_squared_error(y_train, RFE_LR_.predict(X_train))
mse_train_rfe_dtr = mean_squared_error(y_train, RFE_DTR_.predict(X_train))

mse_test_rfe_lr = mean_squared_error(y_test, RFE_LR_.predict(X_test))
mse_test_rfe_dtr = mean_squared_error(y_test, RFE_DTR_.predict(X_test))

print("Train MSE after RFE on Linear Regression Model: ", mse_train_rfe_lr)
print("Test MSE after RFE on Linear Regression Model: ", mse_test_rfe_lr)

print("\nTrain MSE after RFE on Decision Tree Regressor Model: ", mse_train_rfe_dtr)
print("Test MSE after RFE on Decision Tree Regressor Model: ", mse_test_rfe_dtr)
```

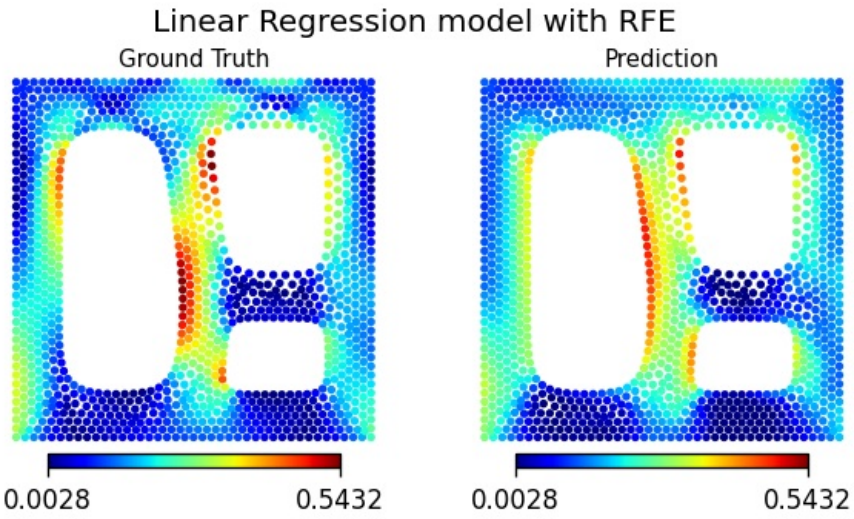
```
Train MSE after RFE on Linear Regression Model:  0.008508719
Test MSE after RFE on Linear Regression Model:  0.010150376

Train MSE after RFE on Decision Tree Regressor Model:  0.0005512811770783042
Test MSE after RFE on Decision Tree Regressor Model:  0.009104145243788579
```

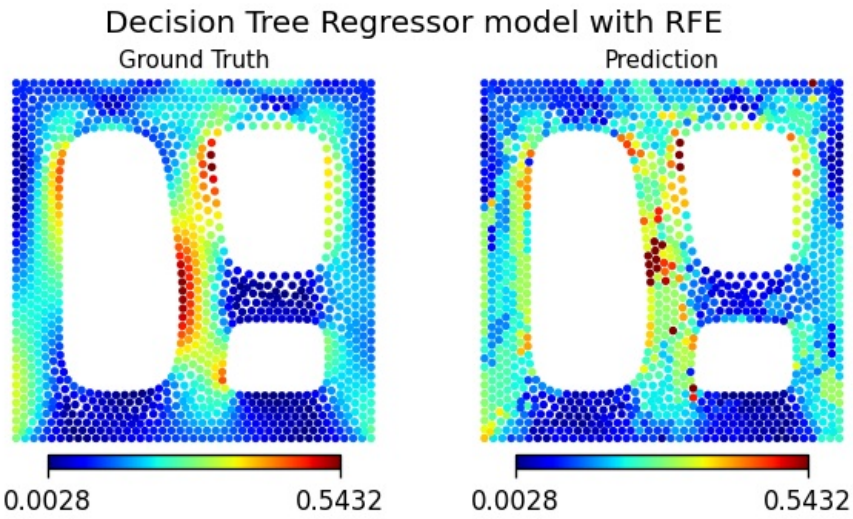
Visualization

Use the `plot_shape_comparison()` function to plot the index 16 shape results in `dataset_test` for each model.

As before, include titles to indicate which plot is which, using the `title` argument.

In [47]:
```
test_idx = 16

# YOUR CODE GOES HERE
plot_shape_comparison(dataset_test, test_idx, RFE_LR_, title = "Linear Regression model with RFE")
plot_shape_comparison(dataset_test, test_idx, RFE_DTR_, title = "Decision Tree Regressor model with RFE")
```



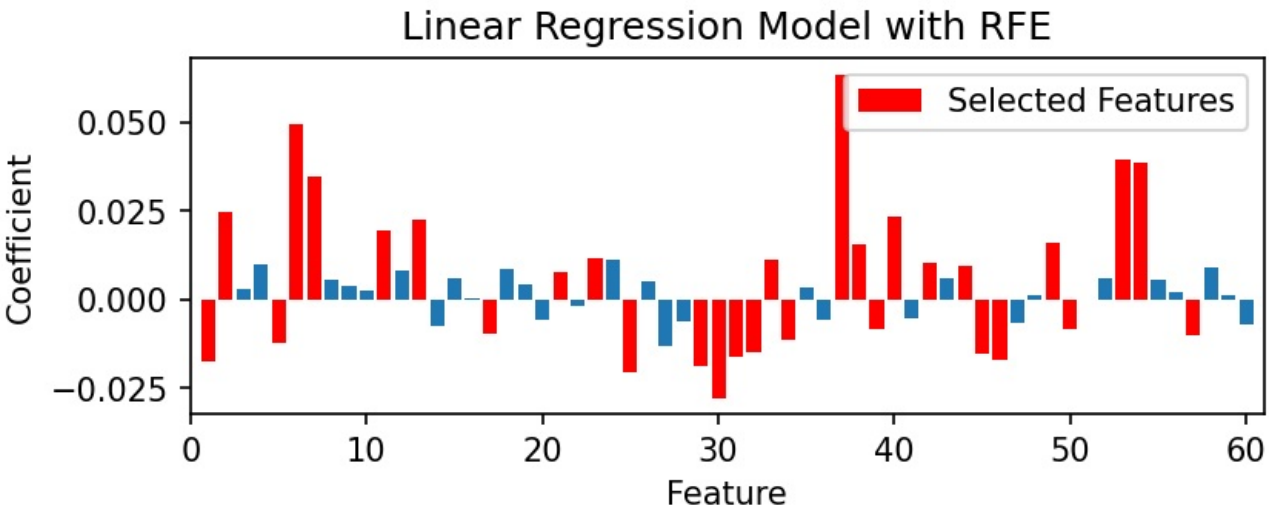


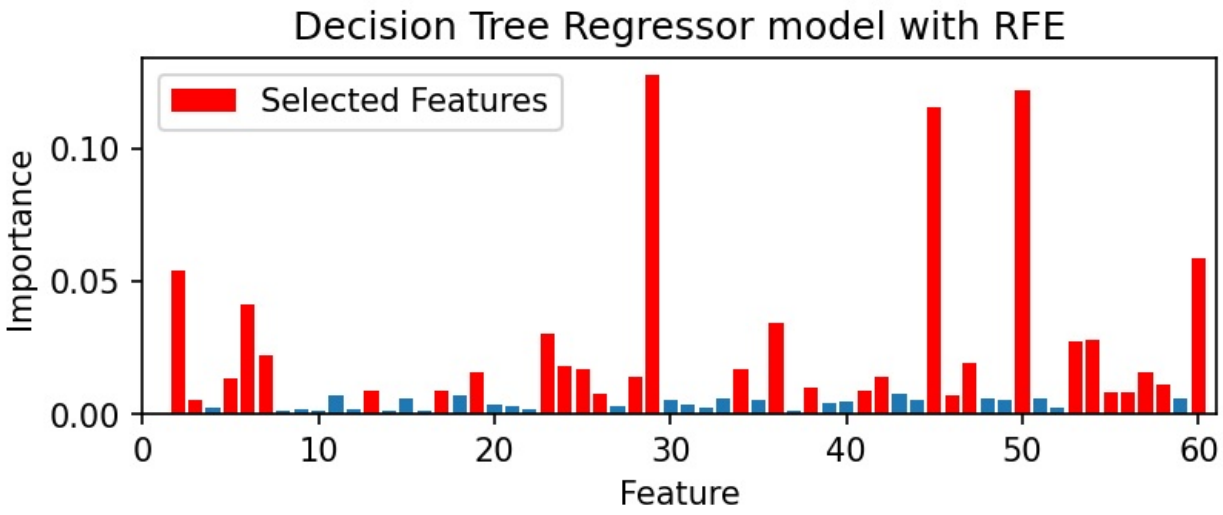
Feature importance with RFE

Recreate the 2 feature importance/coefficent plots from earlier, but this time highlight which features were ultimately selected after performing RFE by coloring those features red. You can do this by setting the `selected` argument equal to an array of selected indices.

For an RFE model `rfe`, the selected feature indices can be obtained via `rfe.get_support(indices=True)`.

In [48]:
```
# YOUR CODE GOES HERE
plot_importances(model = LR_, selected = np.array(RFE_LR_.get_support(indices = True)), coef = True, title = "L
plot_importances(model = DTR_, selected = RFE_DTR_.get_support(indices = True), title = "Decision Tree Regresso
```





Questions

1. Did the MSE increase or decrease on test data for the Linear Regression model after performing RFE?
2. Did the MSE increase or decrease on test data for the Decision Tree model after performing RFE?
3. Describe the qualitative differences between the Linear Regression and the Decision Tree predictions.
4. Describe how the importance of features that were selected by RFE compare to that of features that were eliminated (for the decision tree).
5. Describe how the coefficients that were selected by RFE compare to that of features that were eliminated (for linear regression).

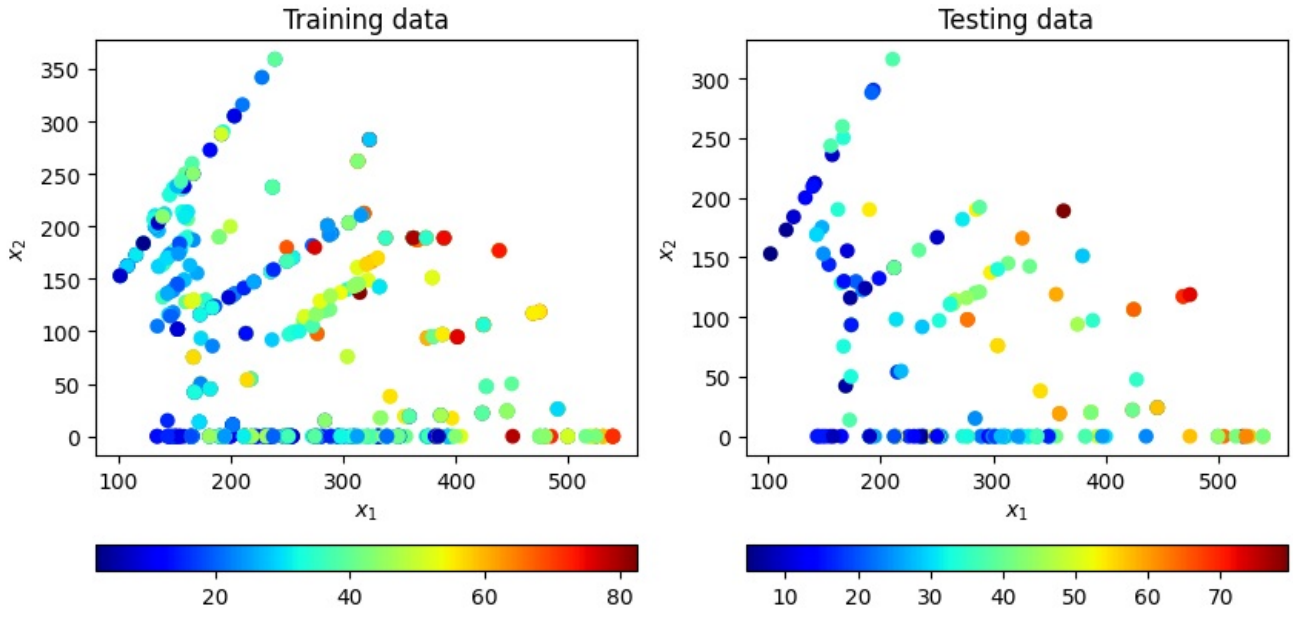
Answers

1. For the Linear Regression model, the MSE on the test data increased after performing RFE
2. For the Decision Tree Regressor Model, the MSE on the test data increased after performing RFE
3. Linear Regression assumes a linear relationship between the features and output of the model, whereas decision tree are able to capture more complex information, and do not assume a specific form for the relationship between the features and the outputs
4. On plotting the feature importances, it is clear that RFE selected the 30 features with the higher importances magnitudes and neglected the others. Since all features have positive importance values, the ones with the lower magnitudes were neglected.
5. In this case, since there are both positive and negative values of feature importances, it seems like the features with the most importance in an absolute sense (magnitude) were selected by RFE.

In [47]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import pandas as pd
from sklearn.feature_selection import RFE
from sklearn.preprocessing import QuantileTransformer, FunctionTransformer
from sklearn.model_selection import train_test_split

data = pd.read_excel('bonus/m06_bonus.xls')
data_np = data.to_numpy()

X, y = data_np[:,:10], data_np[:,-1]
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=int(0.8*len(y)),random_state=0)

def plot_importances(model, selected = None, coef=False, title=""):
    plt.figure(figsize=(6,2),dpi=150)
    y = model.coef_ if coef else model.feature_importances_
    N = 1+len(y)
    x = np.arange(1,N)

    plt.bar(x,y)

    if selected is not None:
        plt.bar(x[selected],y[selected],color="red",label="Selected Features")
        plt.legend()

    plt.xlabel("Feature")

    plt.ylabel("Coefficient" if coef else "Importance")
    plt.xlim(0,N)
    plt.title(title)
    plt.show()
```

Plotting the raw training and test data

In [48]:
```
def plot(X, y, title=""):
    plt.scatter(X[:,0],X[:,1],c=y,cmap="jet")
    plt.colorbar(orientation="horizontal")
    plt.xlabel("$x_1$")
    plt.ylabel("$x_2$")
    plt.title(title)

plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plot(X_train, y_train, "Training data")
plt.subplot(1,2,2)
plot(X_test, y_test, "Testing data")
plt.show()
```



Constructing the pipeline with data preprocessing and model training, and a simple linear regression model

In [49]:
```
def log_transform(x):
    return np.log(x + 1.)
log_transform = FunctionTransformer(log_transform)
QF = QuantileTransformer(n_quantiles = X_train.shape[0])


pipeline = Pipeline([("log", log_transform), ("transform", QF), ("Linear Regression", LinearRegression())])
pipeline.fit(X_train, y_train)

LR_ = LinearRegression()
LR_.fit(X_train, y_train)
```

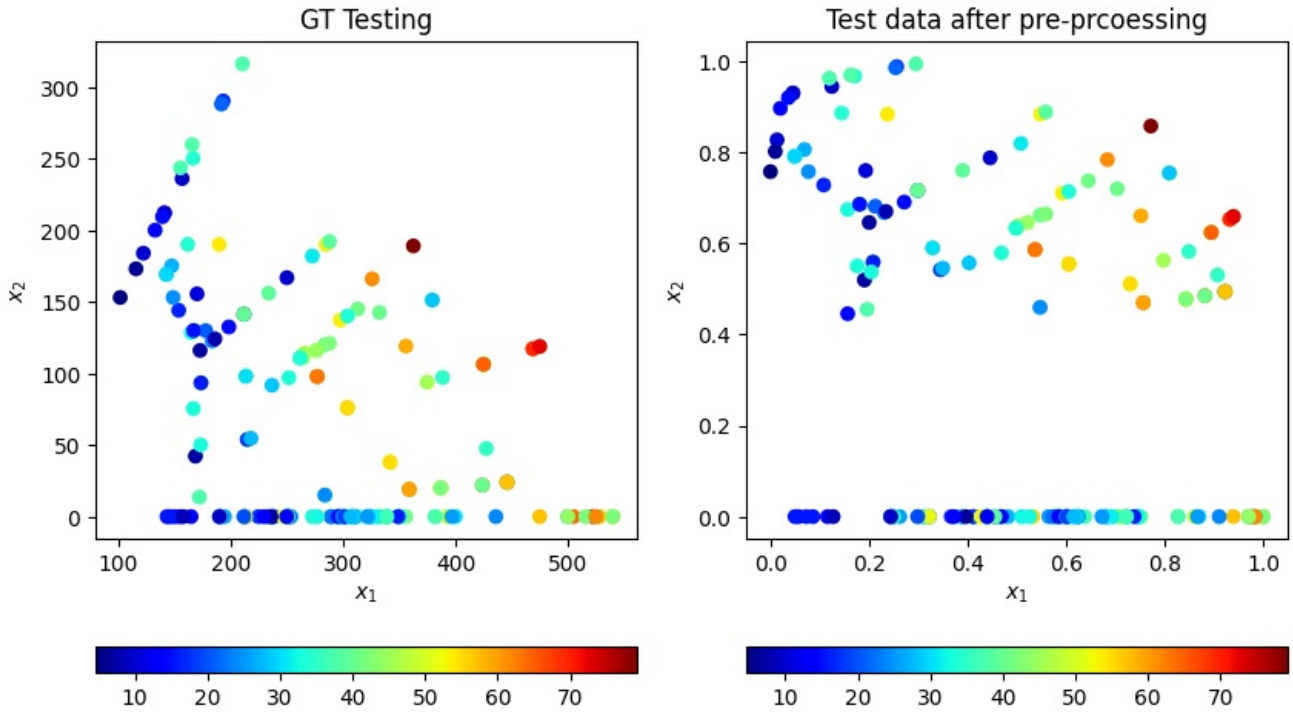
Out[49]:

LinearRegression

LinearRegression()

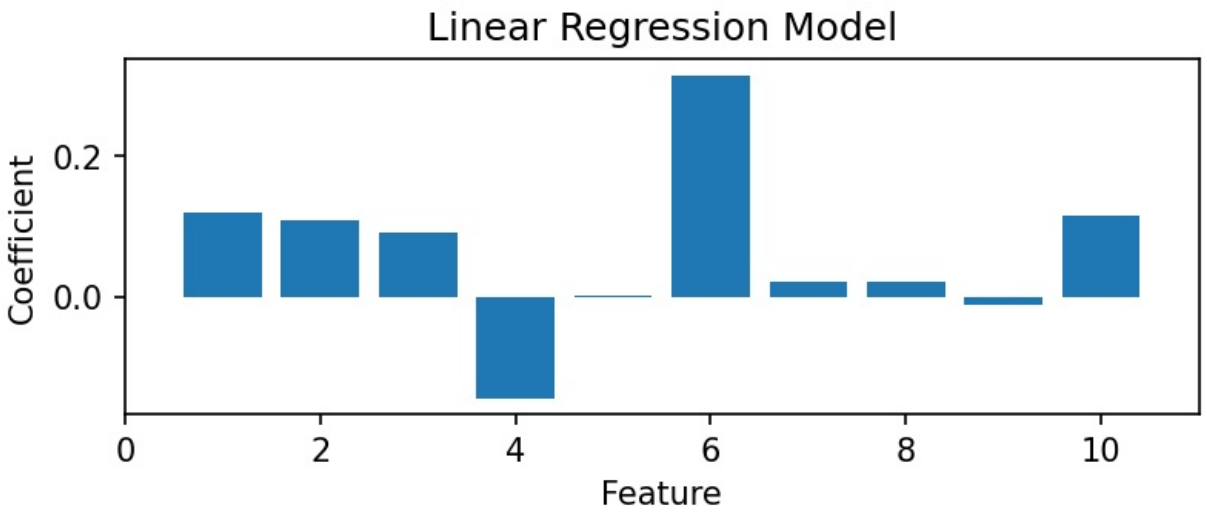
Plotting the raw and processed test data to look for more opportunities of processing

In [50]:
```
X_test_log_transformed = pipeline.named_steps['log'].transform(X_test)
X_test_processed = pipeline.named_steps['transform'].transform(X_test_log_transformed)

plt.figure(figsize=(10,6))
plt.subplot(1,2,1)
plot(X_test, y_test, "GT Testing")

plt.subplot(1,2,2)
plot(X_test_processed, y_test, "Test data after pre-prcoessing")
```



Investigating feature importances

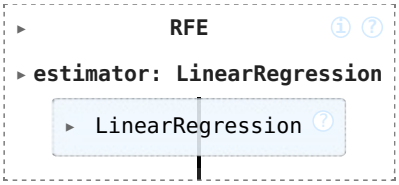
In [51]:
```
plot_importances(LR_, coef = True, title = "Linear Regression Model")
```



Investigating the effects of removing less important features

In [52]:
```
X_train_transformed = pipeline.named_steps['log'].transform(X_train)
X_train_transformed = pipeline.named_steps['transform'].transform(X_train_transformed)

RFE_pipeline_ = RFE(pipeline.named_steps['Linear Regression'], n_features_to_select = 8)
RFE_pipeline_.fit(X_train_transformed, y_train)
```

Out[52]:



Printing train and test data accuracy for simple linear regression, linear regression model with feature engineering, linear regression model with feature engineering and RFE

In [53]:
```
mse_train_LR_ = mean_squared_error(y_train, LR_.predict(X_train))
mse_test_LR_ = mean_squared_error(y_test, LR_.predict(X_test))

print("Train MSE for Linear Regression Model: ", mse_train_LR_)
print("Test MSE for Linear Regression Model: ", mse_test_LR_)
```

```
Train MSE for Linear Regression Model:  110.2933901504166
Test MSE for Linear Regression Model:  95.9503398969457
```

In [54]:
```
mse_train_LR_scaled_ = mean_squared_error(y_train, pipeline.predict(X_train))
mse_test_LR_scaled_ = mean_squared_error(y_test, pipeline.predict(X_test))

print("Train MSE for Linear Regression Model with scaled features: ", mse_train_LR_scaled_)
print("Test MSE for Linear Regression Model with scaled features: ", mse_test_LR_scaled_)
```

```
Train MSE for Linear Regression Model with scaled features:  53.452115554336736
Test MSE for Linear Regression Model with scaled features:  48.21842352448586
```

In [55]:
```
X_test_transformed = pipeline.named_steps['log'].transform(X_test)
X_test_transformed = pipeline.named_steps['transform'].transform(X_test_transformed)

mse_train_LR_scaled_rfe_ = mean_squared_error(y_train, RFE_pipeline_.predict(X_train_transformed))
mse_test_LR_scaled_rfe_ = mean_squared_error(y_test, RFE_pipeline_.predict(X_test_transformed))

print("Train MSE for Linear Regression Model with scaled features and RFE: ", mse_train_LR_scaled_rfe_)
print("Test MSE for Linear Regression Model with scaled features and RFE: ", mse_test_LR_scaled_rfe_)
```

```
Train MSE for Linear Regression Model with scaled features and RFE:  53.508276802406435
Test MSE for Linear Regression Model with scaled features and RFE:  48.348691842336954
```

Conclusions

It can be seen that the test MSE after transforming and scaling the features is much lesser than before. However, removing less important features does not seem to have too much effect on the MSE values