SRECHARAN SELVAM
SSELVAM

Homework 6

Instructions
This homework contains **4** concepts and **7** programming questions. In MS word or a similar text editor, write down the problem number and your answer for each problem. Combine all answers for concept questions in a single PDF file. Export/print the Jupyter notebook as a PDF file including the code you implemented and the outputs of the program. Make sure all plots and outputs are visible in the PDF.

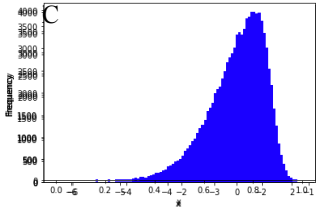
Combine all answers into a single PDF named andrewID_hw6.pdf and submit it to Gradescope before the due date. Refer to the syllabus for late homework policy. Please assign each question a page by using the "Assign Questions and Pages" feature in Gradescope.

Problem 1 (2.5 points)

Multiple choice (select one)
Given the original data in A, the data in B and C appears to be:

1. B) Normalized and C) Standardized
2. B) Standardized and C) Normalized
3. B) and C) both unchanged from the original data

C



Problem 2 (2.5 Points)

Multiple Choice (select one)
Which scaling technique would be best to use on the following data:
X = [0.002, 0.01, 100000, 4000, 500, 0.00008, 7]

1. Normalization
2. Standardization
3. Log Transformation

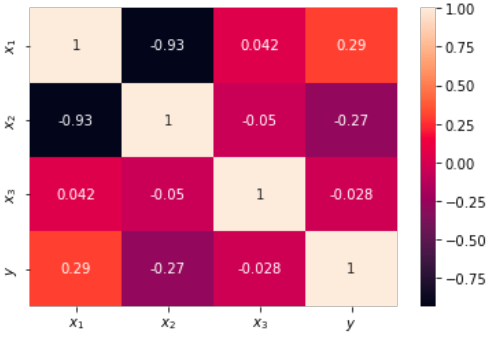
Problem 3 (2.5 Points)

Compute the Pearson's correlation coefficient for the following two features by hand:
$x_1 = [8,4,0,-4]$, $x_2 = [-16,-12,-10,2]$

Problem 4 (2.5 Points)

Multiple choice (select one)
Consider the dataset with features x_1, x_2, x_3, and label y. We have generated the following correlation matrix, and would like to select a feature to remove. We have set the the following threshold $|r| > 0.9$ to drop features. Which of the features should be dropped?

1. x_1
2. x_2
3. x_3



Answers:

PROBLEM 1:-

A: Shows the original data

B: The shape is very much similar to A, but the data is rescaled such that the peak is between 0 and 1. This shows normalization

C: The shape remains similar to A as well, but the data seems to be centered around zero. This shows standardization.

∴ Answer : B → Normalized
C → Standardized

PROBLEM 2:-

Answer: Log Transform
option (3)

Because, this is particularly useful for dealing with datasets that have a large range of values. This is done by reducing the magnitude of large values, making difference between smaller values more distinct.

PROBLEM 3:-

Formula: $r(a,b) = \frac{\sum (a_i - \bar{a})(b_i - \bar{b})}{\sqrt{\sum (a_i - \bar{a})^2 \sum (b_i - \bar{b})^2}}$

$= \frac{a_{ms} \cdot b_{ms}}{\sqrt{(a_{ms} \cdot a_{ms})(b_{ms} \cdot b_{ms})}}$

$\bar{x} = \frac{8+4-4}{4} = 2$; $\bar{y} = -\frac{36}{4} = -9$

Take, $x_1 = 8$ & $y_1 = -16$;

$\Rightarrow (x_1 - \bar{x})(y_1 - \bar{y}) = -42$

Take, $x_2 = 4$, $y_2 = -12$;

$\Rightarrow (x_2 - \bar{x})(y_2 - \bar{y}) = -6$

Take, $x_3 = 0$ & $y_3 = -10$

$\Rightarrow (x_3 - \bar{x})(y_3 - \bar{y}) = 2$

Take, $x_4 = -4$ & $y_4 = 2$

$\Rightarrow (x_4 - \bar{x})(y_4 - \bar{y}) = -66$

Applying into formula;

$$r = \frac{-112}{\sqrt{80 \times 180}} \Rightarrow \frac{-112}{\sqrt{14400}} \Rightarrow \frac{-112}{120}$$

$$r = -0.9333$$

∴ The Pearson's correlation coefficient (r) is $= -0.9333$

PROBLEM 4:-

From the given matrix, it is seen that the absolute value of the correlation coefficient between feature x_1 & x_2 is 0.93 which exceeds $|r| > 0.9$

Thus, one of feature x_1 or x_2 should be dropped to avoid multicollinearity

Answer: option 2 = x_2

Problem 1 (6 Points)

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 =

$$\begin{bmatrix} 10 & 2 \\ 9 & 0 \\ 3 & 8 \\ 4 & 9 \\ 0 & 8 \\ -3 & 2 \\ -4 & -2 \\ 5 & 6 \end{bmatrix}$$

\overset{X}{\longrightarrow} \begin{bmatrix}

1. & 0.364\ 0.929 & 0.182\ 0.5 & 0.909\ 0.571 & 1.\ 0.286 & 0.909\ 0.071 & 0.364\
2. & 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 [5]:
```
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
    mc = (X -Min)/(Max -Min)
    return mc

# 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 =

$$\begin{bmatrix} 10 & 2 \\ 9 & 0 \\ 3 & 8 \\ 4 & 9 \\ 0 & 8 \\ -3 & 2 \\ -4 & -2 \\ 5 & 6 \end{bmatrix}$$

\overset{X}{\longrightarrow} \begin{bmatrix} 1.46 & -0.547\ 1.251 & -1.061\

1. & 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 [6]:
```
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
```

```
    sc = (X-mu)/sigma
    return sc

# 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))
```

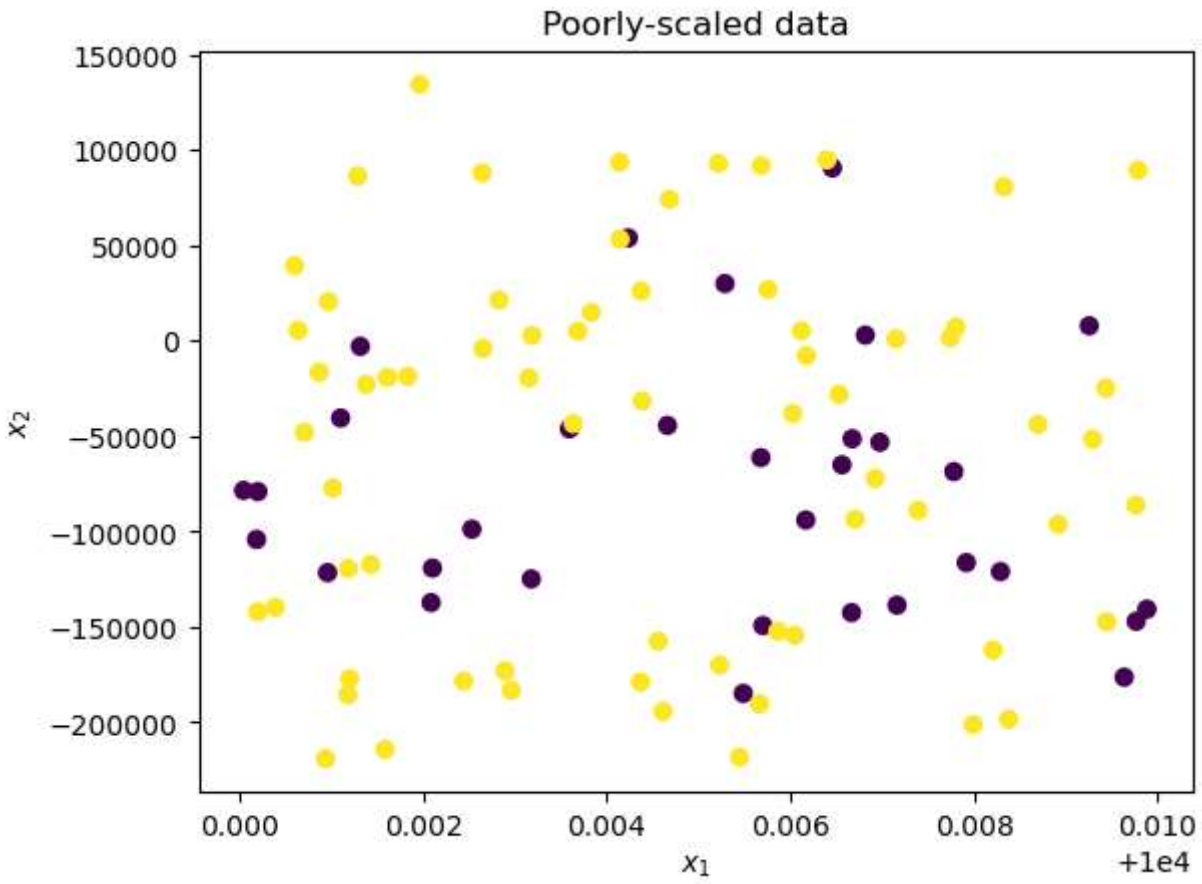
```
X =
 [[10  2]
 [ 9  0]
 [ 3  8]
 [ 4  9]
 [ 0  8]
 [-3  2]
 [-4 -2]
 [ 5  6]]  -->
 [[ 1.46  -0.547]
 [ 1.251 -1.061]
 [ 0.     0.997]
 [ 0.209  1.254]
 [-0.626  0.997]
 [-1.251 -0.547]
 [-1.46  -1.576]
 [ 0.417  0.482]]
A =
 [[ 2  1]
 [ 5  5]
 [11 -1]
 [-3  2]]  -->
 [[-0.209 -0.804]
 [ 0.417  0.225]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

Problem 2 (6 Points)

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 [17]:
```
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, 10000.00
x2 = np.array([-184863.4856705 ,    1074.38382588,  -38090.38042426, -218261.93176495,   53942.6974416 ,   90630.025842
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, 0,

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 [18]:
```
# 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, pred_te
```

```
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 [19]:
```
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))

        # YOUR CODE GOES HERE
        # Create a pipeline
        # Fit the pipeline on X_train, y_train
        # Calculate acc_train and acc_test for the pipeline
        # List of scalers and classifiers to iterate over

for i in range(3):
    for j in range(3):
        scaler_name, scaler = get_scaler(i)
        model_name, model = get_model(j)
        if scaler:
            pipe = Pipeline([(scaler_name,scaler), (model_name,model)])
        else:
            pipe = Pipeline([(model_name,model)])
        pipe.fit(X_train,y_train)
        acc_tr = accuracy_score(y_train, pipe.predict(X_train))
        acc_te = accuracy_score(y_test, pipe.predict(X_test))
        print(f"{scaler_name:>20}, {model_name:>26}:    Train Acc. = {100*acc_tr:5.1f}%    Test Acc. = {100*acc_te:5.1f
```

```
          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?

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

In [25]:
```
print("1) The model's testing accuracy that was improved the most by scaling data is the 1-NN Classifier. Without scali
print()
print("2) For this data, both the MinMax and Standard methods work about the same across different models. Looking at t
```

```
1) The model's testing accuracy that was improved the most by scaling data is the 1-NN Classifier. Without scaling, it
achieved a testing accuracy of 50.0%, but with both MinMax and Standard scaling it reached an accuracy of 85.0%.

2) For this data, both the MinMax and Standard methods work about the same across different models. Looking at the '1-N
N Classifier', using either method helps a lot compared to not using any method. So, for this data and the models used,
both methods are good, and one isn't better than the other.
```

Problem 3 (6 Points)

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.

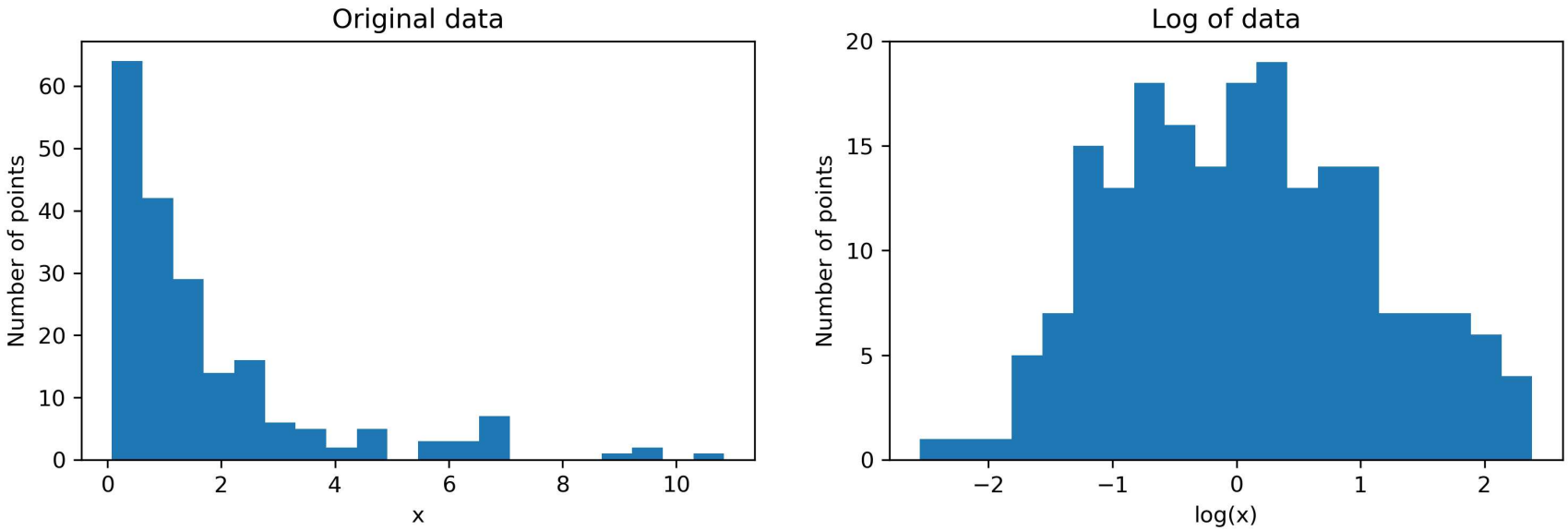
Start by running this cell to import modules and load data:

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,  0.85954061,  0
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+00, -6.2

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()
```

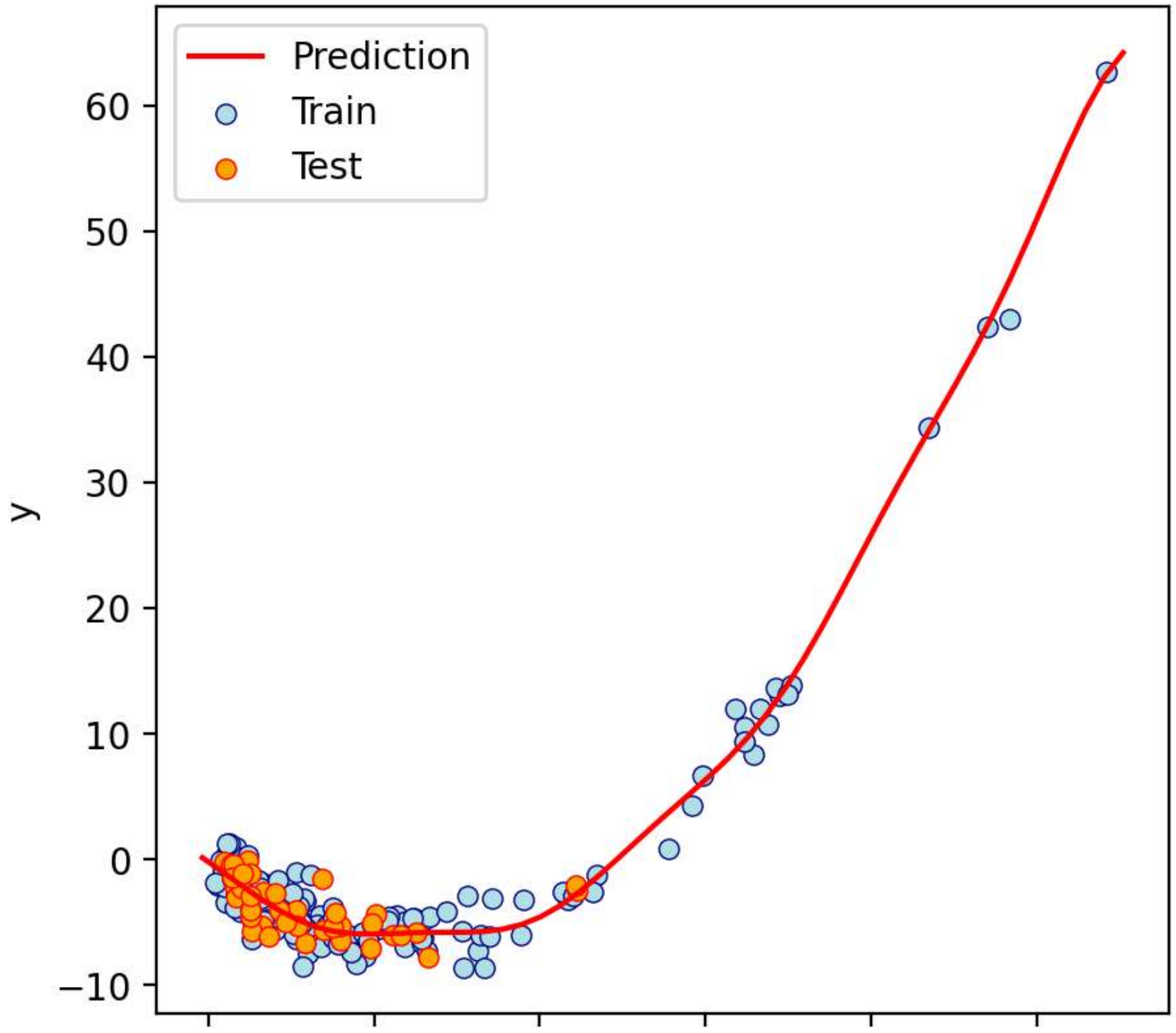


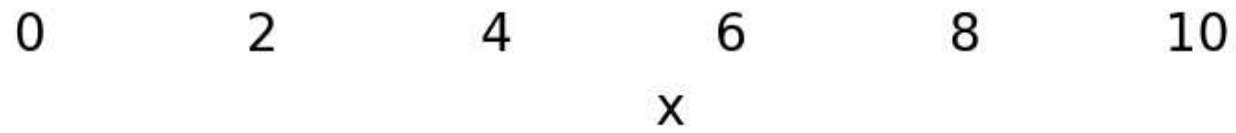
No log transform

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

In [9]:
```
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, pred_test

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

```
Training MSE: 1.9641609271806786    Testing MSE: 1.9088942378065092
```


Prediction
Train
Test
60
50
40
30
20
10
0
−10
y



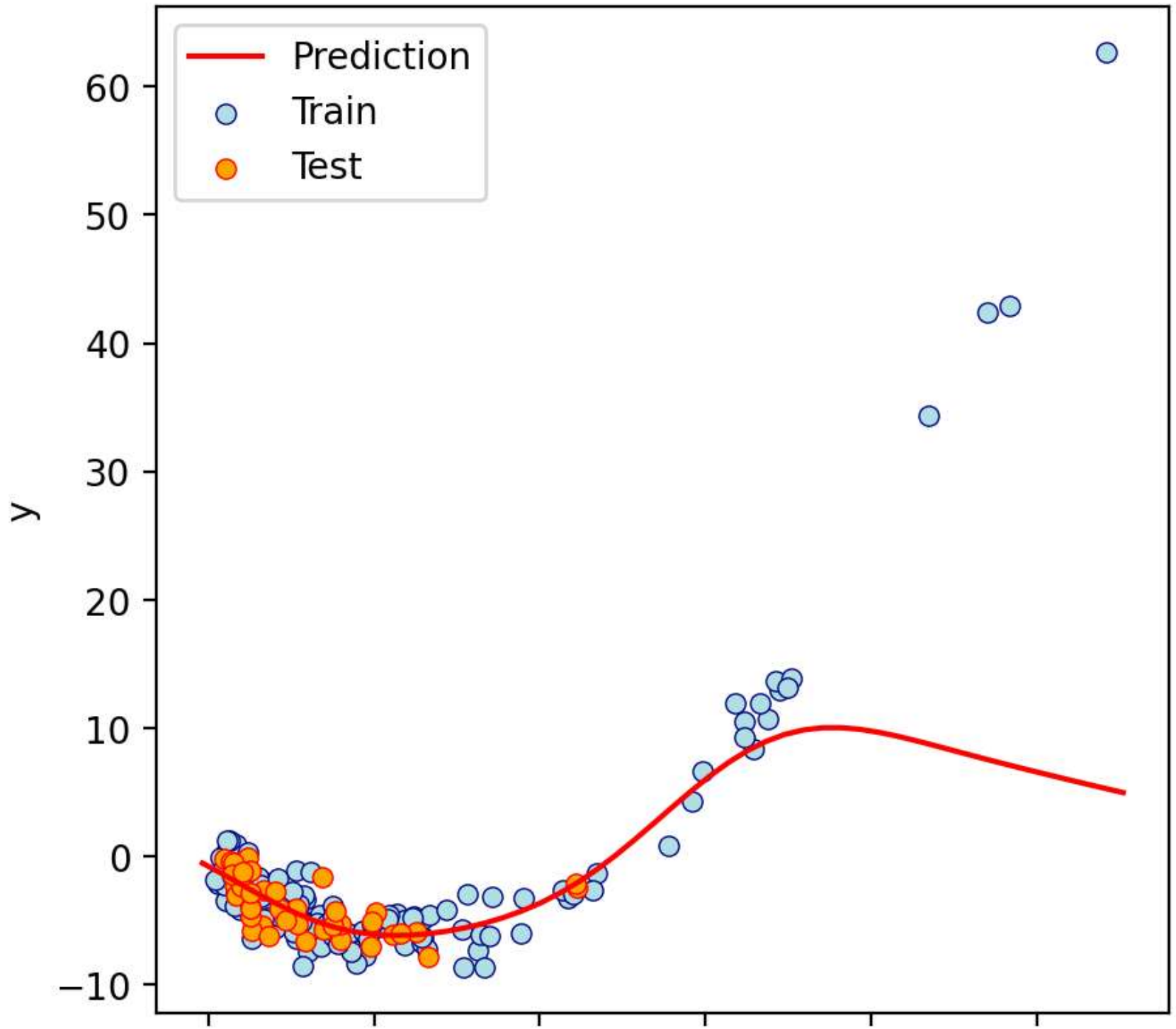
With log transform

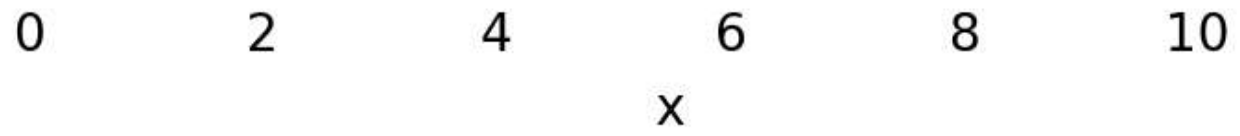
Notice that the data are not spread uniformly across the x axis. Instead, most input data points have low values -- this is a roughly "log normal" distribution. If we take the log of the input, we saw it was more normally distributed, which can improve machine learning model results in some cases. The 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.

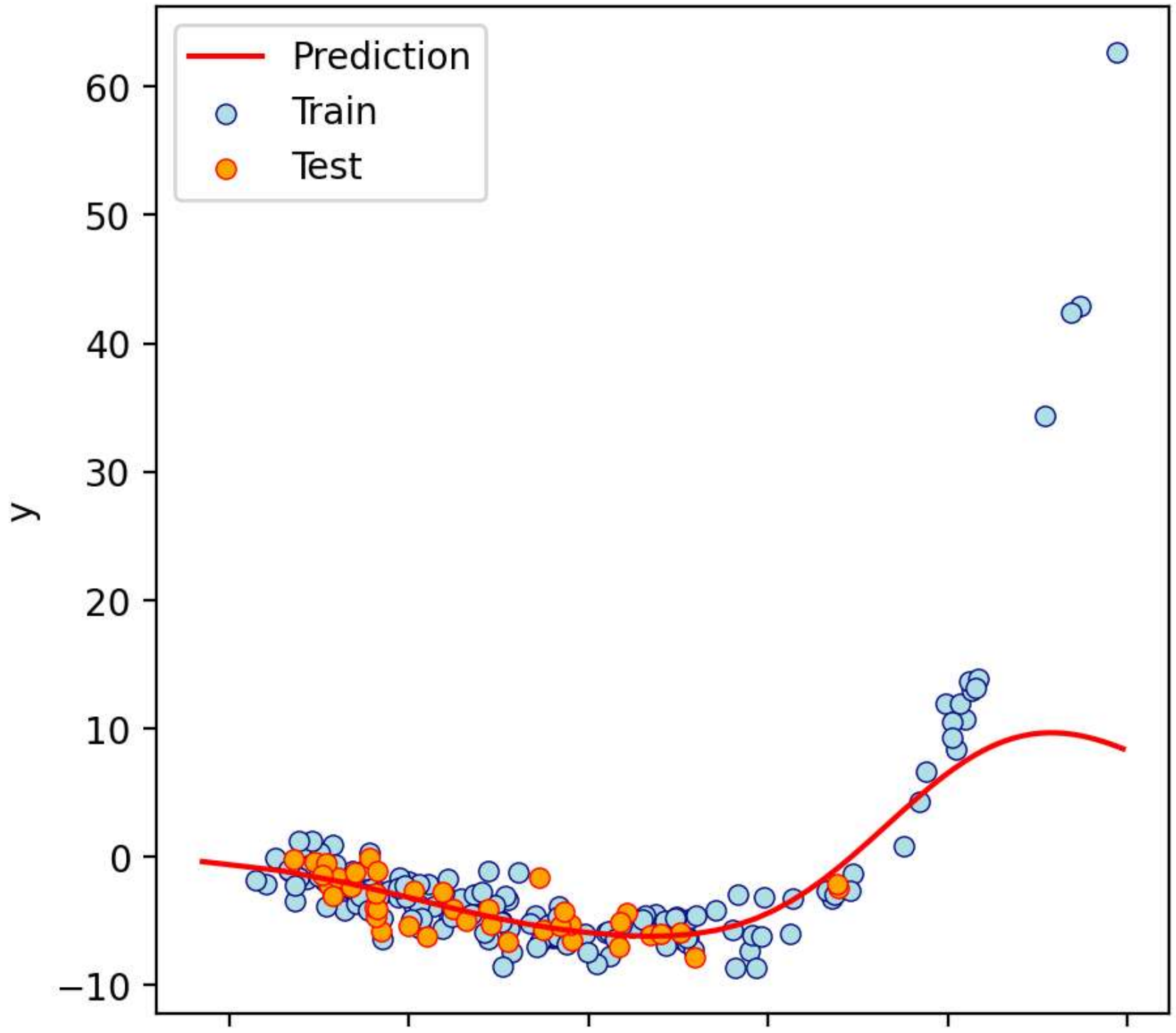
Also, make another plot setting the `log` argument to `True`. This will show the scaling of the x-axis used by the model.

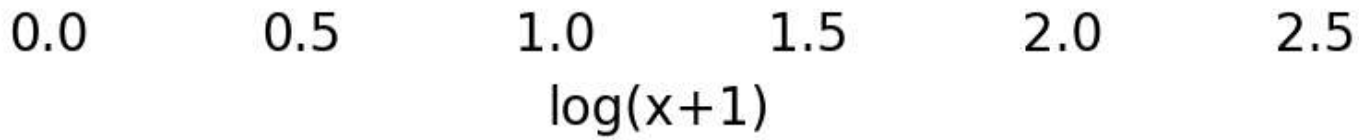
In [10]:
```
def log_transform(x):
    return np.log(x + 1.)

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


# YOUR CODE GOES HERE
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
logt = FunctionTransformer(np.log1p, validate=True)
pipe_log = Pipeline([('log_transform', logt),('regressor', SVR())])
pipe_nolog = Pipeline([('regressor', SVR())])
pipe_log.fit(X_train, y_train)
pipe_nolog.fit(X_train, y_train)
y_pred_log = pipe_log.predict(X_test)
y_pred_nolog = pipe_nolog.predict(X_test)
mse_log = mean_squared_error(y_test, y_pred_log)
mse_nolog = mean_squared_error(y_test, y_pred_nolog)

print(f"The MSE with Log Transformation is = {mse_log:.2f}")
print(f"The MSE without Log Transformation is = {mse_nolog:.2f}")
plot(X_train, X_test, y_train, y_test, model=pipe_nolog, log=False)
plot(X_train, X_test, y_train, y_test, model=pipe_log, log=True)
```

```
The MSE with Log Transformation is = 2.01
The MSE without Log Transformation is = 1.94
```


Prediction
Train
Test
60
50
40
30
20
10
0
−10
y


0
2
4
6
8
10
x


Prediction
Train
Test
60
50
40
30
20
10
0
−10
y


0.0
0.5
1.0
1.5
2.0
2.5
log(x+1)

Problem 4 (6 Points)

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 [41]:
```
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], [22.8,4,108,93,3.85,2.3
                 [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], [24.4,4,146.7,62,3.
                 [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],[17.3,8,275.8,180,3
                 [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],[32.4,4,78.7,66,4.08,2
                 [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],[15.2,8,304,150,3.15
                 [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],[30.4,4,95.1,113,3.77,1.5
                 [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 [42]:
```
def find_redundant_features(data, target_index, threshold):
    # YOUR CODE GOES HERE
    cm = np.corrcoef(data.T)
    abm = np.abs(cm)
    ir = set()
    for i in range(abm.shape[0]):
        for j in range(i+1, abm.shape[1]):
            if abm[i,j] > threshold and i != target_index and j != target_index:

                if abm[i, target_index] < abm[j, target_index]:
                    ir.add(i)
                else:
                    ir.add(j)

    return list(ir)
```

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 [43]:
```
# YOUR CODE GOES HERE
test = [(0, 0.9),(2, 0.7),(10, 0.8)]
for target, threshold in test:
    print(f"The target index is = {target} | The threshold is = {threshold} | The Indices to remove are = {find_redunda
```

```
The target index is = 0 | The threshold is = 0.9 | The Indices to remove are = [2]
The target index is = 2 | The threshold is = 0.7 | The Indices to remove are = [0, 3, 4, 5, 6, 7, 8, 9, 10]
The target index is = 10 | The threshold is = 0.8 | The Indices to remove are = [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 [44]:
```
# YOUR CODE GOES HERE
test2 = [(4, 0.9),(5, 0.8),(6, 0.95)]
for target, threshold in test2:
    print(f"The target index is = {target} | The threshold is = {threshold} | The Indices to remove are = {find_redunda
```

```
The target index is = 4 | The threshold is = 0.9 | The Indices to remove are = [1]
The target index is = 5 | The threshold is = 0.8 | The Indices to remove are = [0, 1, 3, 7]
The target index is = 6 | The threshold is = 0.95 | The Indices to remove are = []
```

Problem 5 (6 Points)

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 [33]:
```
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], [22.8,4,108,93,3.85,2.3
                 [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], [24.4,4,146.7,62,3.
                 [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],[17.3,8,275.8,180,3
                 [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],[32.4,4,78.7,66,4.08,2
                 [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],[15.2,8,304,150,3.15
                 [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],[30.4,4,95.1,113,3.77,1.5
                 [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 [34]:
```
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 [35]:
```
def get_next_pair(X, y, current_indices):
    # YOUR CODE GOES HERE
    mm = float('inf')
    bp = None
    ir = [i for i in range(X.shape[1])
          if i not in current_indices]

    for pair in itertools.combinations(ir, 2):
        it = np.array(list(current_indices) + list(pair), dtype=int)
        _, mt = get_train_test_mse(X, y, it)
        if mt < mm:
            mm = mt
            bp = pair

    return bp
```

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 [36]:
```
indices = np.array([])
while len(indices) < 8:
    pair = get_next_pair(X, y, indices)
    print(f"Adding pair {pair}")
    indices = np.union1d(indices, pair)
```

```
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?

ANSWER: When examining indices from 0 to 10: 0,1,2,3,4,5,6,7,8,9,10

And subtracting the selected indices: 2,5,0,8,6,7,4,9

We are left with: 1,3,10

Given that we're seeking the 2 least significant feature indices, and since the method finalizes with 8 features, the three non-selected features can be ranked by their non-selection, with the last two being the least significant.

Thus, the two least important feature indices according to this wrapper method are:

3 and 10.

Problem 6 (30 Points)

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 [61]:
```
import numpy as np
import matplotlib.pyplot as plt
```

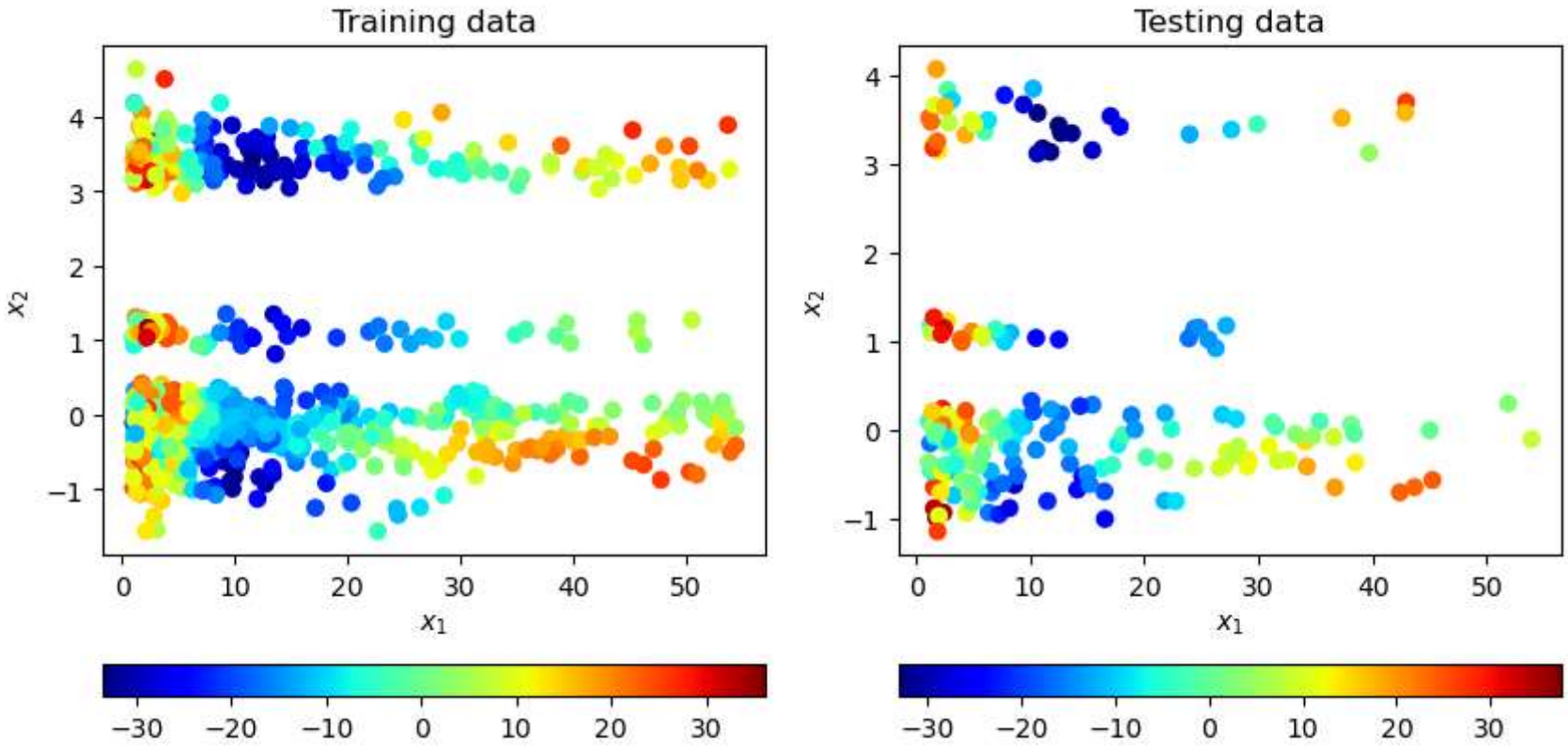
```
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import PolynomialFeatures, QuantileTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics 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 [62]:
```
# YOUR CODE GOES HERE
# Define path
path = r"C:\Users\srech\Downloads\w6-p1-data.npy"
data = np.load(path)
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()
```



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 [63]:
```
model_names = ["LSR", "SVR", "KNN", "RF"]

# YOUR CODE GOES HERE
mod = [Pipeline([("poly",PolynomialFeatures(degree=7)),("lr",LinearRegression())]),SVR(C=1000,kernel="rbf"),KNeighborsR
```

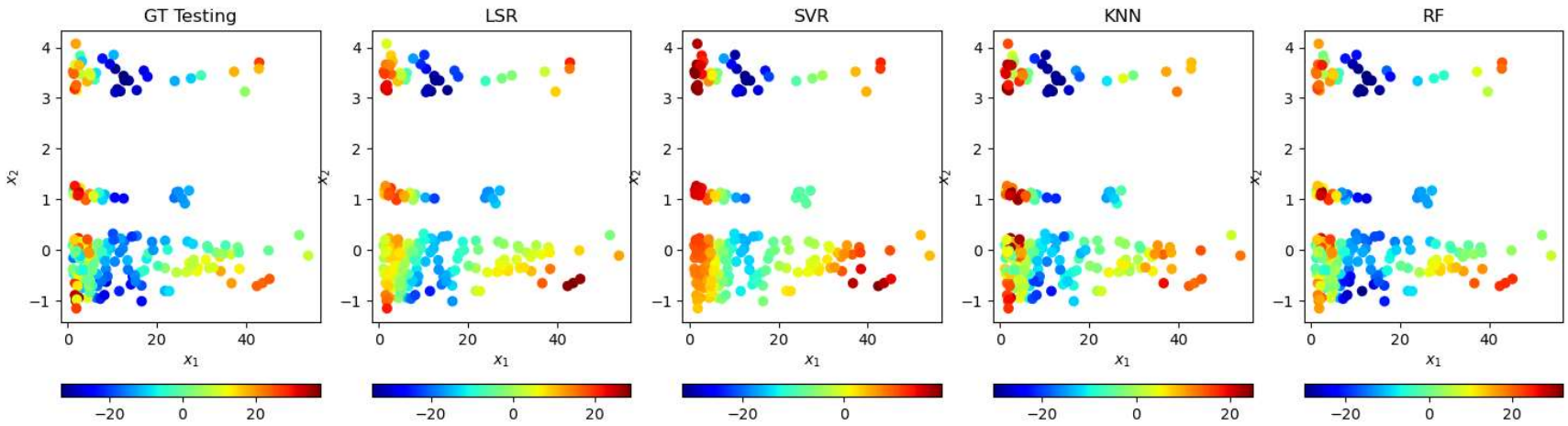
```
l = len(model_names)
for i in range (l):
    m = mod[i]
    m.fit(X_train,y_train)
    train = mean_squared_error(y_train,m.predict(X_train))
    test = mean_squared_error(y_test,m.predict(X_test))

    print(f"{model_names[i]} | The Train MSE is = {train} & The Test MSE is = {test}")
```

```
LSR | The Train MSE is = 50.86639055262535 & The Test MSE is = 57.28724902769158
SVR | The Train MSE is = 82.04352603565974 & The Test MSE is = 98.63319719407525
KNN | The Train MSE is = 26.856498566141628 & The Test MSE is = 47.63617328402055
RF | The Train MSE is = 5.751475923567525 & The Test MSE is = 24.607478006437677
```

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 [64]:
```
plt.figure(figsize=(18,5))
plt.subplot(1,5,1)
plot(X_test, y_test, "GT Testing")

# YOUR CODE GOES HERE
l = len(model_names)
for i in range (l):
    plt.subplot(1,5,i+2)
    plot(X_test, mod[i].predict(X_test), model_names[i])
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 [65]:
```
pipeline_names = ["LSR, scaled", "SVR, scaled", "KNN, scaled", "RF, scaled"]

# YOUR CODE GOES HERE
s = QuantileTransformer(n_quantiles=800)
pipe = [Pipeline([("scaler", s),("poly", PolynomialFeatures(degree=7)), ("lr", LinearRegression())]),Pipeline([("scaler

n = len(pipeline_names)
for i in range(n):
    pipe[i].fit(X_train, y_train)
    train = mean_squared_error(y_train, pipe[i].predict(X_train))
    test = mean_squared_error(y_test, pipe[i].predict(X_test))
    print(f"{pipeline_names[i]} | The Train MSE is = {train} & The Test MSE is = {test}")
```

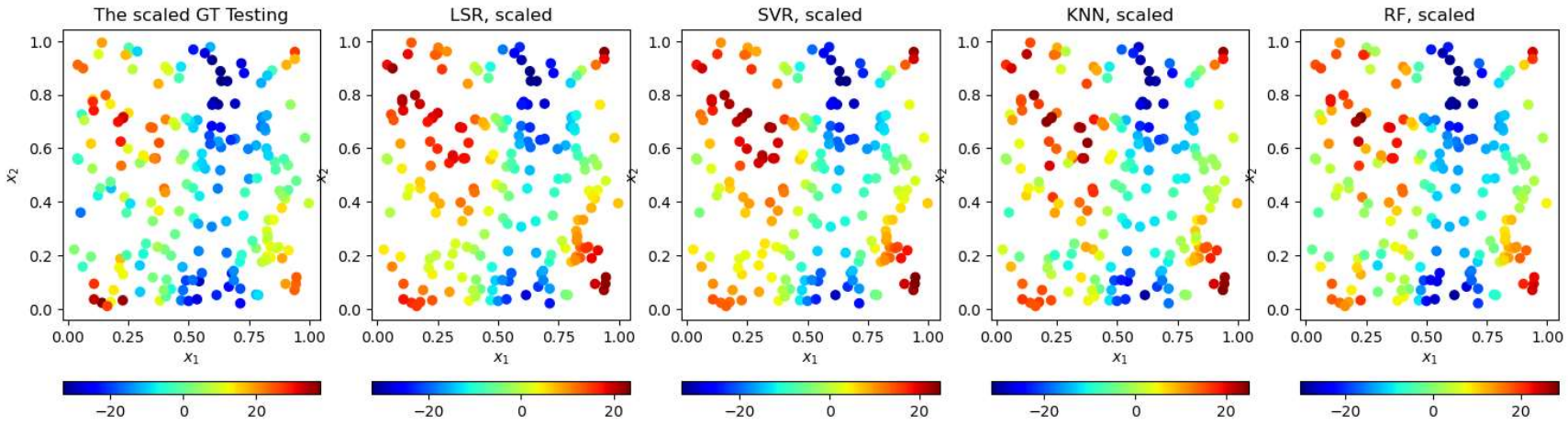
```
LSR, scaled | The Train MSE is = 39.52893428670361 & The Test MSE is = 43.203634922513174
SVR, scaled | The Train MSE is = 41.03425800595977 & The Test MSE is = 43.017915737897745
KNN, scaled | The Train MSE is = 19.687691313922564 & The Test MSE is = 36.397038931930005
RF, scaled | The Train MSE is = 5.859990915084263 & The Test MSE is = 25.79323233084031
```

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 [66]:
```
# YOUR CODE GOES HERE
X_test_s= scaler.transform(X_test)
plt.figure(figsize=(18,5))
plt.subplot(1,5,1)
plot(X_test_s,y_test,"The scaled GT Testing")

n = len(pipeline_names)
for i in range(n):
    plt.subplot(1,5,i+2)
    plot(X_test_s, pipe[i].predict(X_test), pipeline_names[i])
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.

In [67]:
```
print("1) Without transforming the input data, the Random Forest (RF) model performed the best on the test data with th
print()
print("""2) LSR (Linear Regression, with features): Scaling the data enhanced the performance of LSR noticeably. The im

SVR (Support Vector Regression): Scaling also had an impact on SVR's performance. The difference was quite substantial

KNN (K Nearest Neighbors): Applying scaling improved KNN's performance well. The change was moderate but noteworthy res

RF (Random Forest): Scaling had minimal impact on RF's performance. The Test MSE remained almost unchanged before and a
""")
```

```
1) Without transforming the input data, the Random Forest (RF) model performed the best on the test data with the lowes
t Test MSE. After scaling, the Random Forest (RF) model still performed the best on the test data with the lowest Test
MSE.

2) LSR (Linear Regression, with features): Scaling the data enhanced the performance of LSR noticeably. The improvement
was moderate but significant resulting in a reduction of the Test MSE by 14 units. This could be attributed to the fact
that scaling assists linear regression models when the features exhibit varying scales or non-standard distributions. P
articularly, quantile scaling can promote a relationship between variables.

SVR (Support Vector Regression): Scaling also had an impact on SVR's performance. The difference was quite substantial
leading to a decrease in Test MSE by around 55 units. SVR, with the RBF kernel, is highly sensitive to feature scaling.
Bringing all features to a scale can significantly enhance its effectiveness.

KNN (K Nearest Neighbors): Applying scaling improved KNN's performance well. The change was moderate but noteworthy res
ulting in a reduction of the Test MSE by 11 units. Scaling plays a pivotal role in KNN since it relies on calculating d
istances between data points. If one feature has a larger range than others, it can dominate distance calculations and
impact accuracy.

RF (Random Forest): Scaling had minimal impact on RF's performance. The Test MSE remained almost unchanged before and a
fter scaling, with a slight decrease observed after scaling. This is because decision trees, which are a component of t
he forest, are not as affected by the data's scale. However, there might be some enhancements if scaling assists in imp
roving the segmentation or division of the data points.
```

Problem 7 (30 Points)

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 [44]:
```
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()
```

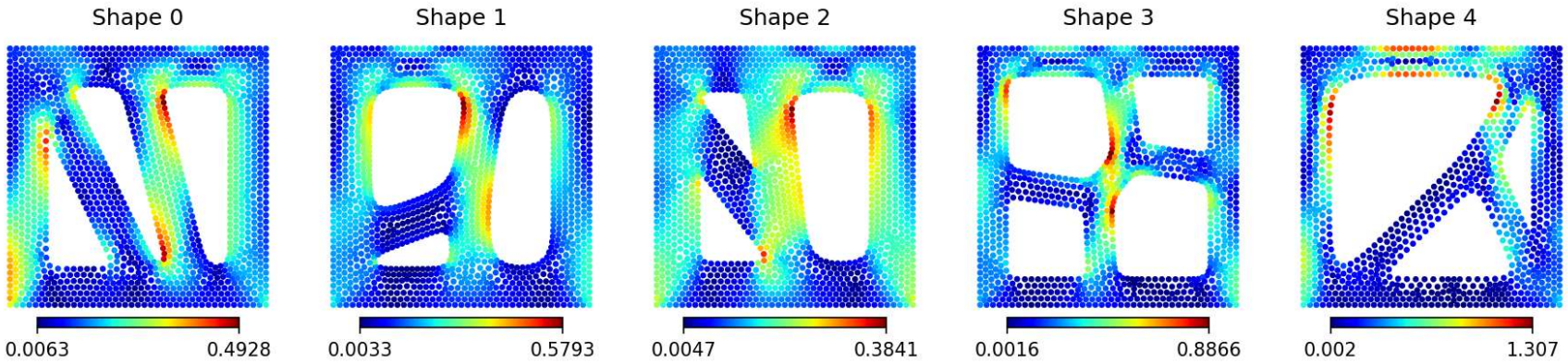
Loading the data

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 [45]:
```
# YOUR CODE GOES HERE
# Define data_path
data_path = r"C:\Users\srech\Downloads\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 [46]:
```
# YOUR CODE GOES HERE
lr = LinearRegression().fit(X_train,y_train)
dt = DecisionTreeRegressor(max_depth=20).fit(X_train,y_train)
lr_trp = lr.predict(X_train)
lr_tep = lr.predict(X_test)
dt_trp = dt.predict(X_train)
dt_tep = dt.predict(X_test)

lr_trm = mean_squared_error(y_train,lr_trp)
lr_tem = mean_squared_error(y_test,lr_tep)
dt_trm = mean_squared_error(y_train,dt_trp)
dt_tem = mean_squared_error(y_test,dt_tep)

print("The Linear Regression MSE | Train =", lr_trm, "& Test =", lr_tem)
print("Decision Tree MSE | Train =", dt_trm, "& Test =", dt_tem)
```

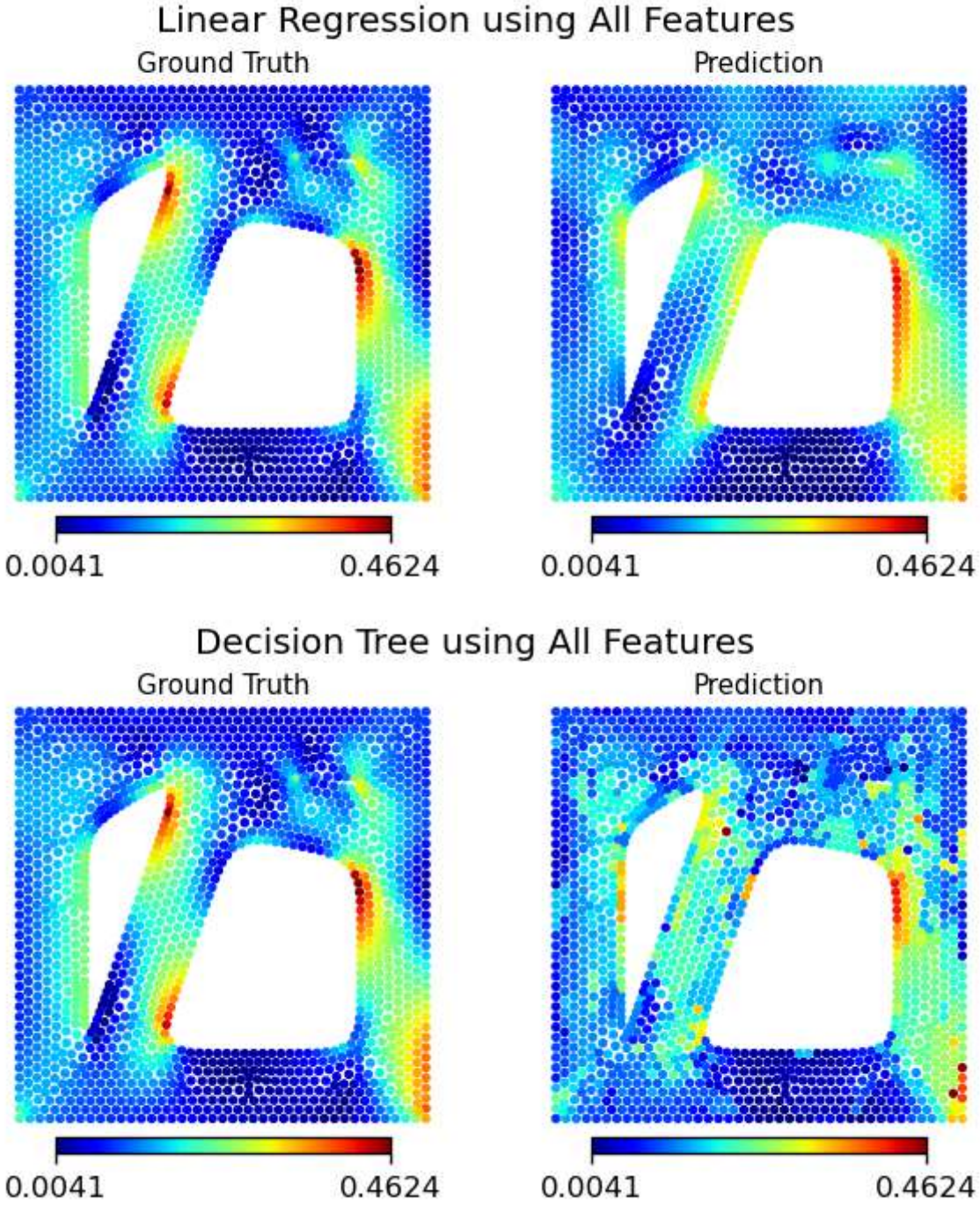
```
The Linear Regression MSE | Train = 0.008110601 & Test = 0.009779482
Decision Tree MSE | Train = 0.0004944875978805109 & Test = 0.00826281263461088
```

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 [47]:
```
test_idx = 8

# YOUR CODE GOES HERE
plot_shape_comparison(dataset_test,test_idx,lr,title="Linear Regression using All Features")
plot_shape_comparison(dataset_test,test_idx,dt,title="Decision Tree using All Features")
```

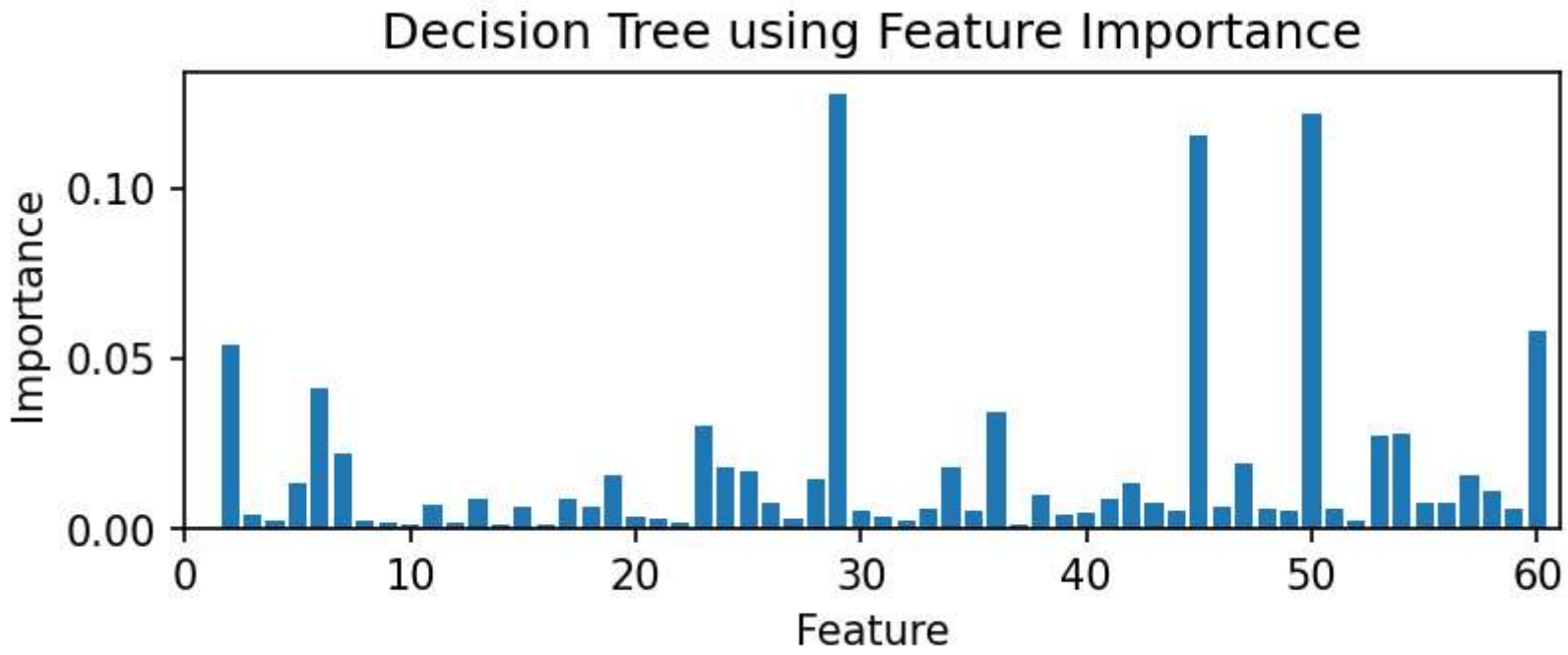

Linear Regression using All Features
Ground Truth
Prediction
0.0041
0.4624
0.0041
0.4624
Decision Tree using All Features
Ground Truth
Prediction
0.0041
0.4624
0.0041
0.4624

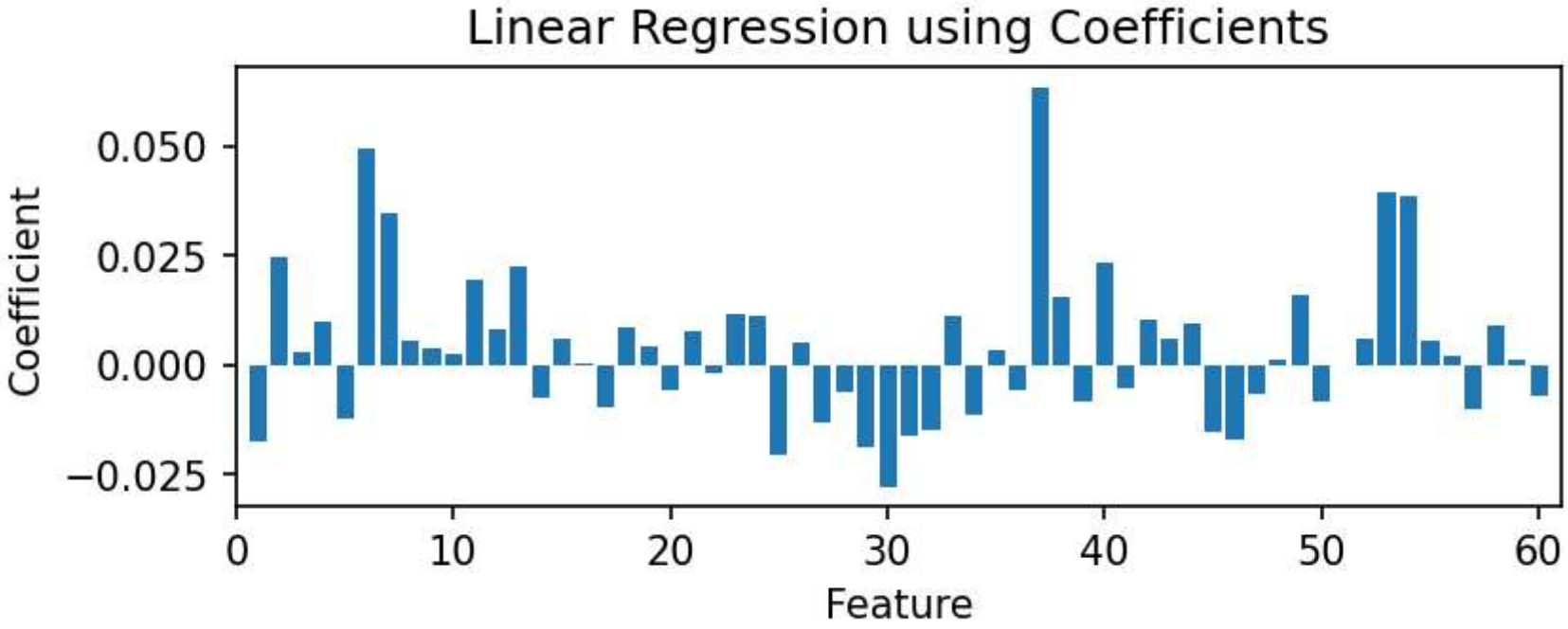
Feature importance

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 [48]:
```
# YOUR CODE GOES HERE
plot_importances(dt,title="Decision Tree using Feature Importance")
plot_importances(lr,coef=True, title="Linear Regression using Coefficients")
```





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 [49]:
```
# YOUR CODE GOES HERE
rfe_lr = RFE(LinearRegression(),n_features_to_select=30)
rfe_lr.fit(X_train,y_train)
rfe_dt = RFE(DecisionTreeRegressor(max_depth=20),n_features_to_select=30)
```

```
rfe_dt.fit(X_train,y_train)

# Predict using the models with RFE
lr_rfe_trp = rfe_lr.predict(X_train)
lr_rfe_tep = rfe_lr.predict(X_test)
dt_rfe_trp = rfe_dt.predict(X_train)
dt_rfe_tep = rfe_dt.predict(X_test)

# Calculate MSE
lr_rfe_trm = mean_squared_error(y_train,lr_rfe_trp)
lr_rfe_tem = mean_squared_error(y_test,lr_rfe_tep)
dt_rfe_trm = mean_squared_error(y_train,dt_rfe_trp)
dt_rfe_tem = mean_squared_error(y_test,dt_rfe_tep)

print("Linear Regression with RFE MSE | Train =", lr_rfe_trm, "& Test =", lr_rfe_tem)
print("Decision Tree with RFE MSE | Train =", dt_rfe_trm, "& Test =", dt_rfe_tem)
```

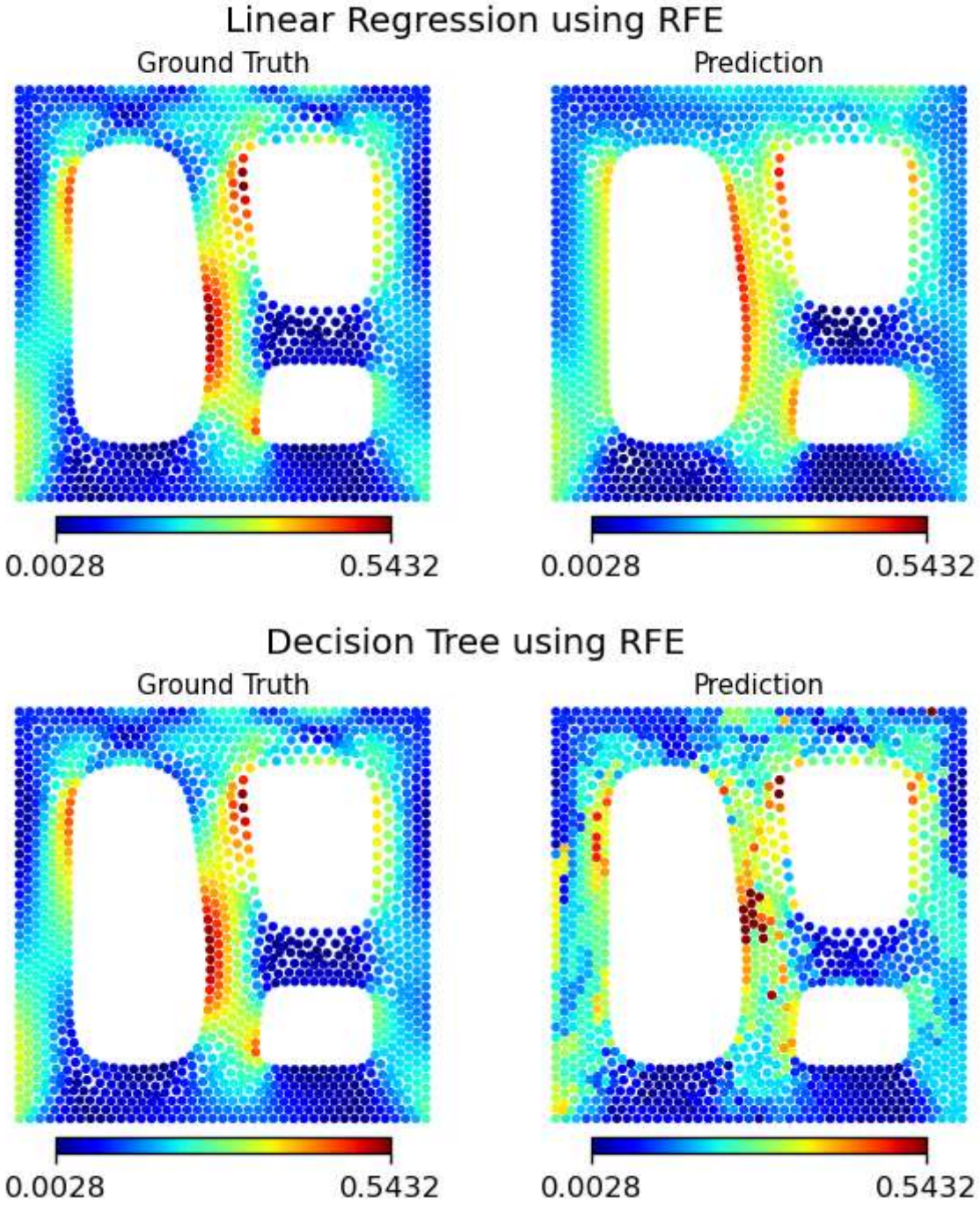
```
Linear Regression with RFE MSE | Train = 0.008508719 & Test = 0.010150377
Decision Tree with RFE MSE | Train = 0.000560310068845692 & Test = 0.009020585012541065
```

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 [50]:
```
test_idx = 16

# YOUR CODE GOES HERE
plot_shape_comparison(dataset_test,test_idx,rfe_lr,title="Linear Regression using RFE")
plot_shape_comparison(dataset_test,test_idx,rfe_dt,title="Decision Tree using RFE")
```

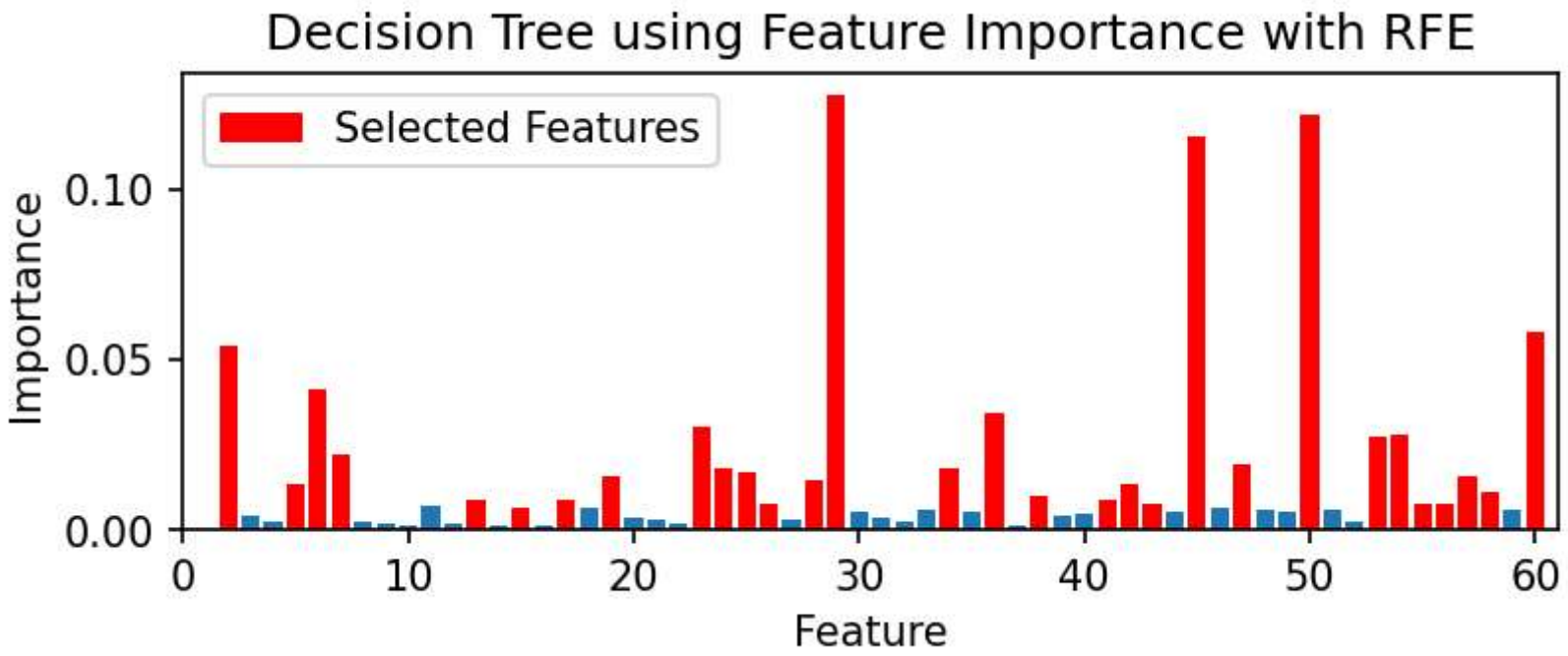

Linear Regression using RFE
Ground Truth
Prediction
0.0028
0.5432
0.0028
0.5432
Decision Tree using RFE
Ground Truth
Prediction
0.0028
0.5432
0.0028
0.5432

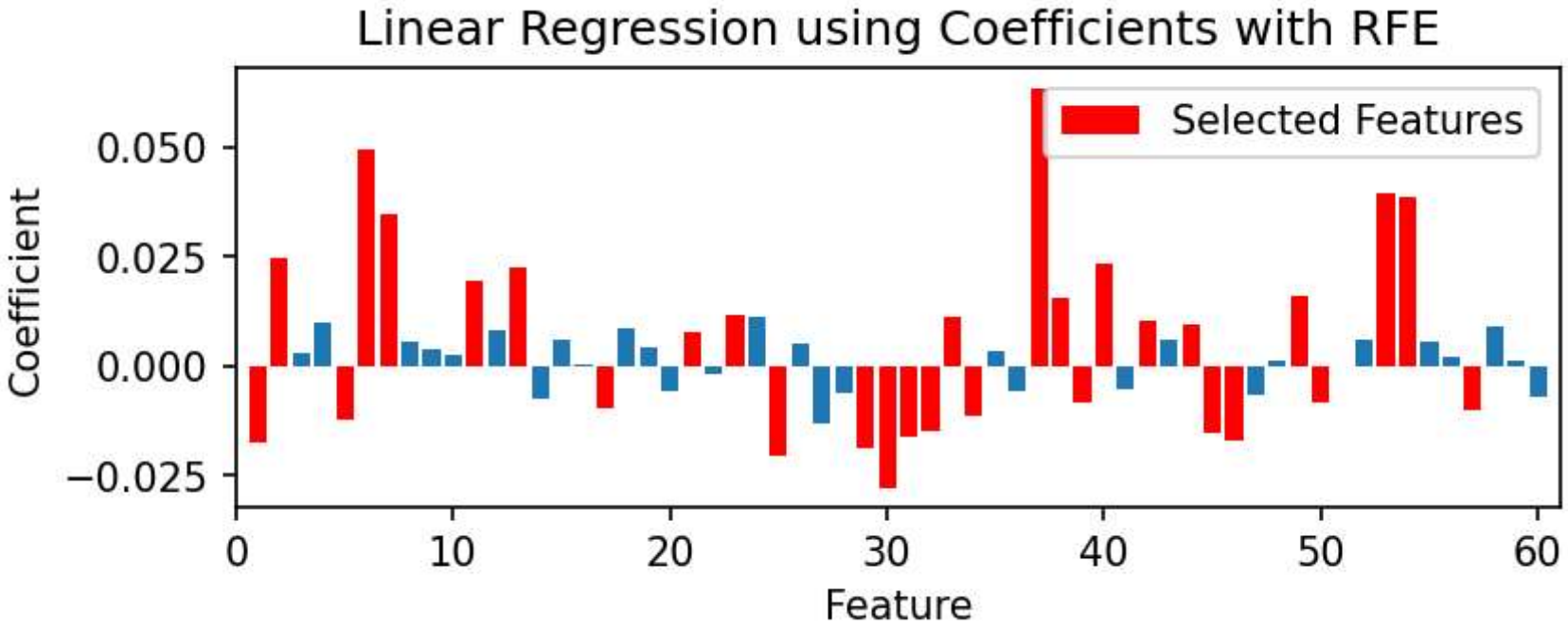
Feature importance with RFE

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 [51]:
```
# YOUR CODE GOES HERE
sel_lr = rfe_lr.get_support(indices=True)
sel_dt = rfe_dt.get_support(indices=True)

plot_importances(dt,selected=sel_dt,title="Decision Tree using Feature Importance with RFE")
plot_importances(lr,selected=sel_lr,coef=True,title="Linear Regression using Coefficients with RFE")
```





Questions

1. Did the MSE increase or decrease on test data for the Linear Regression model after performing RFE?

1. Did the MSE increase or decrease on test data for the Decision Tree model after performing RFE?

1. Describe the qualitative differences between the Linear Regression and the Decision Tree predictions.

1. Describe how the importance of features that were selected by RFE compare to that of features that were eliminated (for the decision tree).

1. Describe how the coefficients that were selected by RFE compare to that of features that were eliminated (for linear regression).

In [59]:
```
print("1) After performing RFE, the test MSE for Linear Regression increased from 0.009779482 to 0.010150377. So, the M
print()
print("2) After performing RFE, the test MSE for the Decision Tree increased from 0.00826281263461088 to 0.009020585012
print()
print("3) Linear Regression produces smoother predictions, often leading to better generalizations on new, unseen data,
print()
```

```
print("4) The features pinpointed by RFE are considered crucial for the prediction endeavor. When you observe the Decis
print()
print("5) In Linear Regression, the coefficients show how important each feature is for making predictions. Features pi
```

```
1) After performing RFE, the test MSE for Linear Regression increased from 0.009779482 to 0.010150377. So, the MSE incr
eased.

2) After performing RFE, the test MSE for the Decision Tree increased from 0.00826281263461088 to 0.009020585012541065.
So, the MSE increased.

3) Linear Regression produces smoother predictions, often leading to better generalizations on new, unseen data, partic
ularly when the relationships are relatively linear. This approach relies on a linear amalgamation of features. Convers
ely, Decision Trees work by setting specific thresholds on features, enabling them to capture non-linear patterns. Thei
r predictions often manifest as step-like constant values, allowing them to detect sharp variations or outliers more ef
ficiently. However, this capability might make them more susceptible to overfitting, especially when the trees have gre
ater depths.

4) The features pinpointed by RFE are considered crucial for the prediction endeavor. When you observe the Decision Tre
e's feature importance visualization, with emphasis on RFE-selected elements, it becomes evident that these highlighted
features carry more weight. On the contrary, the discarded features often have diminished or trivial importance scores,
suggesting their minor role in the tree's overall decision logic.

5) In Linear Regression, the coefficients show how important each feature is for making predictions. Features picked by
RFE have bigger coefficients, meaning they really affect the prediction. Features not chosen by RFE probably have coeff
icients near zero, showing they don't impact much. If we plot these coefficients and highlight the RFE-chosen ones, the
y'll stand out because of their larger values compared to the ones left out.
```