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.

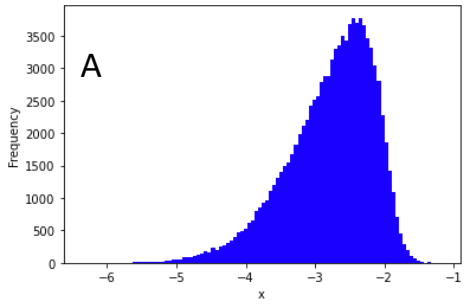
Question	**Points**
Concept 1	3
Concept 2	3
Concept 3	3
Concept 4	3
M6_L1_P1	6
M6_L1_P2	6
M6_L1_P3	9
M6_L2_P1	6
M6_L2_P2	9
M6_HW1	36
M6_HW2	36
Total	**120**
Bonus	6

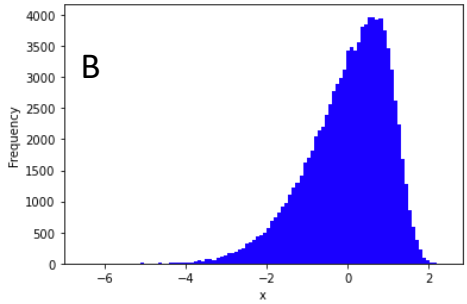
Problem 1

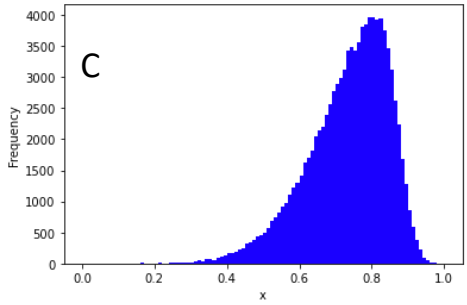
Multiple choice (select one)
Given the original data in A, the data in B and C appears to be:

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

2







Problem 2

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

3, as log transformation handles extreme variations the best.

Problem 3

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

$x_1 = [8, 4, 0, -4] \quad x_2 = [-16, -12, -10, 2]$

$$r = \frac{\sum (x_1 - \bar{x}_1)(x_2 - \bar{x}_2)}{\sqrt{\sum (x_1 - \bar{x}_1)^2 \cdot \sum (x_2 - \bar{x}_2)^2}}$$

$\bar{x}_1 = 2$

$\bar{x}_2 = -9$

$$\sum (x_1 - \bar{x}_1)(x_2 - \bar{x}_2) = \sum [6, 2, -2, -6][-7, -3, -1, 11]$$

$$= -42 - 6 + 2 - 66 = -112$$

$$\sqrt{\sum (x_1 - \bar{x}_1)^2 \sum (x_2 - \bar{x}_2)^2} = \sqrt{(36 + 4 + 4 + 36)(49 + 9 + 1 + 121)}$$

$$= \sqrt{80 \cdot 180} = 120.$$

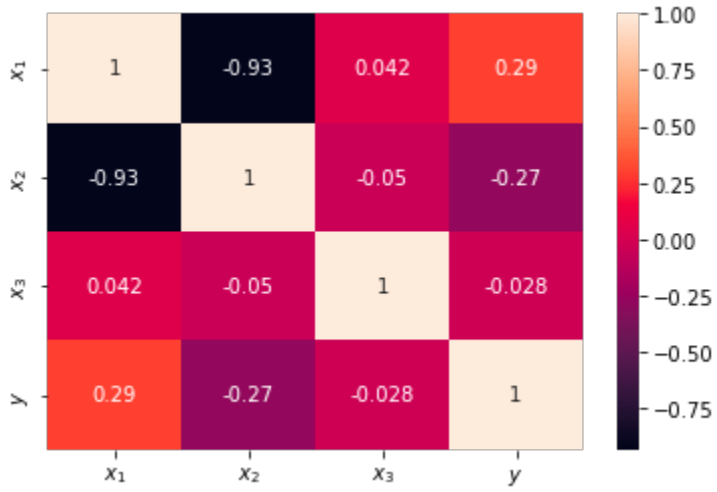
$$r = \frac{-112}{120} \approx -0.933$$

Problem 4

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

2



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 =

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

\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 [4]:

```
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
    return (X-Min)/(Max-Min)

# 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
    return (X-mu)/sigma

# 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]]
```

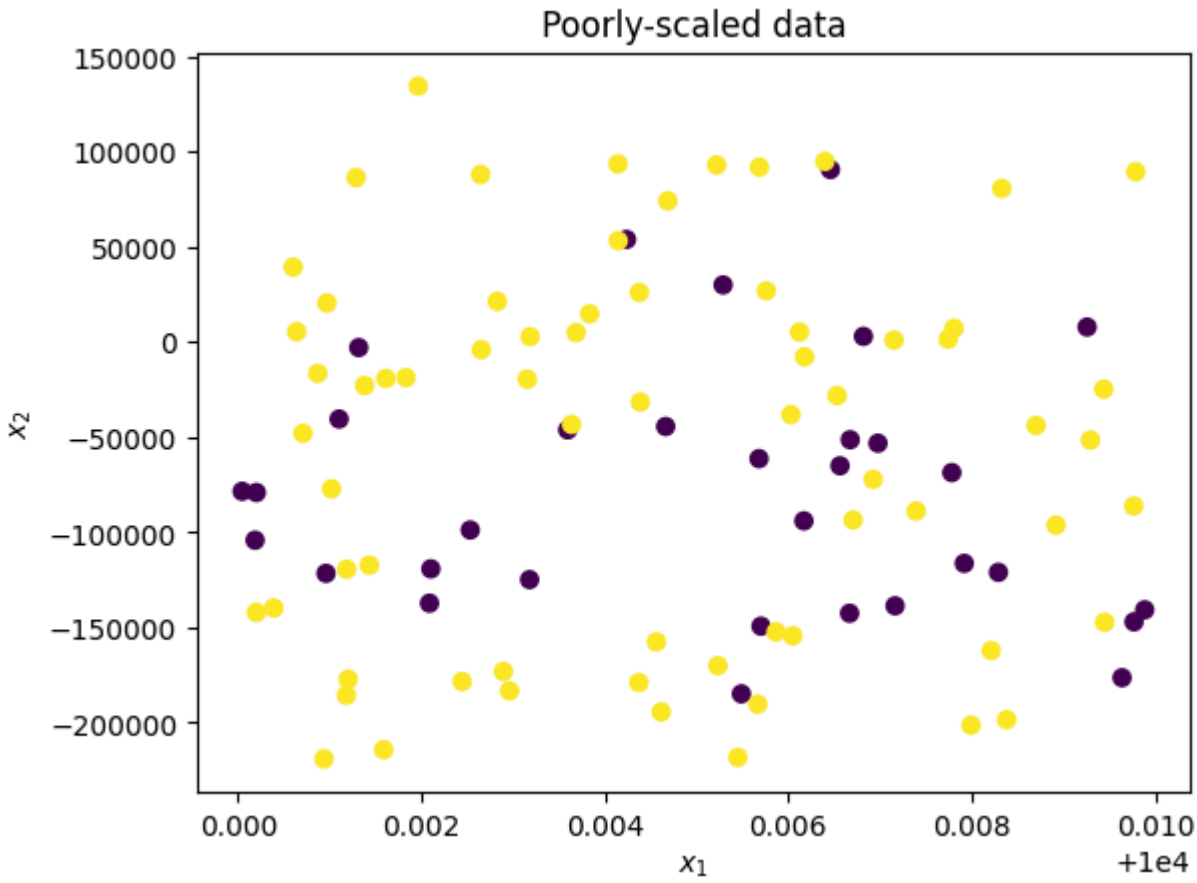
In [ ]:

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, 10000.0
x2 = np.array([-184863.4856705 ,    1074.38382588,  -38090.38042426, -218261.93176495
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,

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

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:
```

```
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
        # Fit the pipeline on X_train, y_train
        # Calculate acc_train and acc_test for the pipeline

        pipeline = Pipeline([scaler, model])
        # Fit the pipeline to the training data
        pipeline.fit(X_train, y_train)
        pred_train = pipeline.predict(X_train)
        pred_test = pipeline.predict(X_test)

        acc_train =  np.sum(pred_train == y_train) / len(y_train)
        acc_test =  np.sum(pred_test == y_test) / len(y_test)

        print(f"{scaler[0]:>15},{model[0]:>26}:    Train Acc. = {100*acc_train: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?

1. Support vector model was imporved the most
2. Standard scaler perform slightly better than MinMax Scaler, but still not great.

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:

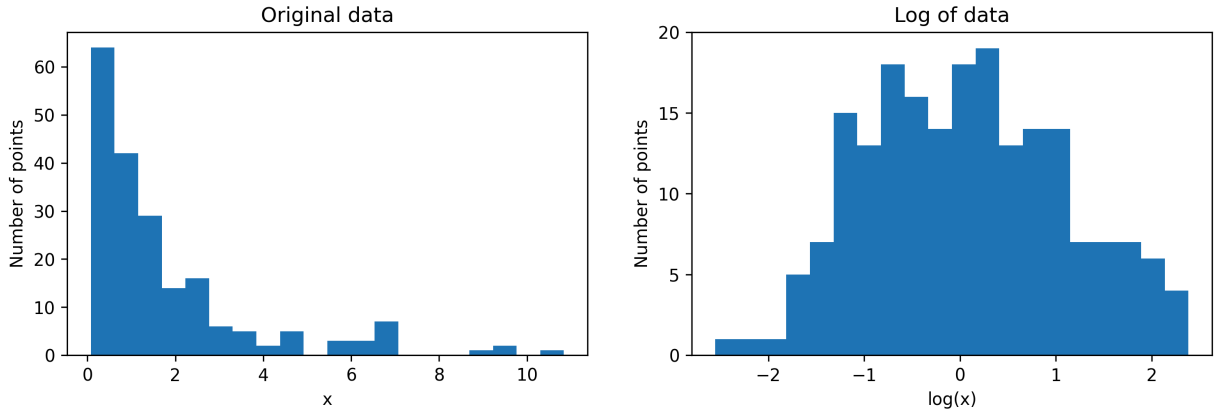
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer
from sklearn.svm import SVR

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,l
    plt.scatter(X_test,y_test,s=30,c="orange",edgecolors="red",linewidths=.5,label="T
    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, 0.37633
X = x.reshape(-1,1)
y = np.array([ 4.32538472e+00, -5.59312420e+00, -4.57455876e+00,  4.23667057e+01, 1.04

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

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()
```



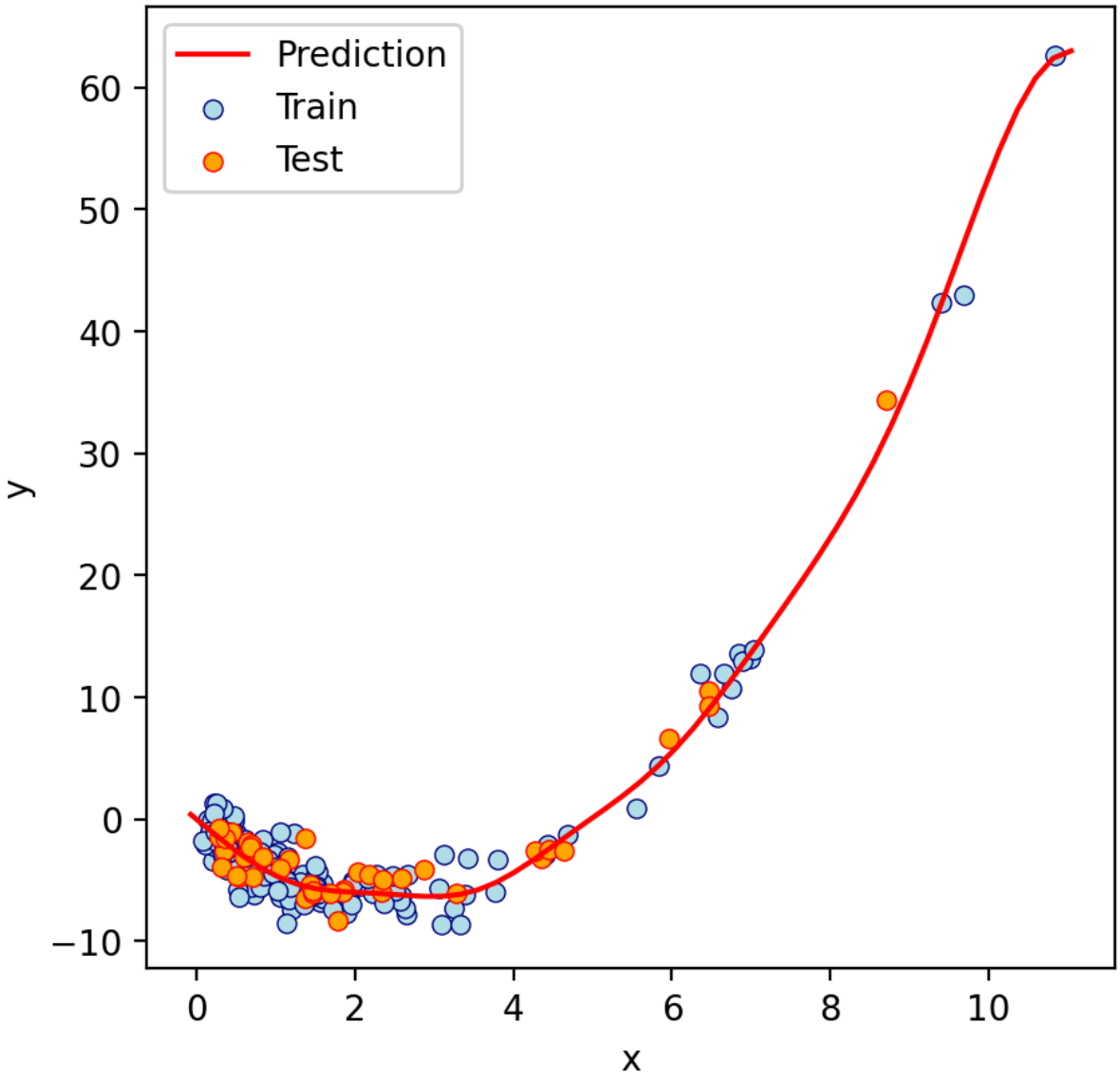
No log transform

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:", mea

# 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.

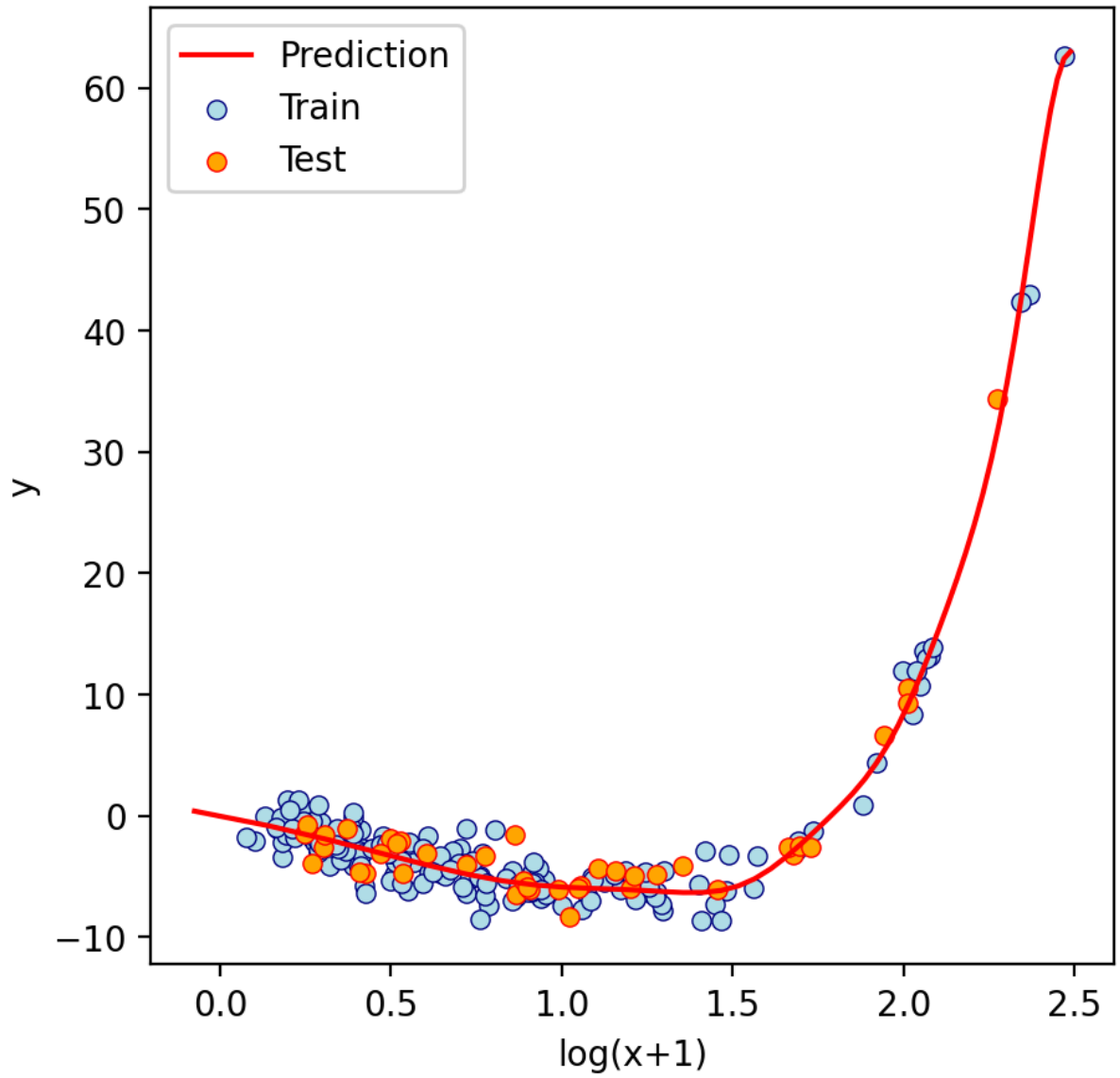
Also, make another plot setting the `log` argument to `True`. This will show the scaling of the x-axis used by the model.

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([("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:", mea

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

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



In [ ]:

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 [159…

```
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,
                 [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
                 [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,
                 [10.4,8,460,215,3,5.424,17.82,0,0,3,4],[14.7,8,440,230,3.23,5.345,17.4
                 [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
                 [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
                 [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
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 [160…

```
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
    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:.1
```

```
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 [161…

```
def get_next_pair(X, y, current_indices):
    # YOUR CODE GOES HERE
    current_indices = np.array(current_indices, dtype=int)
    all_indices = list(range(X.shape[1]))  # Generate all feature indices
    available_indices = [i for i in all_indices if i not in current_indices]  # Excl
    print(available_indices)
    # Generate all combinations of two features from available indices
    pairs = list(itertools.combinations(available_indices, 2))

    best_mse = np.inf
    best_pair = None

    for pair in pairs:
        mse_train, mse_test = get_train_test_mse(X, y, feature_indices=(list(current_i


        if mse_test < best_mse:
            best_mse = 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 [162…

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

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Adding pair (2, 5)
[0, 1, 3, 4, 6, 7, 8, 9]
Adding pair (0, 8)
[1, 3, 4, 6, 7, 9]
Adding pair (6, 7)
[1, 3, 4, 9]
Adding pair (4, 9)
```

Question

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

feature 1 and 3 appears to be the least important.

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 [1]:
```
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,
                 [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
                 [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,
                 [10.4,8,460,215,3,5.424,17.82,0,0,3,4],[14.7,8,440,230,3.23,5.345,17.4
                 [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
                 [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
                 [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
```

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 [13]:
```
def find_redundant_features(data, target_index, threshold):
    # YOUR CODE GOES HERE
    index = []
    cc = np.abs(np.corrcoef(data.T))
    for i in range(cc.shape[0]):
        for j in range(i+1,cc.shape[1]):
            if cc[i,j]>threshold and i != target_index and j != target_index:
                if cc[i,target_index] < cc[j,target_index]:
                    index.append(i)
                else:
```

```
                    index.append(j)
    index = list(set(index))
    return index
```

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 [19]:
```
# YOUR CODE GOES HERE
print(find_redundant_features(data, [0], 0.9))

print(find_redundant_features(data, [2], 0.7))

print(find_redundant_features(data, [10], 0.8))
```

```
[2]
[0, 1, 3, 4, 5, 6, 7, 8, 9, 10]
[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 [20]:
```
# YOUR CODE GOES HERE
print(find_redundant_features(data, [4], 0.9))
print(find_redundant_features(data, [5], 0.8))
print(find_redundant_features(data, [6], 0.85))
```

```
[1]
[0, 1, 2, 3, 7]
[0, 2, 5]
```

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

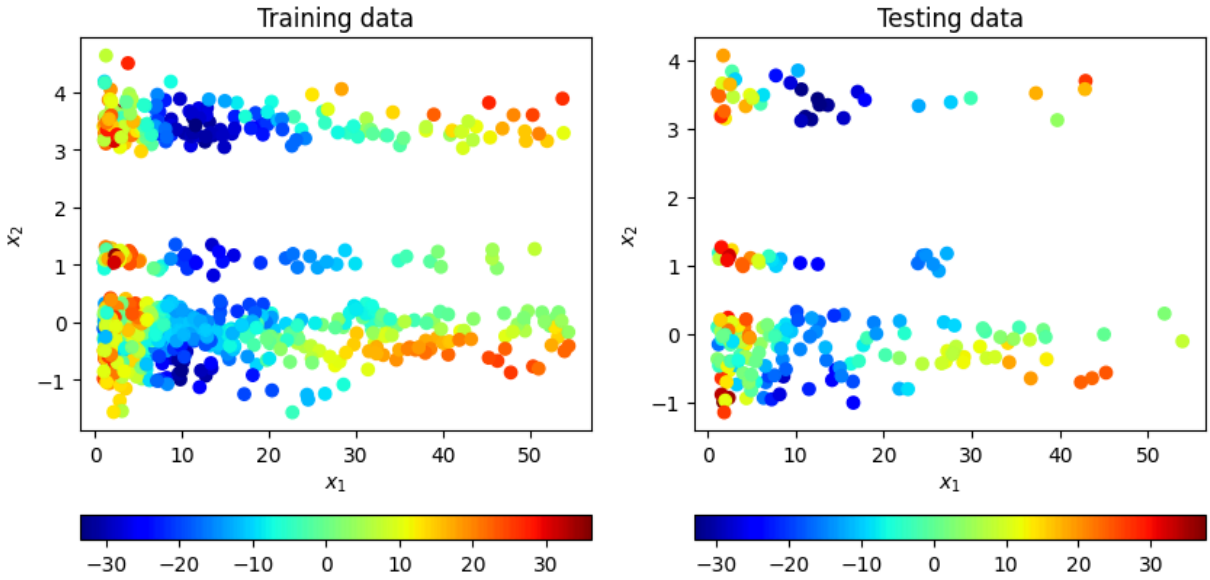
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt


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 [3]:
```
# YOUR CODE GOES HERE
# Define path
path = r"data/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)),r

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 [26]:
```
model_names = ["LSR", "SVR", "KNN", "RF"]

# YOUR CODE GOES HERE
model = Pipeline([('poly',PolynomialFeatures(degree=7)),
                  ('linear',LinearRegression())
```

```
                    ])
model.fit(X_train,y_train)
linear_train = model.predict(X_train)
linear_test = model.predict(X_test)
mse_linear_train = mean_squared_error(y_train,linear_train)
mse_linear_test = mean_squared_error(y_test,linear_test)
print("Linear_train:",mse_linear_train)
print("Linear_test:",mse_linear_test)

model = SVR(C=1000,kernel='rbf')
model.fit(X_train,y_train)
SVR_train = model.predict(X_train)
SVR_test = model.predict(X_test)
mse_SVR_train = mean_squared_error(y_train,SVR_train)
mse_SVR_test = mean_squared_error(y_test,SVR_test)
print("SVR_train:",mse_SVR_train)
print("SVR_test:",mse_SVR_test)

model = KNeighborsRegressor(n_neighbors=4)
model.fit(X_train,y_train)
KNN_train = model.predict(X_train)
KNN_test = model.predict(X_test)
mse_KNN_train = mean_squared_error(y_train,KNN_train)
mse_KNN_test = mean_squared_error(y_test,KNN_test)
print("KNN_train:",mse_KNN_train)
print("KNN_test:",mse_KNN_test)

model = RandomForestRegressor(max_depth=10)
model.fit(X_train,y_train)
rf_train = model.predict(X_train)
rf_test = model.predict(X_test)
mse_rf_train = mean_squared_error(y_train,rf_train)
mse_rf_test = mean_squared_error(y_test,rf_test)
print("rf_train:",mse_rf_train)
print("rf_test:",mse_rf_test)
```

```
Linear_train: 50.86638998029529
Linear_test: 57.2864277523449
SVR_train: 82.04352603565974
SVR_test: 98.63319719407525
KNN_train: 26.856498566141628
KNN_test: 47.63617328402055
rf_train: 5.886004696040535
rf_test: 25.067041685178502
```

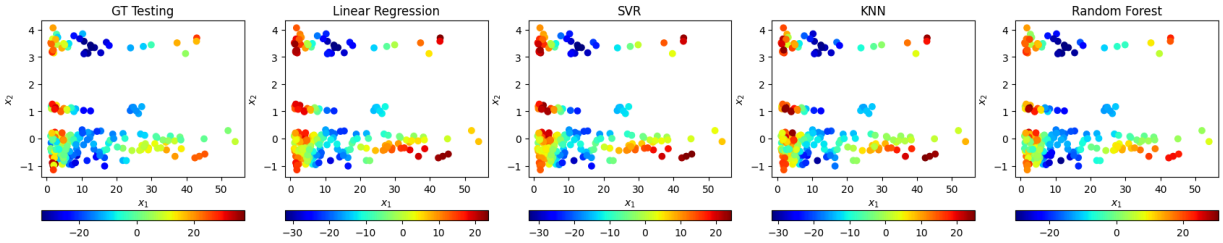
Visualizing the predictions

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 [46]:

```
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, linear_test, "Linear Regression")
plt.title("Linear Regression")

plt.subplot(1, 5, 3)
plot(X_test, SVR_test,"SVR")
plt.title("SVR")
```

```
plt.subplot(1, 5, 4)
plot(X_test, KNN_test, "KNN")
plt.title("KNN")
plt.subplot(1, 5, 5)
plot(X_test, rf_test,"Random Forest")
plt.title("Random Forest")

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 [44]:

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

# YOUR CODE GOES HERE

model = Pipeline([('scaler',QuantileTransformer(n_quantiles=800)),
                  ('poly',PolynomialFeatures(degree=7)),
                  ('linear',LinearRegression())
                  ])
model.fit(X_train,y_train)
linear_train = model.predict(X_train)
linear_test = model.predict(X_test)
mse_linear_train = mean_squared_error(y_train,linear_train)
mse_linear_test = mean_squared_error(y_test,linear_test)
print("Linear_train:",mse_linear_train)
print("Linear_test:",mse_linear_test)

model = Pipeline([('scaler',QuantileTransformer(n_quantiles=800)),
                  ('SVR',SVR(C=1000,kernel='rbf'))
                  ])
model.fit(X_train,y_train)
SVR_train = model.predict(X_train)
SVR_test = model.predict(X_test)
mse_SVR_train = mean_squared_error(y_train,SVR_train)
mse_SVR_test = mean_squared_error(y_test,SVR_test)
print("SVR_train:",mse_SVR_train)
print("SVR_test:",mse_SVR_test)

model = Pipeline([('scaler',QuantileTransformer(n_quantiles=800)),
                  ('KNN',KNeighborsRegressor(n_neighbors=4))
                  ])
model.fit(X_train,y_train)
KNN_train = model.predict(X_train)
KNN_test = model.predict(X_test)
```

```
mse_KNN_train = mean_squared_error(y_train,KNN_train)
mse_KNN_test = mean_squared_error(y_test,KNN_test)
print("KNN_train:",mse_KNN_train)
print("KNN_test:",mse_KNN_test)

model = Pipeline([('scaler',QuantileTransformer(n_quantiles=800)),
                  ('rf',RandomForestRegressor(max_depth=10))
                  ])
model.fit(X_train,y_train)
rf_train = model.predict(X_train)
rf_test = model.predict(X_test)
mse_rf_train = mean_squared_error(y_train,rf_train)
mse_rf_test = mean_squared_error(y_test,rf_test)
print("rf_train:",mse_rf_train)
print("rf_test:",mse_rf_test)
```

```
Linear_train: 39.52893428670628
Linear_test: 43.2036349225095
SVR_train: 41.03425800595977
SVR_test: 43.017915737897745
KNN_train: 19.687691313922564
KNN_test: 36.397038931930005
rf_train: 5.929603361123817
rf_test: 24.31405312970533
```

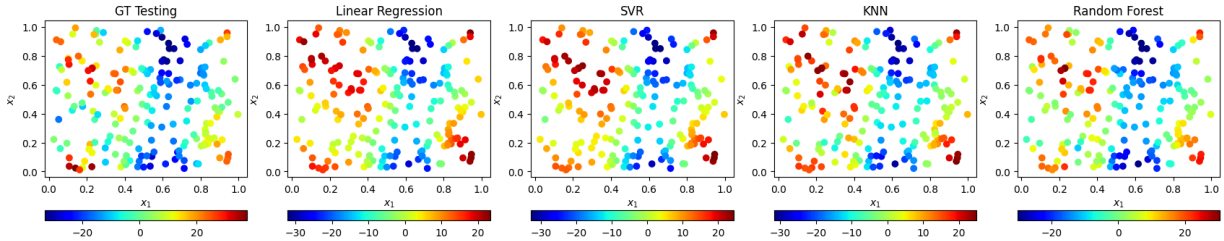
Visualization with scaled input

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 [52]:

```
# YOUR CODE GOES HERE
plt.figure(figsize=(21,4))
X_test = model.named_steps['scaler'].transform(X_test)
plt.subplot(1,5,1)
plot(X_test, y_test, "GT Testing")

# YOUR CODE GOES HERE
plt.subplot(1, 5, 2)
plot(X_test, linear_test, "Linear Regression")
plt.title("Linear Regression")

plt.subplot(1, 5, 3)
plot(X_test, SVR_test,"SVR")
plt.title("SVR")

plt.subplot(1, 5, 4)
plot(X_test, KNN_test, "KNN")
plt.title("KNN")
plt.subplot(1, 5, 5)
plot(X_test, rf_test,"Random Forest")
plt.title("Random Forest")

plt.show()
```



Questions

1. Without transforming the input data, which model performed the best on test data? What about after scaling?

1. For each method, say whether scaling the input improved or worsened, how extreme the change was, and why you think this is.

1. Random Forest performs the best as it is the closest to the ground truth
2.

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 [60]:
```
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], s
    dataset_test = dict(coordinates=coordinates[split:], features=features[split:], st
    X_train, X_test = np.concatenate(features[:split], axis=0), np.concatenate(feature
    y_train, y_test = np.concatenate(stress[:split], axis=0), np.concatenate(stress[sp
    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()
```

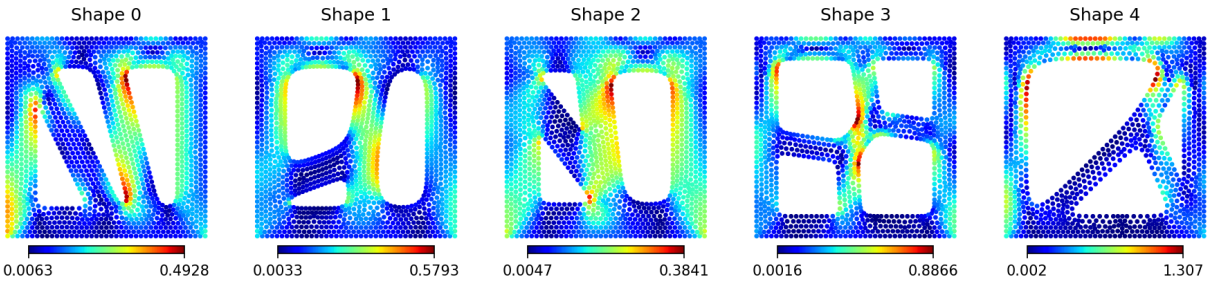
Loading the data

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 [61]:
```
# 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 [62]:
```
# YOUR CODE GOES HERE
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)
linear_train = linear_model.predict(X_train)
linear_test= linear_model.predict(X_test)
mse_linear_train = mean_squared_error(y_train, linear_train)
mse_linear_test = mean_squared_error(y_test, linear_test)
print("Linear_train:", mse_linear_train)
print("Linear_test:", mse_linear_test)

dt_model = DecisionTreeRegressor(max_depth=20)
dt_model.fit(X_train, y_train)
dt_model.predict(X_train)
dt_train = dt_model.predict(X_train)
dt_test= dt_model.predict(X_test)
mse_dt_train = mean_squared_error(y_train, dt_train)
mse_dt_test = mean_squared_error(y_test, dt_test)
print("df_train:", mse_dt_train)
print("df_test:", mse_dt_test)
```

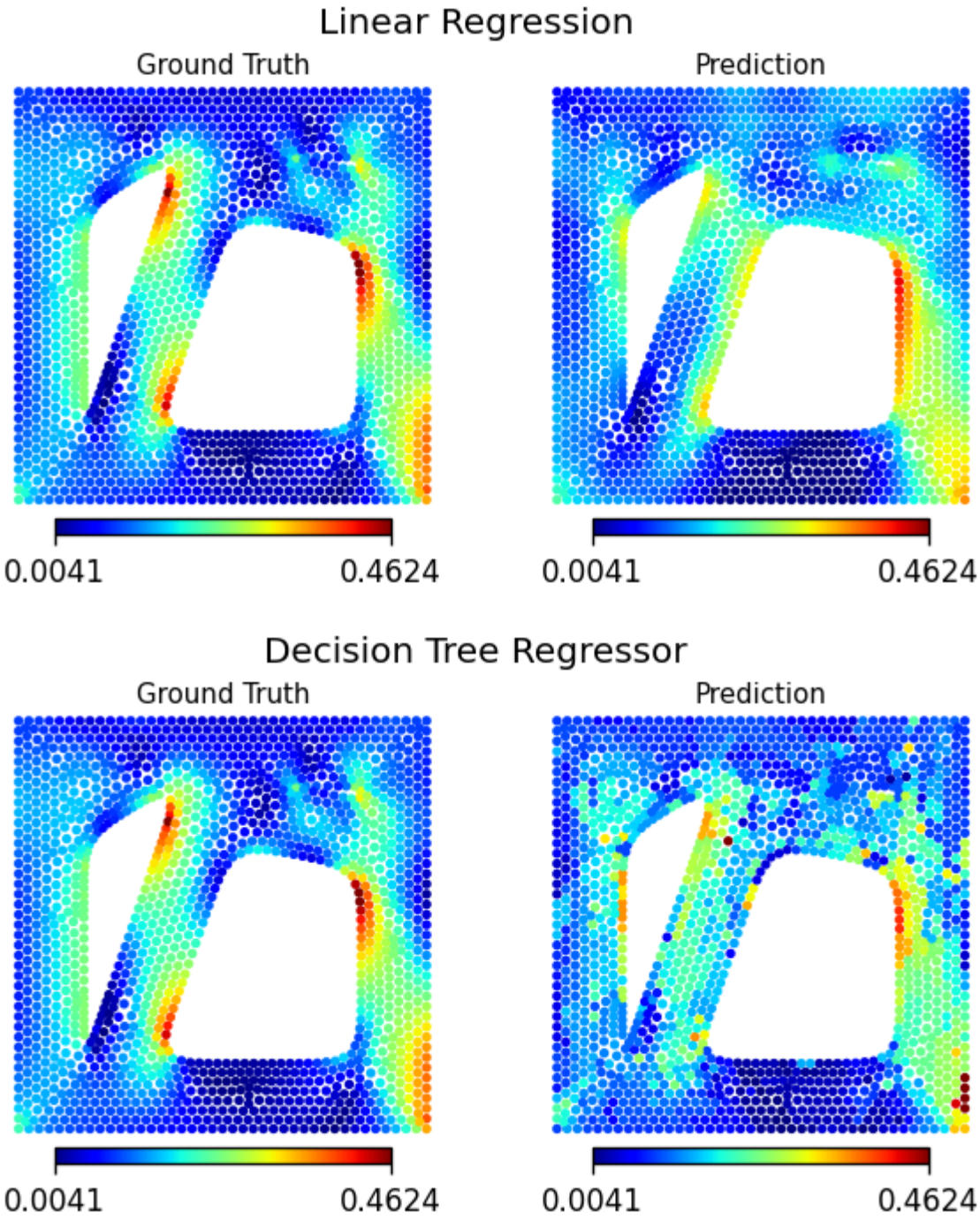
```
Linear_train: 0.008110601
Linear_test: 0.009779482
df_train: 0.0004944875978805109
df_test: 0.008283868036732832
```

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 [63]:
```
test_idx = 8

# YOUR CODE GOES HERE

plot_shape_comparison(dataset_test, test_idx, linear_model, title = "Linear Regression")
plot_shape_comparison(dataset_test, test_idx, dt_model, title = "Decision Tree Regressor"
```

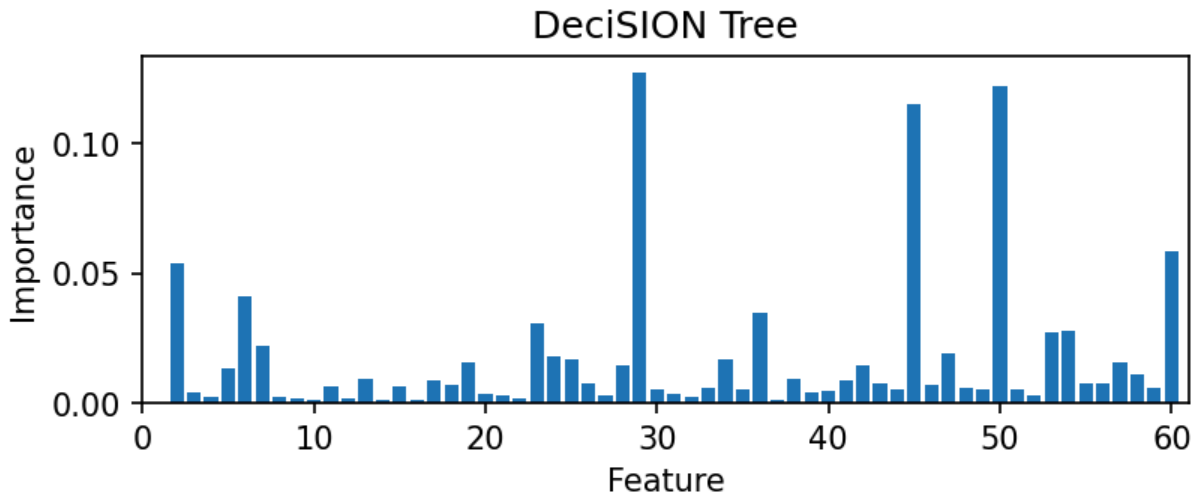


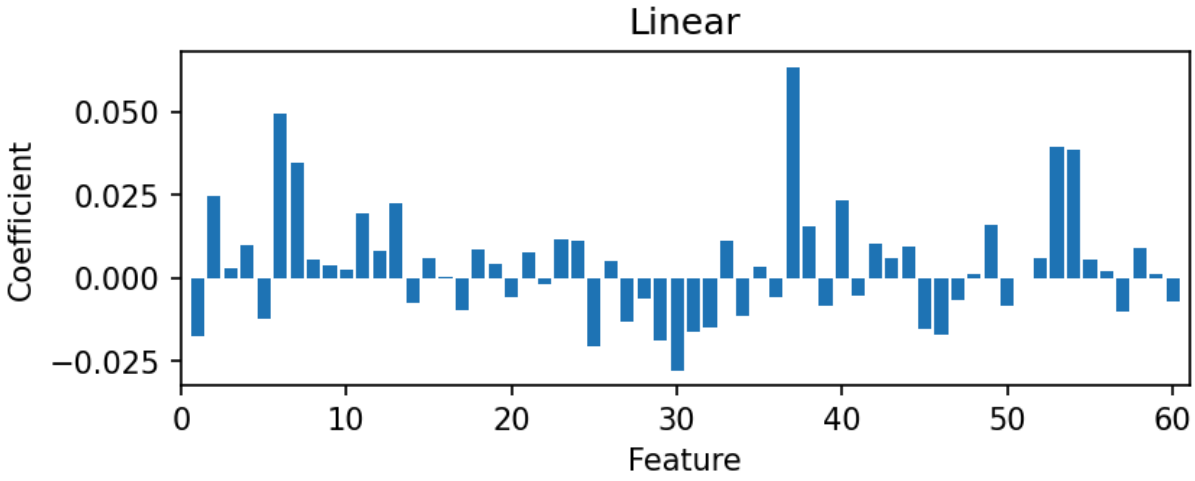
Feature importance

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 [64]:
```
# YOUR CODE GOES HERE


plot_importances(dt_model, selected = None, coef=False, title="DeciSION Tree")
plot_importances(linear_model, selected = None, coef=True, title="Linear")
```





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 [65]:
```
# YOUR CODE GOES HERE
rfe_linear = RFE(LinearRegression(),n_features_to_select=30)
rfe_linear.fit(X_train,y_train)
rfe_linear_train = rfe_linear.predict(X_train)
rfe_linear_test = rfe_linear.predict(X_test)
rfe_mse_linear_train = mean_squared_error(y_train,rfe_linear_train)
rfe_mse_linear_test = mean_squared_error(y_test,rfe_linear_test)
print("Linear_train:",rfe_mse_linear_train)
print("Linear_test:",rfe_mse_linear_test)

rfe_dt = RFE(DecisionTreeRegressor(max_depth=20),n_features_to_select=30)
```

```
rfe_dt.fit(X_train,y_train)
rfe_dt_train = rfe_dt.predict(X_train)
rfe_dt_test = rfe_dt.predict(X_test)
rfe_mse_dt_train = mean_squared_error(y_train,rfe_dt_train)
rfe_mse_dt_test = mean_squared_error(y_test,rfe_dt_test)
print("rf_train:",rfe_mse_dt_train)
print("rf_test:",rfe_mse_dt_test)
```

```
Linear_train: 0.008508718
Linear_test: 0.010150376
rf_train: 0.0005586730433575364
rf_test: 0.009001621507497446
```

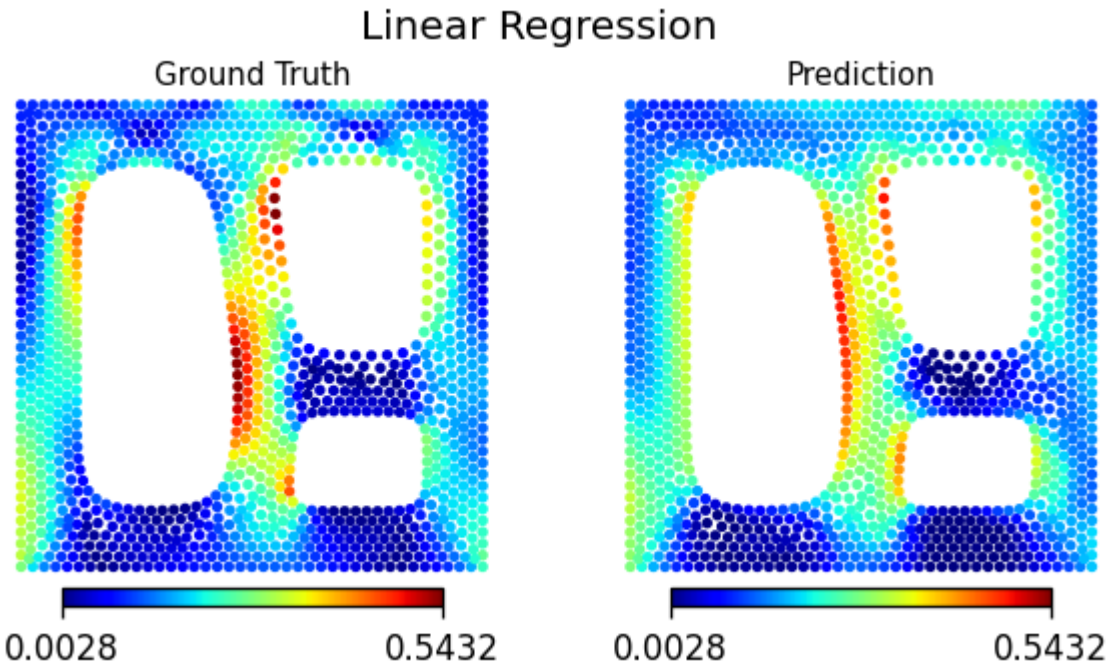
Visualization

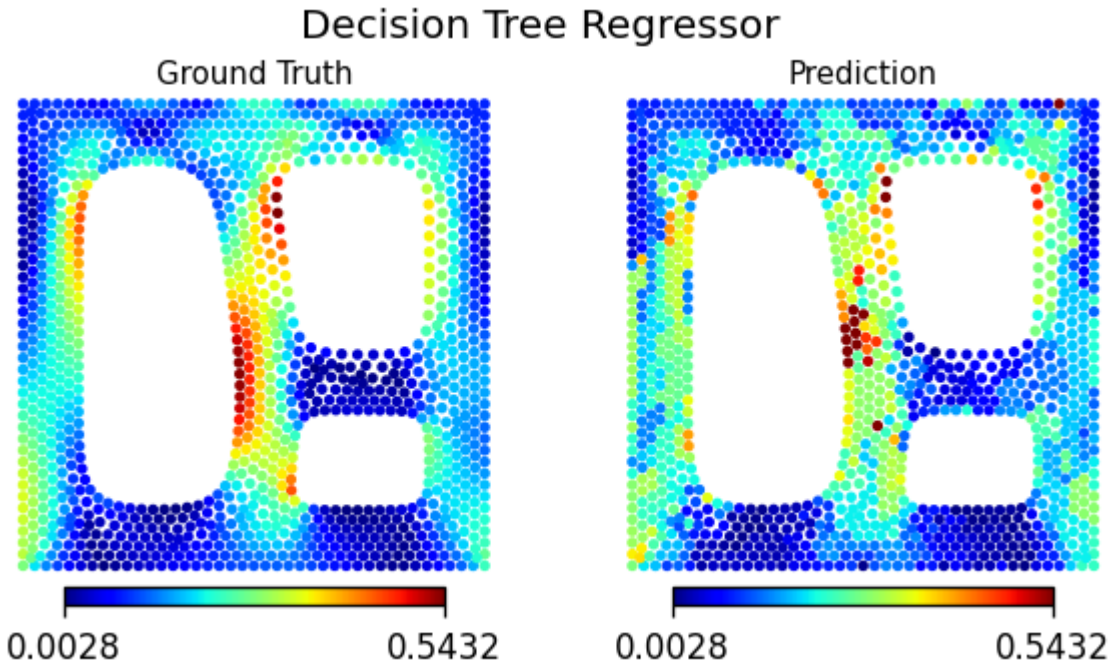
Use the `plot_shape_comparison()` function to plot the index 16 shape results in `dataset_test` for each model.

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

In [66]:

```
test_idx = 16

plot_shape_comparison(dataset_test,test_idx,rfe_linear,title = "Linear Regression")
plot_shape_comparison(dataset_test,test_idx,rfe_dt,title = "Decision Tree Regressor")
```



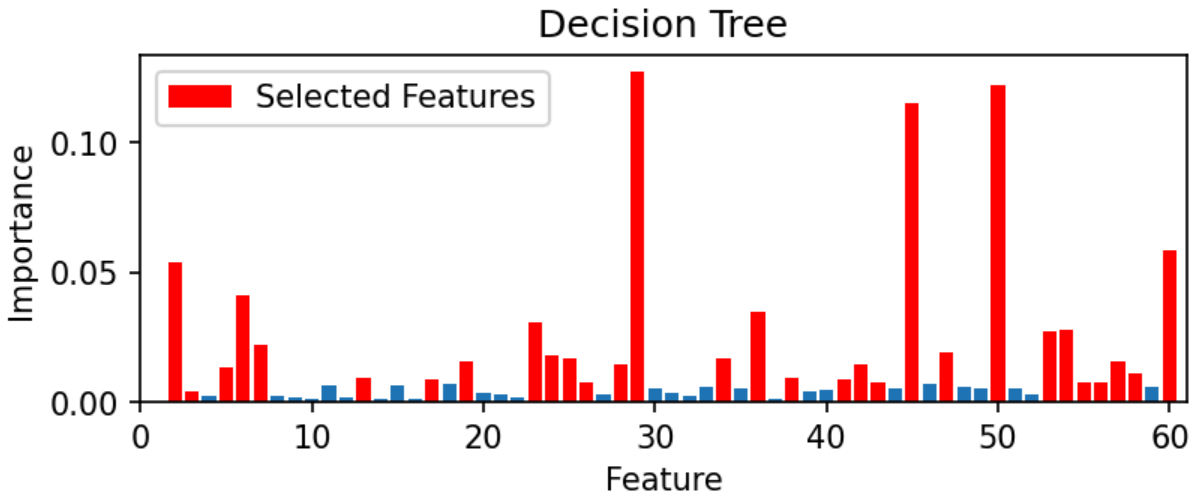


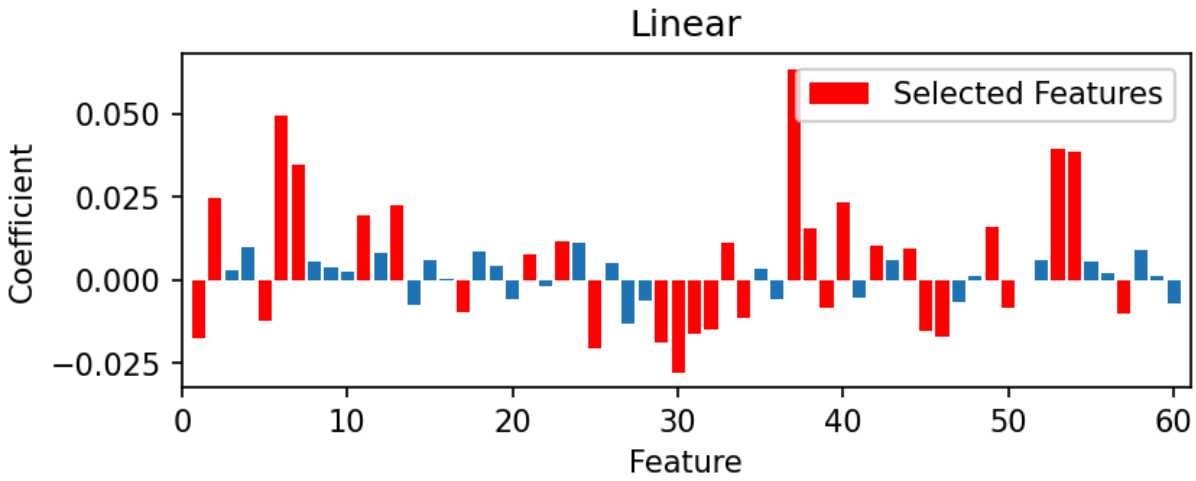
Feature importance with RFE

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 [67]:
```
# YOUR CODE GOES HERE
dt_select = rfe_dt.get_support(indices = True)
linear_select = rfe_linear.get_support(indices = True)
plot_importances(dt_model, selected = dt_select,coef = False, title="Decision Tree")
plot_importances(linear_model, selected = linear_select,coef = True,title="Linear")
```





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).

1. The MSE decrease after performing RFE
2. The MSE decrease after performing RFE
3. MSE of Decision tree model is always smaller than that of lienar regression model
4. The feature that were selected have much higher coefficient than non-selected ones.
5. The feature that were selected have relatively higher coefficient than non-selected ones.