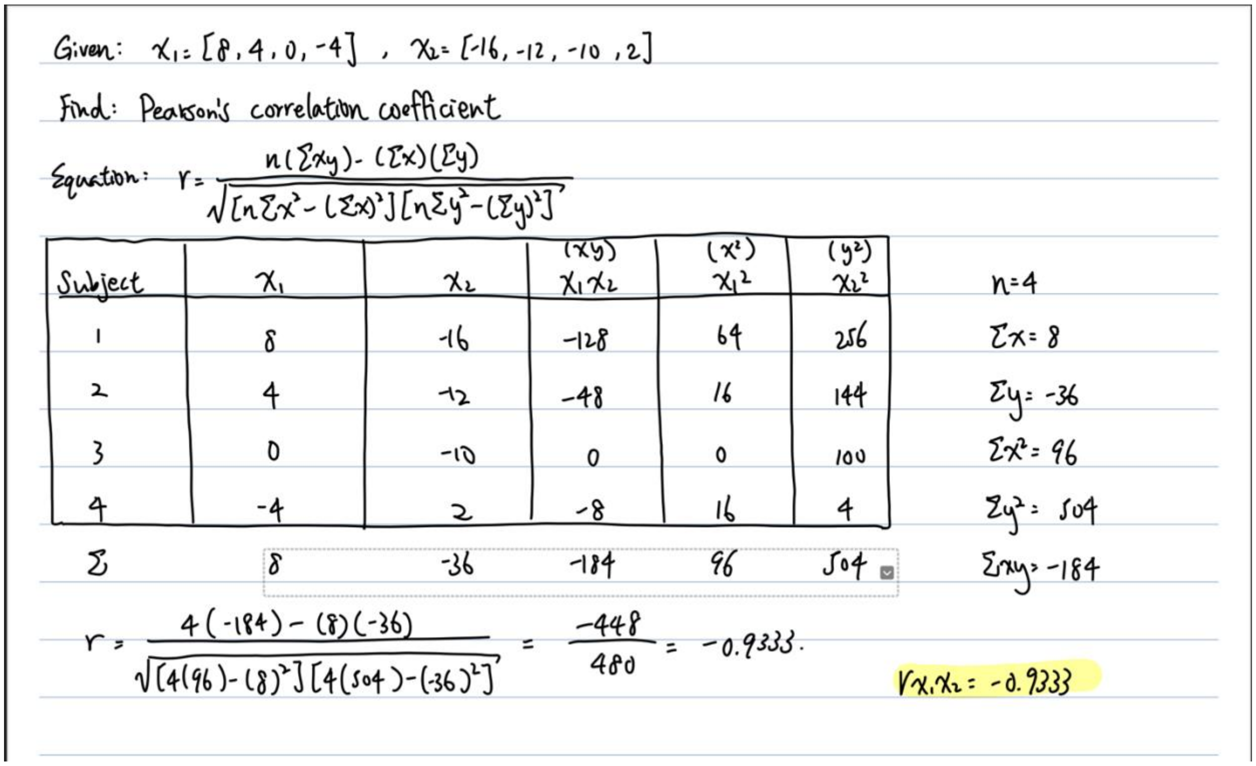
24-787: Machine Learning and Artificial Intelligence for Engineers
Ryan Wu
ID: weihuanw
Homework 6
Due: Mar 2 2024

Concept Questions:

Problem 1
B) Standardized and C) Normalized.

Problem 2
Log Transformation.

Problem 3

Given: $x_1 = [8, 4, 0, -4]$, $x_2 = [-16, -12, -10, 2]$

Find: Pearson's correlation coefficient

Equation: $r = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{\sqrt{[n\Sigma x^2 - (\Sigma x)^2][n\Sigma y^2 - (\Sigma y)^2]}}$

Subject	x_1	x_2	(xy) $x_1 x_2$	(x^2) x_1^2	(y^2) x_2^2
1	8	-16	-128	64	256
2	4	-12	-48	16	144
3	0	-10	0	0	100
4	-4	2	-8	16	4
Σ	8	-36	-184	96	504

$n = 4$
$\Sigma x = 8$
$\Sigma y = -36$
$\Sigma x^2 = 96$
$\Sigma y^2 = 504$
$\Sigma xy = -184$

$$r = \frac{4(-184) - (8)(-36)}{\sqrt{[4(96) - (8)^2][4(504) - (-36)^2]}} = \frac{-448}{480} = -0.9333.$$

$r_{x_1 x_2} = -0.9333$

Problem 4
X_2.

Problem 1 (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

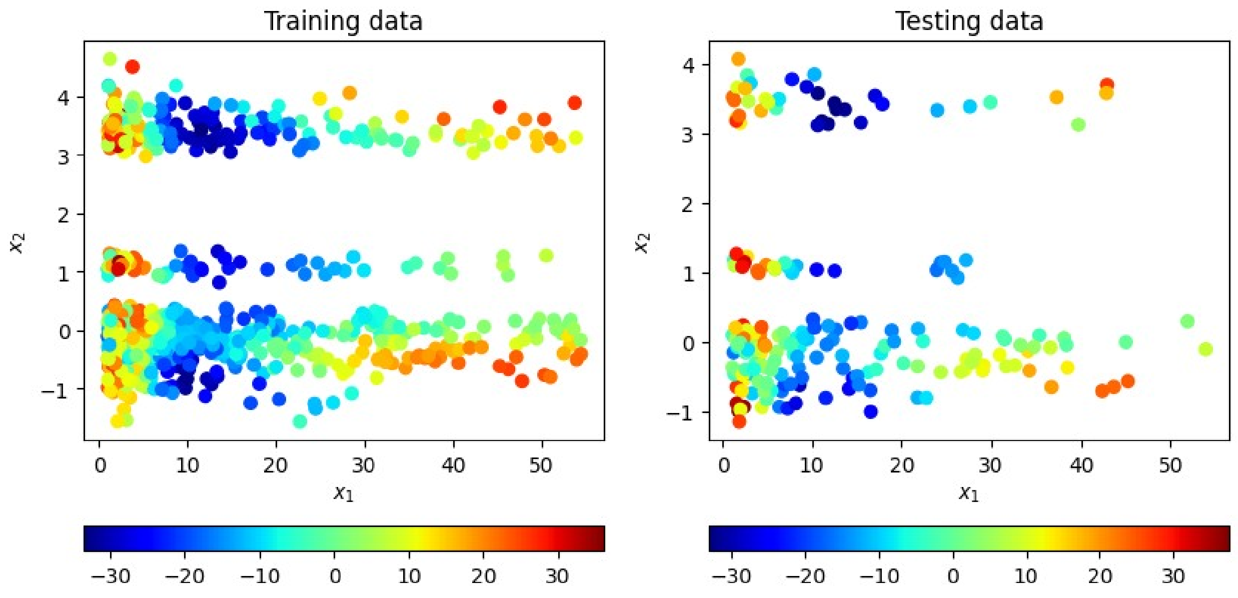
```
import numpy as np
import matplotlib.pyplot as plt


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`.

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

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



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

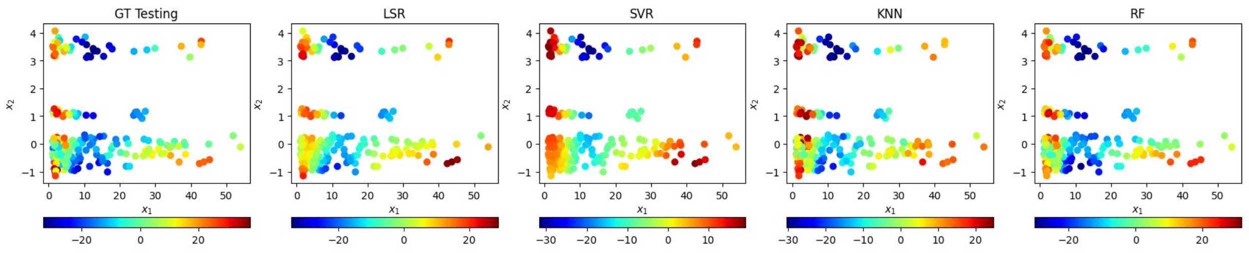
```
model_names = ["LSR", "SVR", "KNN", "RF"]

# YOUR CODE GOES HERE
# linear regression model with polynomial features of degree 7
LSR = Pipeline([("poly", PolynomialFeatures(degree=7)), ("reg",
LinearRegression())])
# SVR model with rbf kernel and c = 1000
SVR = SVR(kernel="rbf", C=1000)
# KNN model with 4 neighbors
KNN = KNeighborsRegressor(n_neighbors=4)
# Random forest model with 100 estimator and max depth of 10
RF = RandomForestRegressor(n_estimators=100, max_depth=10,
random_state=0)

# train the models and print train & test MSE
models = [LSR, SVR, KNN, RF]
for model in models:
    model.fit(X_train, y_train)
    train_pred = model.predict(X_train)
    test_pred = model.predict(X_test)
    print(f"{model_names[models.index(model)]} Train MSE:
{mean_squared_error(y_train, train_pred)}")
    print(f"{model_names[models.index(model)]} Test MSE:
{mean_squared_error(y_test, test_pred)}")
    print()
```

```
LSR Train MSE: 50.8663899566442
LSR Test MSE: 57.28650448965208

SVR Train MSE: 82.04352603565977
SVR Test MSE: 98.63319719407623

KNN Train MSE: 26.856498566141628
KNN Test MSE: 47.63617328402055

RF Train MSE: 6.045950294198362
RF Test MSE: 25.252061900335093
```

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.

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

# YOUR CODE GOES HERE
for model in models:
    plt.subplot(1,5,models.index(model)+2)
    plot(X_test, model.predict(X_test),
model_names[models.index(model)])

plt.show()
```



Quantile Scaling

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

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

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

# YOUR CODE GOES HERE
# quantile scaler with n_quantiles = 800
quantile_scaler = QuantileTransformer(n_quantiles=800)
# pipeline for 4 models used earlier
pipelines = [Pipeline([("quantile", quantile_scaler), ("LSR", LSR)]),
Pipeline([("quantile", quantile_scaler), ("SVR", SVR)]),
Pipeline([("quantile", quantile_scaler), ("KNN", KNN)]),
Pipeline([("quantile", quantile_scaler), ("RF", RF)])]

# train the models and print train & test MSE
for pipeline in pipelines:
    pipeline.fit(X_train, y_train)
    train_pred = pipeline.predict(X_train)
    test_pred = pipeline.predict(X_test)
    print(f"{pipeline_names[pipelines.index(pipeline)]} Train MSE: 
{mean_squared_error(y_train, train_pred)}")
    print(f"{pipeline_names[pipelines.index(pipeline)]} Test MSE: 
{mean_squared_error(y_test, test_pred)}")
    print()
```

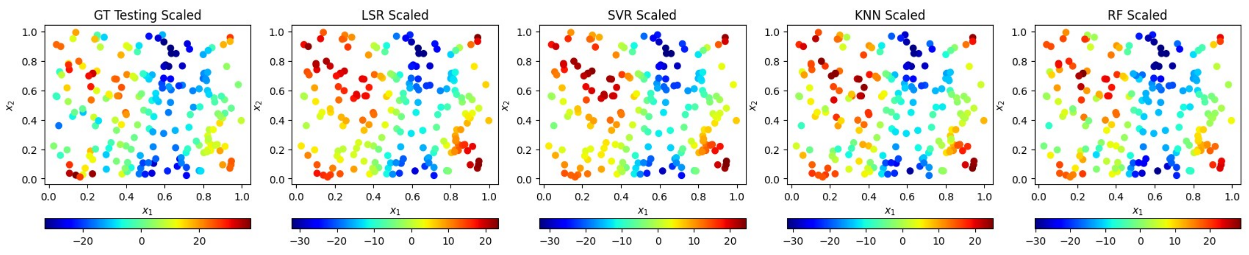
```
LSR Scaled Train MSE: 39.52893428670415
LSR Scaled Test MSE: 43.20363492251461

SVR Scaled Train MSE: 41.03425800596019
SVR Scaled Test MSE: 43.01791573789873

KNN Scaled Train MSE: 19.687691313922564
KNN Scaled Test MSE: 36.397038931930005

RF Scaled Train MSE: 6.077215326436146
RF Scaled Test MSE: 25.190003357399775
```

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.

```
# YOUR CODE GOES HERE
# scale the test data
X_test_scaled = quantile_scaler.transform(X_test)
plt.figure(figsize=(21,4))
plt.subplot(1,5,1)
plot(X_test_scaled, y_test, "GT Testing Scaled")

# plot the predictions
for pipeline in pipelines:
    plt.subplot(1,5,pipelines.index(pipeline)+2)
    plot(X_test_scaled, pipeline.predict(X_test), 
pipeline_names[pipelines.index(pipeline)])

plt.show()
```



Questions

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

   Without transforming the input data, the random forest regressor model performed the best on given test data.

   After scaling the input data, the random forest regressor model also performed the best on given test data.

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

   For the linear regressor model, scaling the input improved the model's performance slightly. This is because linear regressors are sensitive to the scale of the features and helps

   For the support vector regressor model, scaling the input improved the model's performance significantly. This is because support vector regressors rely on the distance between points, thus scaling the features can lead to great improvements in performance.

   For the K-nearest neighbor regressor model, scaling the input improved the model's performance slightly. This is because K-nearest neighbors compute the distance between points, and scaling the features can impact the performance.

   For the random forest regressor model, scaling the input had little to no change in the model's performance. This is because the random forest regressor relies on averaging the decision trees, which makes it less sensitive to the scale of the features.

Problem 2 (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:

```python
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

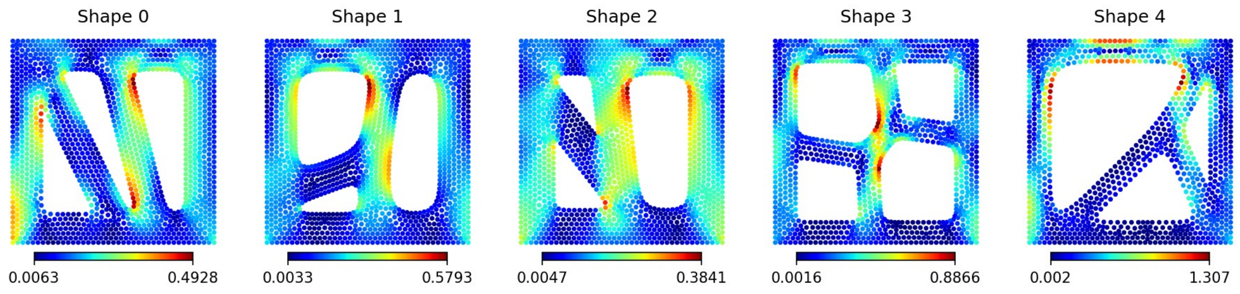
First, complete the code below to load the data and plot the von Mises stress fields for a few shapes.
You'll need to input the path of the data file, the rest is done for you.

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.

```
# YOUR CODE GOES HERE
# Define data_path

dataset_train, dataset_test, X_train, X_test, y_train, y_test =
load_dataset("data/stress_nodal_features.npy")
```

```
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.

```
# YOUR CODE GOES HERE
model_names = ["Linear Regression", "Decision Tree Regressor"]
# linear regression model
LSR = LinearRegression()
# decision tree regressor model with max depth 20
DTR = DecisionTreeRegressor(max_depth=20, random_state=0)

# train the models and print train & test MSE
models = [LSR, DTR]
for model in models:
    model.fit(X_train, y_train)
    train_pred = model.predict(X_train)
    test_pred = model.predict(X_test)
    print(f"{model_names[models.index(model)]} Train MSE:
{mean_squared_error(y_train, train_pred)}")
    print(f"{model_names[models.index(model)]} Test MSE:
{mean_squared_error(y_test, test_pred)}")
    print()
```

```
Linear Regression Train MSE: 0.008110600523650646
Linear Regression Test MSE: 0.009779523126780987
```

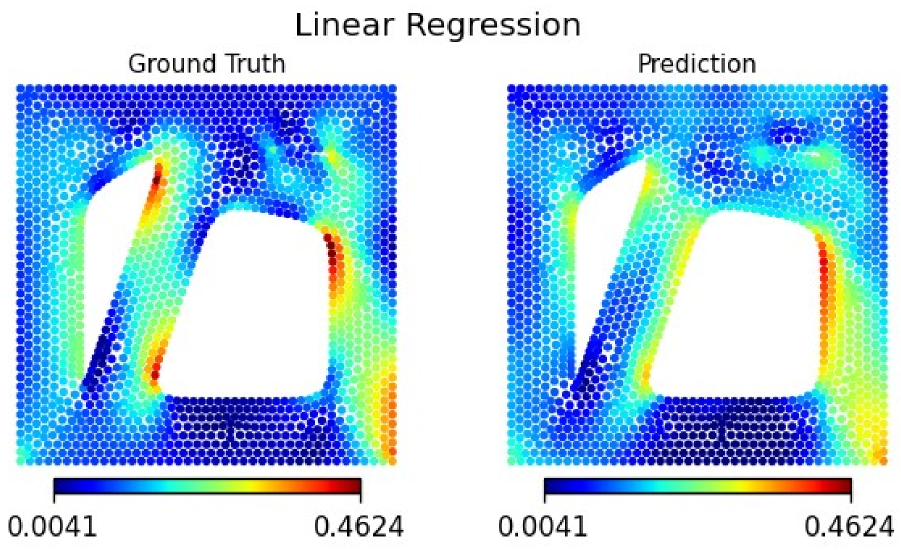
```
Decision Tree Regressor Train MSE: 0.00049448759788805109
Decision Tree Regressor Test MSE: 0.00838549943577
```

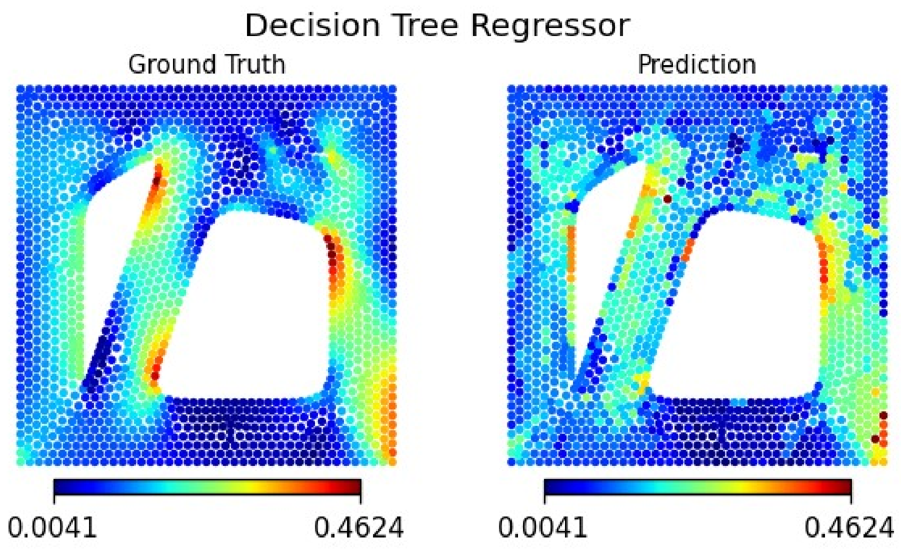
Visualization

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.

```
test_idx = 8

# YOUR CODE GOES HERE
for model in models:
    plot_shape_comparison(dataset_test, test_idx, model,
model_names[models.index(model)])
```



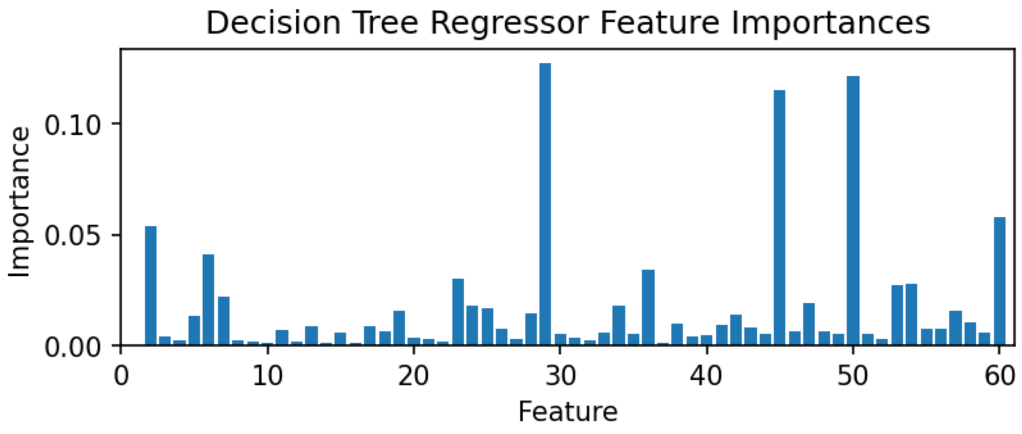


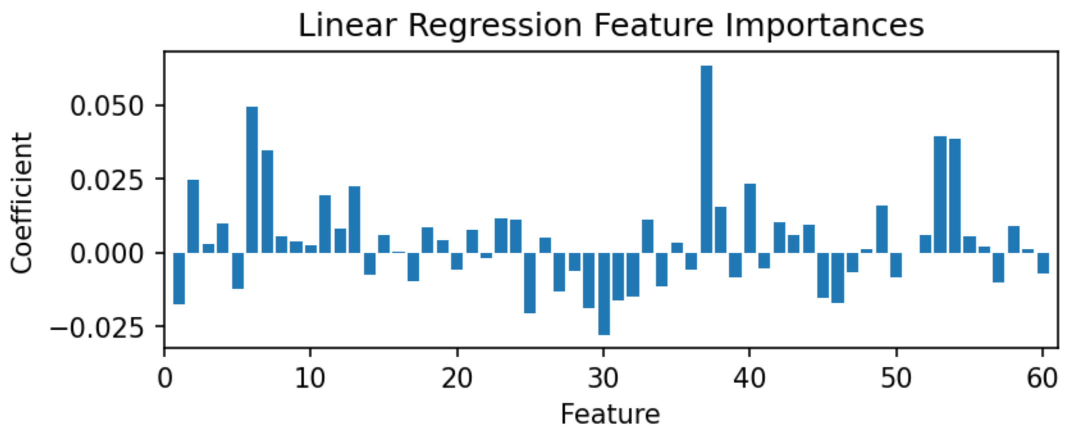
Feature importance

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`.

```
# YOUR CODE GOES HERE
# plot the feature importances for the decision tree regressor
plot_importances(DTR, title="Decision Tree Regressor Feature
Importances")
# plot the feature importances for the linear regression model
plot_importances(LSR, coef=True, title="Linear Regression Feature
Importances")
```





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.

```
# YOUR CODE GOES HERE
# RFE with n_features_to_select=30

model_names = ["Linear Regression RFE", "Decision Tree Regressor RFE"]
# RFE with n_features_to_select=30
LSR_RFE = RFE(LSR, n_features_to_select=30)
DTR_RFE = RFE(DTR, n_features_to_select=30)

# train the RFE models and print train & test MSE
models = [LSR_RFE, DTR_RFE]
for model in models:
    model.fit(X_train, y_train)
    train_pred = model.predict(X_train)
    test_pred = model.predict(X_test)
    print(f"{model_names[models.index(model)]} Train MSE:
{mean_squared_error(y_train, train_pred)}")
    print(f"{model_names[models.index(model)]} Test MSE:
{mean_squared_error(y_test, test_pred)}")
    print()
```

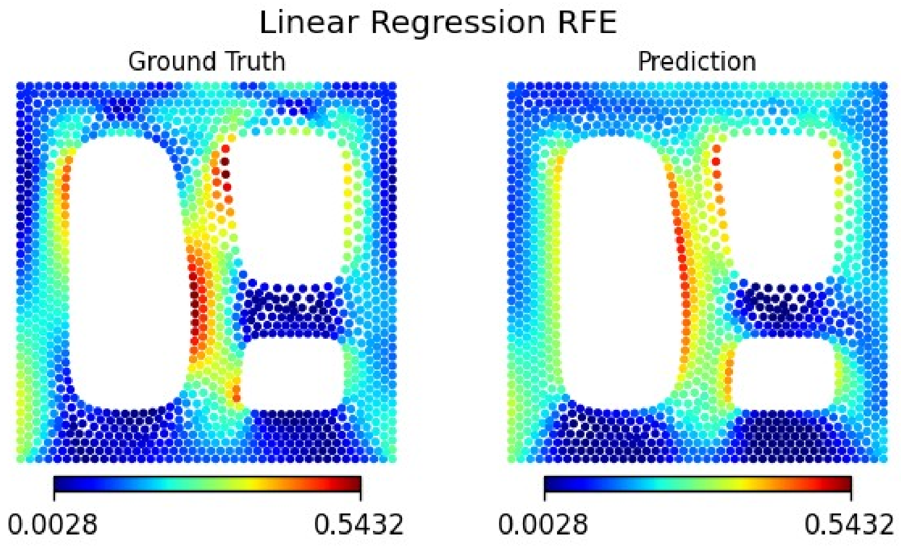
```
Linear Regression RFE Train MSE: 0.008508718572556973
Linear Regression RFE Test MSE: 0.010150418616831303

Decision Tree Regressor RFE Train MSE: 0.0005351821126843501
Decision Tree Regressor RFE Test MSE: 0.009041167760149434
```

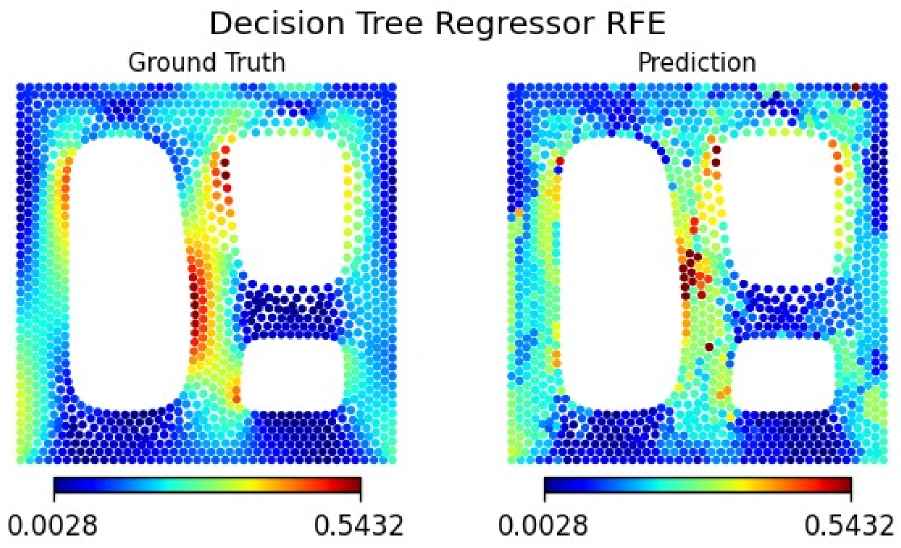
Visualization

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.

```
test_idx = 16

# YOUR CODE GOES HERE
for model in models:
    plot_shape_comparison(dataset_test, test_idx, model, 
model_names[models.index(model)])
```



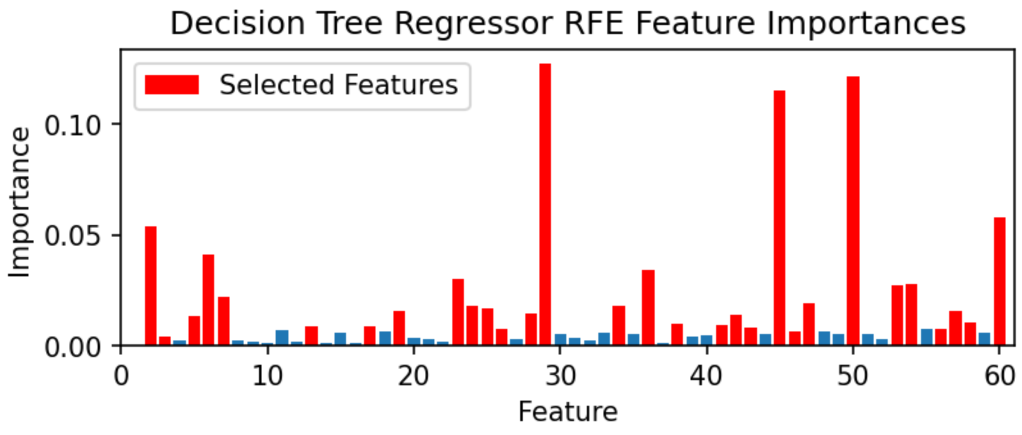


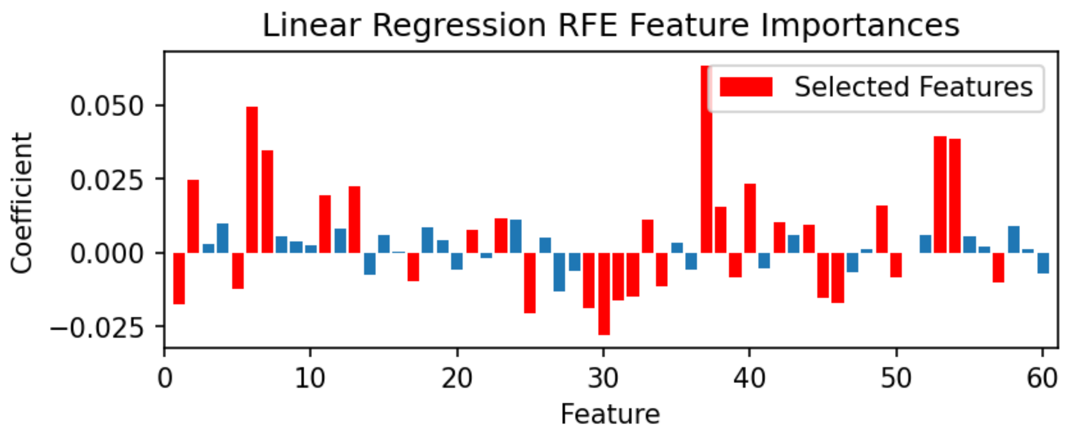
Feature importance with RFE

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)`.

```
# YOUR CODE GOES HERE
# plot the feature importances for the decision tree regressor RFE
plot_importances(DTR, selected=DTR_RFE.get_support(indices=True),
title="Decision Tree Regressor RFE Feature Importances")
# plot the feature importances for the linear regression model RFE
plot_importances(LSR, selected=LSR_RFE.get_support(indices=True),
coef=True, title="Linear Regression RFE Feature Importances")
```





Questions

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

   The MSE increased on test data for the linear regression model after performing RFE.

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

   The MSE increased on test data for the decision tree model after performing RFE.

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

   The decision tree has a lower MSE compared to the linear regression model. For the linear regression model, the predicted nodal von Mises stress has a smoother mesh representation compared to the decision tree model.

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

   In the decision tree, the coefficients that were selected by RFE are of higher importance and influence on the model outcome compared to the features that were eliminated. From the above figure, we can observe that the selected features (red) have a higher importance score compared to the eliminated features (blue) that have a lower importance score.

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

   In linear regression, the coefficients that were selected (red) by RFE are expected to be larger positive or negative coefficients that are non-zero, which has the most influence on the model outcome. The features that are eliminated (blue) are features that are close to zero or zero, which have little to no influence on the model outcome.

M6-L1 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:

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

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

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

Implementing MinMax Scaling

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

Validate your results by comparing to the above.

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


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

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

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

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

Standard Scaling

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

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

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

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

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

Implementing Standard Scaling

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

Validate your results by comparing to the above.

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

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

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

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

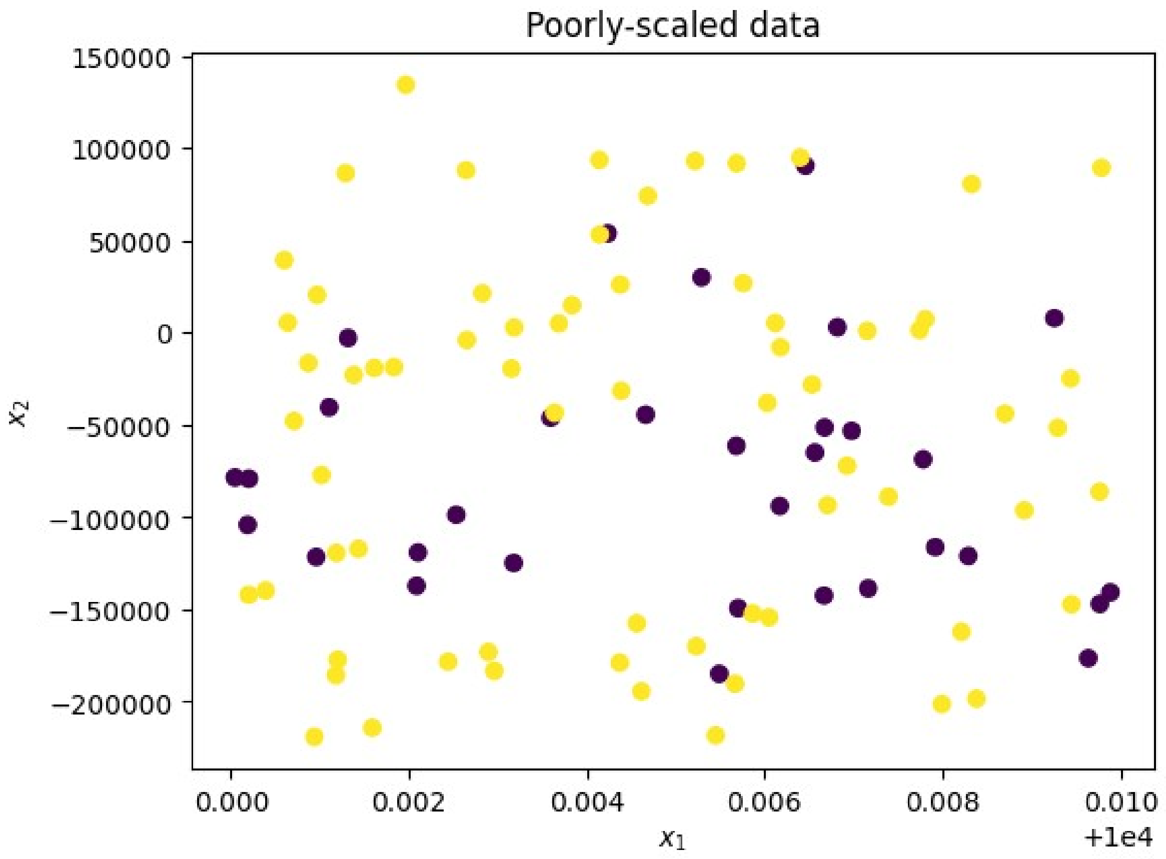
```
X =
 [[10  2]
 [ 9  0]
 [ 3  8]
 [ 4  9]
 [ 0  8]
 [-3  2]
 [-4 -2]
 [ 5  6]]  -->
 [[ 1.46  -0.547]
 [ 1.251 -1.061]
 [ 0.     0.997]
 [ 0.209  1.254]
 [-0.626  0.997]
 [-1.251 -0.547]
 [-1.46  -1.576]
 [ 0.417  0.482]]
A =
 [[ 2  1]
 [ 5  5]
 [11 -1]
 [-3  2]]  -->
 [[-0.209 -0.804]
 [ 0.417  0.225]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

M6-L1 Problem 2 (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.

```
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, 10000.00437587,
10000.00891773, 10000.00963663, 10000.00383442, 10000.00791725,
10000.00528895, 10000.00568045, 10000.00925597, 10000.00071036,
10000.00087129, 10000.00020218, 10000.0083262 , 10000.00778157,
10000.00870012, 10000.00978618, 10000.00799159, 10000.00461479,
10000.00780529, 10000.00118274, 10000.00639921, 10000.00143353,
10000.00944669, 10000.00521848, 10000.00414662, 10000.00264556,
10000.00774234, 10000.0045615 , 10000.00568434, 10000.0001879 ,
10000.00617635, 10000.00612096, 10000.00616934, 10000.00943748,
10000.0068182 , 10000.00359508, 10000.00437032, 10000.00697631,
10000.00060225, 10000.00666767, 10000.00670638, 10000.00210383,
10000.00128926, 10000.00315428, 10000.00363711, 10000.00570197,
10000.00438602, 10000.00988374, 10000.00102045, 10000.00208877,
10000.0016131 , 10000.00653108, 10000.00253292, 10000.00466311,
10000.00244426, 10000.0015897 , 10000.00110375, 10000.0065633 ,
10000.00138183, 10000.00196582, 10000.00368725, 10000.00820993,
10000.00097101, 10000.00837945, 10000.00096098, 10000.00976459,
10000.00468651, 10000.00976761, 10000.00604846, 10000.00739264,
10000.00039188, 10000.00282807, 10000.00120197, 10000.0029614 ,
10000.00118728, 10000.00317983, 10000.00414263, 10000.00064147,
10000.00692472, 10000.00566601, 10000.00265389, 10000.00523248,
10000.00093941, 10000.00575946, 10000.00929296, 10000.00318569,
10000.0066741 , 10000.00131798, 10000.00716327, 10000.00289406,
10000.00183191, 10000.00586513, 10000.00020108, 10000.0082894 ,
10000.00004695])
x2 = np.array([-184863.4856705 ,    1074.38382588,  -38090.38042426, -
218261.93176495,   53942.6974416 ,   90630.02584275, 26090.16140437,
-96193.23522311, -176367.73593595, 14900.6554238 , -116285.92522759,
30020.05633442, -61255.25197308,    7897.51328353,  -47927.0242543 , -
16408.41486272,  -79054.99813513,   80728.34445153, -68577.91165667,
-43820.95728998,   89483.56273506, -201298.31550282, -194343.64986372,
7245.70373422, -185581.10646027,   94925.90670844, -117225.70826838, -
147270.93302967,   93064.78238323,   53246.3312291 , 88080.30643839,
1544.01924478, -157510.31165492, 91905.84577891, -104120.30338562,   -
7778.92437832, 5252.67709964,  -93950.90837818,  -24732.85666885,
2998.60044099,  -46121.70219599, -178946.07115258, -53158.56432145,
39374.73070183, -142511.10737582, -93467.10862949, -119163.81965495,
86433.73556314, -19493.47186888,  -43328.4347383 , -149292.44670008, -
31467.57278374, -140689.93945916,  -77135.24975531, -137226.1470541 ,
-19121.00345482,  -28106.82650466, -98746.88800202,  -44359.39586045,
-178375.53578575, -214213.1833435 ,  -40454.74688619,  -
64999.38541647, -22847.17067971,  134483.02973775,    5003.15382914, -
162154.00028997,   20531.46592863, -198431.66694604, -121542.61443332,
-86141.74447922,   74200.84494844, -147027.93398436, -154379.46847931,
-88860.72719829, -139713.04577259,   21397.23298959, -177193.83575271,
-183272.178717  , -119403.804027  , -124822.92056231, 93657.88484353,
5447.87262332,  -72120.38827533, -190289.19669472,   -4007.33212386, -
170019.38126506, -219029.39870999,   26922.68131171,  -51475.16492676,
2877.29414027,  -51314.51123513,   -2885.24492876, -138592.30339701, -
173081.8557606 ,  -18656.49335465, -152306.86977565, -142059.47999752,
-120997.92531656, -78426.87568774])
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, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1,
0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1,
1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 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

```
# 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_test))

Training accuracy: 0.825    Testing accuracy: 0.6
```

Testing several pipelines

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

Scalers:

- None
- MinMax
- Standard

Classifiers:

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

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

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

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

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

        # YOUR CODE GOES HERE
        # Create a pipeline
        pipeline = Pipeline([scaler, model])
        # Fit the pipeline on X_train, y_train
        pipeline.fit(X_train, y_train)
        # Calculate acc_train and acc_test for the pipeline
        acc_train = accuracy_score(y_train, pipeline.predict(X_train))
        acc_test = accuracy_score(y_test, pipeline.predict(X_test))

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

   The 1-NN classifier model's testing accuracy improved the most by scaling data.

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

   On this data overall, the standard scalar, the min/max scalar, and the no scaler performed the same for logistic regression models. For support vector classifiers, The standard scalar and the min/max scaler gave arguably worst results. For 1-NN classifiers, the standard scalar and the min/max scalar gave better test accuracy.

M6-L1 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:

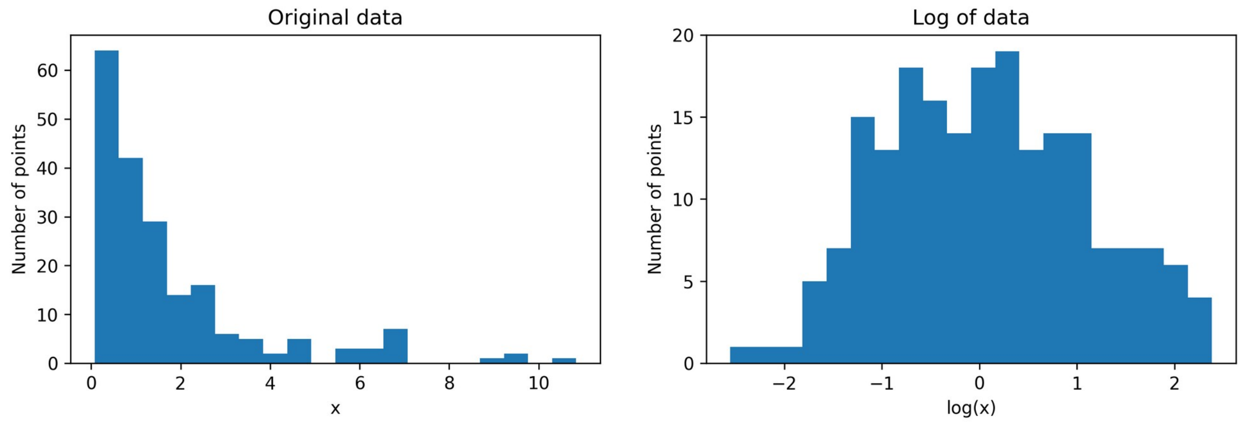
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer
from sklearn.svm import SVR

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",line
widths=.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,  0.90192956,
1.50771989, 1.15493443,  4.28137195,  2.14049632,  1.12938701,
1.55871729, 1.3960884 ,  4.45523172,  0.8145184 ,  1.36761412,
0.42566793, 0.07784856,  1.92248495,  2.37366743,  0.47608207,
9.67702601, 0.23354846,  1.04682159,  0.82929126,  4.63102958,
4.34644717, 1.16759657,  1.45960014,  0.41156606,  0.13795931,
0.70616091, 1.16923416,  3.42222417,  3.32802771,  0.67886919,
0.73911426, 0.35044449,  0.24170968,  0.18154165,  7.0341397 ,
0.60070448, 0.64527784,  0.28570503,  2.17600441,  0.19911   ,
0.80836606, 0.408417  ,  1.47241292,  0.60001229,  0.30708454,
0.97221119, 1.53469532,  1.06877937,  1.35319965,  0.53029486,
0.6957665 , 0.51045109,  0.69798814,  0.44346062,  0.17794467,
1.19413986, 0.66912731,  0.19589072,  1.58848742,  0.40361317,
1.05331823, 2.07319431,  1.13767068,  3.12489501,  0.29088542,
1.49532211, 0.50418597,  0.41861772,  0.56054281,  0.73230914,
1.05777256, 0.31187593,  2.46163678,  1.59306915,  0.2151879 ,
4.42934711, 6.65846632,  3.25040489,  0.835333  ,  0.34275046,
2.87040096, 0.66819385,  3.39547978,  1.23155177,  2.65551613,
1.42813072, 2.02703304,  1.01055534,  5.96476998,  1.13531721,
1.49479543, 6.57418553,  0.25982185,  0.28069545,  2.63635349,
0.30939905, 6.98399558,  0.66125285,  0.47357035,  6.84105546,
4.39520771, 6.47247753,  2.4745156 ,  0.42264374,  6.75352745,
0.7649052 , 2.23101446,  2.5786138 ,  0.85640653,  1.84795453,
2.51483368, 1.45706703,  0.3330706 ,  1.34748269,  3.76740297,
0.49929016, 0.86102259,  0.64716529,  6.35513869,  1.95872697,
1.50299808, 0.46305193,  1.71471895,  0.50949631,  1.03234257,
0.52948731, 1.96685003,  1.77995987,  0.81196442,  1.48587929,
0.33518874, 0.22508941,  1.551763  ,  1.18136848,  1.88708146,
10.83893534, 2.57147454,  0.40138981,  3.05572319,  0.26823082,
0.6302841 , 0.93403478,  5.54747418,  0.47485071,  0.43760503,
0.90623872, 0.5150567 ,  3.08525997,  0.33961879,  0.3174393 ,
0.64544192, 0.60772521,  6.88628708,  2.58421247,  1.09149819,
0.29362979, 2.32649531,  0.36780023,  0.2133607 ,  3.28061135,
1.37292378, 2.51144635,  1.37537669,  2.35568278,  0.52151064,
0.35549545, 1.97702763,  0.44779951,  0.50180194,  0.63411021,
1.01763281, 0.70187924,  0.25285191,  0.52538792,  0.10824012,
1.86867841, 0.20148151,  0.33141519,  1.05354965,  0.47732246,
4.67867334, 0.27448548,  1.30610689,  0.96147875,  0.31095922,
1.68754812, 0.84236124,  2.16363689,  2.27846997,  8.69924247,
3.80580659])
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, -6.27910380e+00, -
4.66593935e+00, -3.27628398e+00, -5.41260576e+00, -3.07553025e+00, -
2.60088666e+00, -4.94126516e+00, -5.07104868e+00, -6.78065624e+00, -
5.64645372e+00, -2.45259954e+00, -2.84042416e+00, -1.57873879e+00, -
2.01053220e+00, -1.81709993e+00, -6.43544903e+00, -6.92943404e+00, -
1.43153401e+00, 4.29485069e+01, -1.01830444e+00, -3.90351271e+00, -
3.11046074e+00, -2.60468704e+00, -3.19751543e+00, -6.61079247e+00, -
5.90754795e+00, -2.70273587e+00, -4.66887251e-02, -4.76641497e+00, -
3.30726512e+00, -3.15777577e+00, -8.66934765e+00, -2.29409449e+00, -
2.13391937e+00, -2.58556664e+00, -1.74603256e+00, -1.07407173e+00,
1.38617365e+01, -3.10619598e+00, -5.32401140e+00,  3.81599556e-01, -
4.52559897e+00, -2.17595159e+00, -5.58801110e+00, -1.09368325e+00, -
6.05774675e+00, -2.42711696e+00, -1.92011443e+00, -2.87855321e+00, -
4.27606315e+00, -5.29000358e+00, -7.00989489e+00, -4.74466924e+00, -
2.07917240e+00, -4.07498403e+00, -3.76297780e+00, -2.91511682e+00, -
```

```
9.36910003e-01, -7.44914900e+00, -2.61473730e+00, -1.55243871e-01, -
5.28651169e+00, -2.32149151e+00, -4.01101159e+00, -5.46926738e+00, -
8.55294796e+00, -2.92563777e+00, -7.84672807e-01, -6.21923521e+00, -
2.85315642e+00, -1.17723512e+00, -2.66266171e+00, -6.17129572e+00, -
1.07324073e+00, -1.62792403e+00, -4.71826920e+00, -6.46555121e+00,
1.27493192e+00, -2.09810420e+00,  1.19561079e+01, -7.25477255e+00, -
1.66216583e+00, -5.61547171e-01, -4.16003379e+00, -3.83661758e+00, -
6.16965664e+00, -1.18516405e+00, -7.81583847e+00, -5.30502079e+00, -
4.32096521e+00, -3.88496715e+00,  6.62906156e+00, -4.98681443e+00, -
4.68447995e+00, 8.38919748e+00,  1.25559415e+00, -1.50193339e+00, -
7.25167503e+00, -4.51692863e-01,  1.31651367e+01, -4.87039664e+00, -
4.16365912e+00, 1.36354222e+01, -2.89754788e+00,  9.33002536e+00, -
4.63273484e+00, -3.62482967e+00,  1.07464791e+01, -3.81676576e+00, -
6.03611939e+00, -6.30707705e+00, -3.97893131e+00, -5.91727631e+00, -
6.41073788e+00, -6.24169740e+00, -2.77390647e+00, -4.50992930e+00, -
5.98006234e+00, -3.98319304e+00, -4.03219142e+00, -3.05350405e+00,
1.19971796e+01, -7.01407392e+00, -3.84109609e+00, -9.80060053e-01, -
7.41675111e+00, -1.12801561e+00, -5.87180262e+00, -6.35583810e+00, -
5.05627183e+00, -8.36537808e+00, -2.72413419e+00, -6.24757554e+00,
9.25733994e-01, 5.27982307e-01, -6.03092529e+00, -5.54296733e+00, -
7.69544697e+00, 6.26264586e+01, -6.66542463e+00, -2.39287559e+00, -
5.70611595e+00, -4.01424069e-01, -2.22968078e+00, -4.94396881e+00,
8.80411673e-01, -1.01972575e-01, -3.03070076e+00, -4.68836537e+00, -
3.09178407e+00, -8.67207510e+00, -2.29971402e+00, -2.20591252e+00, -
1.88007689e+00, -1.62161041e+00,  1.29951706e+01, -4.84513570e+00, -
3.75617518e+00, -1.40545367e+00, -5.97850387e+00, -1.98970437e+00, -
1.61355500e+00, -6.04224622e+00, -6.67171619e+00, -5.82920642e+00, -
6.47490784e+00, -4.96672564e+00, -2.70976774e+00, -1.57685774e+00, -
5.23574473e+00, -1.07350146e+00, -4.61313963e+00, -3.07881081e+00, -
2.73231916e+00, -5.56392046e+00, -6.19404753e-01, -5.73425346e+00, -
2.06324496e+00, -5.79348723e+00, -3.45188541e+00, -3.02550603e+00, -
6.36553389e+00, 3.13426823e-01, -1.25704084e+00, -4.46149712e-01, -
5.15863188e+00, -4.41309998e+00, -3.88281175e+00, -6.02767799e+00, -
4.64447206e+00, -4.84997397e+00, -4.48927165e+00,  3.43804753e+01, -
3.26604125e+00])

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.

```
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()
```

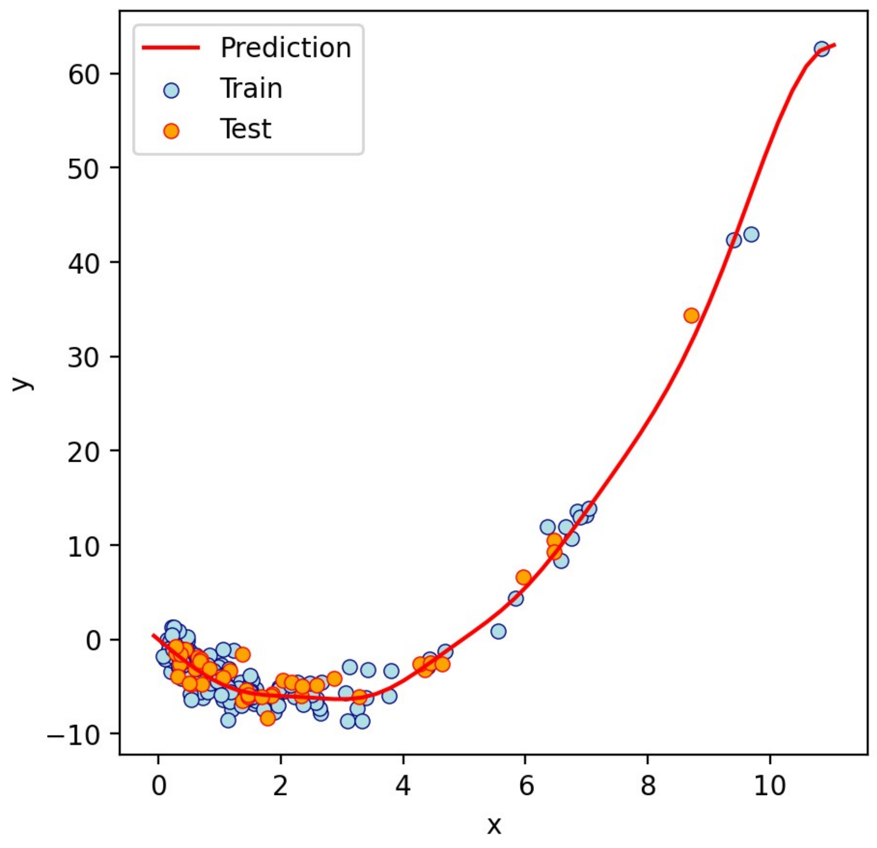


No log transform

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

```
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: 2.0710061552336008    Testing MSE: 1.9453578716771713
```



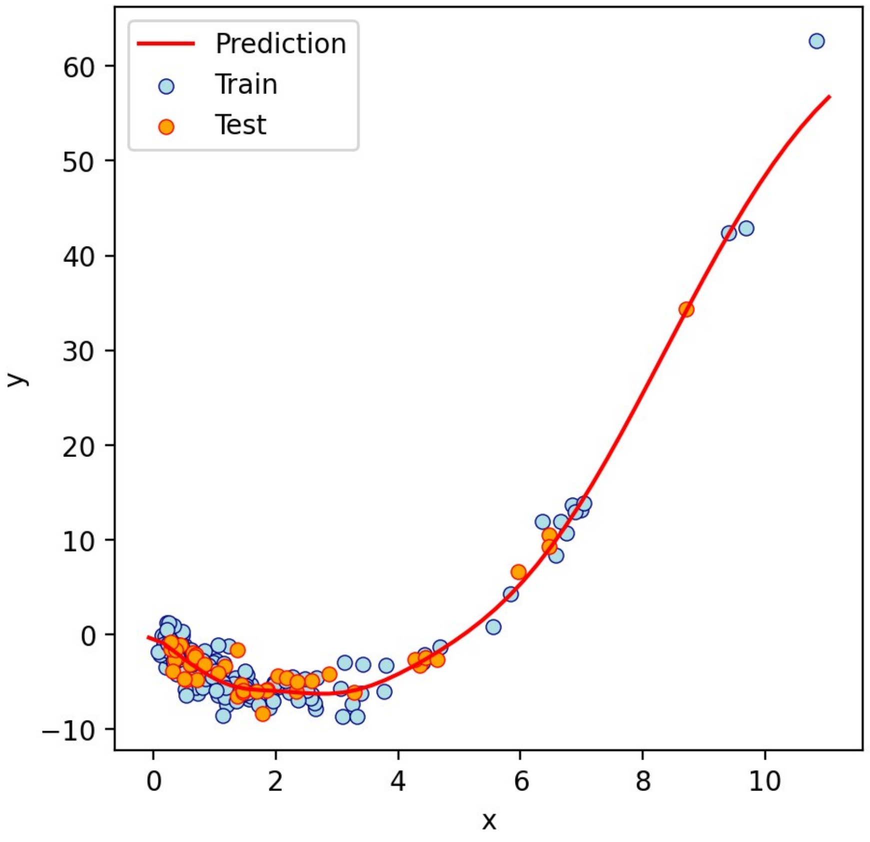
With log transform

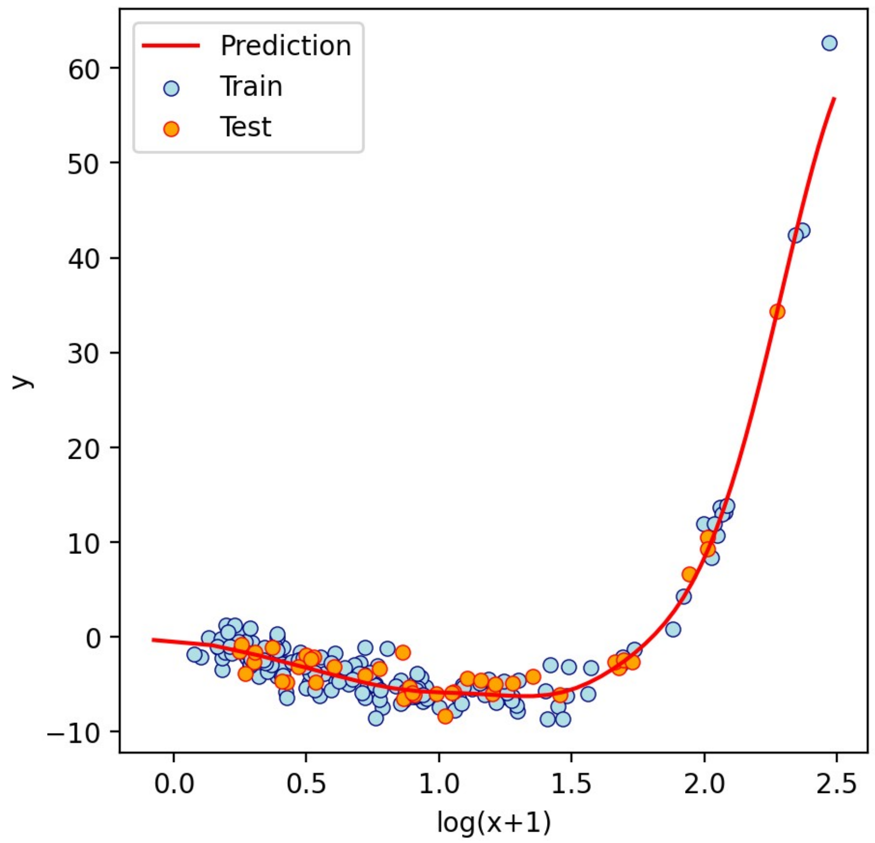
Notice that the data are not spread uniformly across the x axis. Instead, most input data points have low values -- this is a roughly "log normal" distribution. If we take the log of the input, we saw it was more normally distributed, which can improve machine learning model results in some cases. The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

```python
def log_transform(x):
    return np.log(x + 1.)

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


# YOUR CODE GOES HERE
pipeline = Pipeline([("log",transform),("SVR", model)])
pipeline.fit(X_train, y_train)
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(X_train, X_test, y_train, y_test, pipeline)
plot(X_train, X_test, y_train, y_test, pipeline,log=True)

Training MSE: 2.31392147316065    Testing MSE: 1.7200875366333044
```





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)

M6-L2 Problem 1 (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`.

```
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],
[22.8,4,108,93,3.85,2.32,18.61,1,1,4,1],
[21.4,6,258,110,3.08,3.215,19.44,1,0,3,1],
[18.7,8,360,175,3.15,3.44,17.02,0,0,3,2],
                [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],
[24.4,4,146.7,62,3.69,3.19,20,1,0,4,2],
[22.8,4,140.8,95,3.92,3.15,22.9,1,0,4,2],
[19.2,6,167.6,123,3.92,3.44,18.3,1,0,4,4],
                [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],
[17.3,8,275.8,180,3.07,3.73,17.6,0,0,3,3],
[15.2,8,275.8,180,3.07,3.78,18,0,0,3,3],
[10.4,8,472,205,2.93,5.25,17.98,0,0,3,4],
                [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],
[32.4,4,78.7,66,4.08,2.2,19.47,1,1,4,1],
[30.4,4,75.7,52,4.93,1.615,18.52,1,1,4,2],
[33.9,4,71.1,65,4.22,1.835,19.9,1,1,4,1],
                [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],
[15.2,8,304,150,3.15,3.435,17.3,0,0,3,2],
[13.3,8,350,245,3.73,3.84,15.41,0,0,3,4],
[19.2,8,400,175,3.08,3.845,17.05,0,0,3,2],
                [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],
[30.4,4,95.1,113,3.77,1.513,16.9,1,1,5,2],
[15.8,8,351,264,4.22,3.17,14.5,0,1,5,4],
[19.7,6,145,175,3.62,2.77,15.5,0,1,5,6],
                [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.

```
def find_redundant_features(data, target_index, threshold):
    # YOUR CODE GOES HERE
    redundant_features = set()
    correlation_matrix = np.abs(np.corrcoef(data.T))
    features = correlation_matrix.shape[0]

    for i in range(features):
        for j in range(i+1, features):
            if i != target_index and j != target_index and 
correlation_matrix[i, j] > threshold:
                redundant_features.add(i if 
correlation_matrix[target_index, i] < correlation_matrix[target_index, 
j] else j)

    return list(redundant_features)
```

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.

```
# YOUR CODE GOES HERE
test_cases = [(0, 0.9), (2, 0.7), (10, 0.8)]
```

```python
for target_index, threshold in test_cases:
    removed_indices = find_redundant_features(data, target_index,
threshold)
    print(f"Target Index: {target_index}, Threshold: {threshold} =>
Removed Indices: {removed_indices}")
```

```
Target Index: 0, Threshold: 0.9 => Removed Indices: [2]
Target Index: 2, Threshold: 0.7 => Removed Indices: [0, 3, 4, 5, 6, 7,
8, 9, 10]
Target Index: 10, Threshold: 0.8 => Removed Indices: [1, 2, 5]
```

Using your function

Run these additional cases and print the results: | target_index | threshold | | Indices to remove | |---|---|---|---| | 4 | 0.9 | | ? | | 5 | 0.8 | | ? | | 6 | 0.95 | | ? |

```python
# YOUR CODE GOES HERE
additional_test_cases = [(4, 0.90), (5, 0.80), (6, 0.95)]

for target_index, threshold in additional_test_cases:
    removed_indices = find_redundant_features(data, target_index,
threshold)
    print(f"Target Index: {target_index}, Threshold: {threshold} =>
Removed Indices: {removed_indices}")
```

```
Target Index: 4, Threshold: 0.9 => Removed Indices: [1]
Target Index: 5, Threshold: 0.8 => Removed Indices: [0, 1, 3, 7]
Target Index: 6, Threshold: 0.95 => Removed Indices: []
```

M6-L2 Problem 2 (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`.

```
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],
[22.8,4,108,93,3.85,2.32,18.61,1,1,4,1],
[21.4,6,258,110,3.08,3.215,19.44,1,0,3,1],
[18.7,8,360,175,3.15,3.44,17.02,0,0,3,2],
                [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],
[24.4,4,146.7,62,3.69,3.19,20,1,0,4,2],
[22.8,4,140.8,95,3.92,3.15,22.9,1,0,4,2],
[19.2,6,167.6,123,3.92,3.44,18.3,1,0,4,4],
                [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],
[17.3,8,275.8,180,3.07,3.73,17.6,0,0,3,3],
[15.2,8,275.8,180,3.07,3.78,18,0,0,3,3],
[10.4,8,472,205,2.93,5.25,17.98,0,0,3,4],
                [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],
[32.4,4,78.7,66,4.08,2.2,19.47,1,1,4,1],
[30.4,4,75.7,52,4.93,1.615,18.52,1,1,4,2],
[33.9,4,71.1,65,4.22,1.835,19.9,1,1,4,1],
                [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],
[15.2,8,304,150,3.15,3.435,17.3,0,0,3,2],
[13.3,8,350,245,3.73,3.84,15.41,0,0,3,4],
[19.2,8,400,175,3.08,3.845,17.05,0,0,3,2],
                [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],
[30.4,4,95.1,113,3.77,1.513,16.9,1,1,5,2],
[15.8,8,351,264,4.22,3.17,14.5,0,1,5,4],
[19.7,6,145,175,3.62,2.77,15.5,0,1,5,6],
                [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.

```
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.

```
def get_next_pair(X, y, current_indices):
    # YOUR CODE GOES HERE
    lowest_mse = np.inf
    best_pair = ()

    for i, j in itertools.combinations(range(X.shape[1]), 2):
        if i not in current_indices and j not in current_indices:
            indices = list(current_indices) + [i, j]
            mse_train, mse_test = get_train_test_mse(X, y,
list(map(int, indices)))
```

```
            if mse_test < lowest_mse:
                lowest_mse = mse_test
                best_pair = (i, j)
    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)

```
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?

Features 1 and 3 were deemed least important by the wrapper method.