

Exercise 2: Regularization

(a)

Simulate a data set with $n=100$ observations based on the relationship $y = \sin(x_1) + \varepsilon$ with noise term ε following some distribution. Simulate $p=100$ additional covariates x_2, \dots, x_{101} that are not related to y .

```
# if you want to reproduce the results, please install the following
packages
!pip install numpy==1.26.4 matplotlib==3.8.4 scikit-learn==1.5.2
tensorflow==2.18.0

import numpy as np
import matplotlib.pyplot as plt

plt.style.use('ggplot')

np.random.seed(509)

num_points = 100
num_additional_covs = 100

# the reader may find it interesting to play around with the noise
amount
noise_amount = 0.1

X = np.random.randn(num_additional_covs, num_additional_covs + 1)
y = np.sin(X[:, 0]) + np.random.randn(num_points) * noise_amount
print(f"X shape: {X.shape}\ny shape: {y.shape}")

X shape: (100, 101)
y shape: (100,)
```

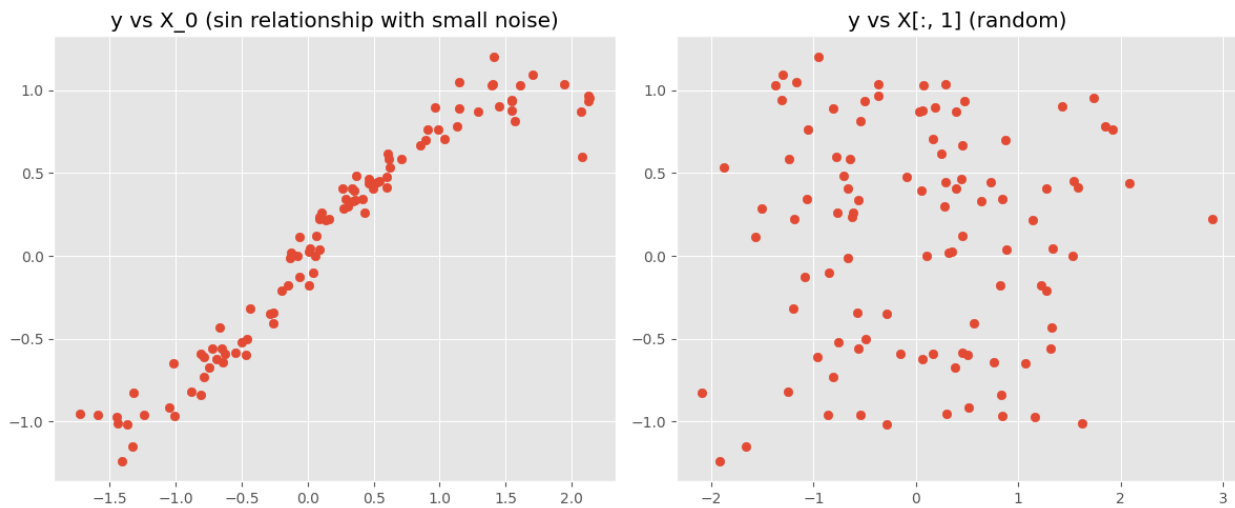
let's see the relationship of Y with x_1 and an unrelated covariate x_2

```
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

axs[0].scatter(X[:, 0], y)
axs[0].set_title('y vs X_0 (sin relationship with small noise)')

axs[1].scatter(X[:, 1], y)
axs[1].set_title('y vs X[:, 1] (random)')
```

```
plt.tight_layout()
plt.show()
```



(b)

On this data set, use different models (and software packages) of your choice to demonstrate

- overfitting and underfitting;
- L_1 , L_2 and elastic net regularization;
- the underdetermined problem;
- the bias-variance trade-off;
- early stopping using a simple neural network. %(use a simple neural network as in Exercise 2).

overfitting and underfitting

Let's try to approximate the (sinusoidal) relationship between y and x_1 with a simple linear model and with a polynomial model of degree 7.

Also, in order for us to see the effects of under/over-fitting not only usually but also numerically, let's split the data into training and test sets and calculate the training and test errors for each model.

Underfitting

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

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

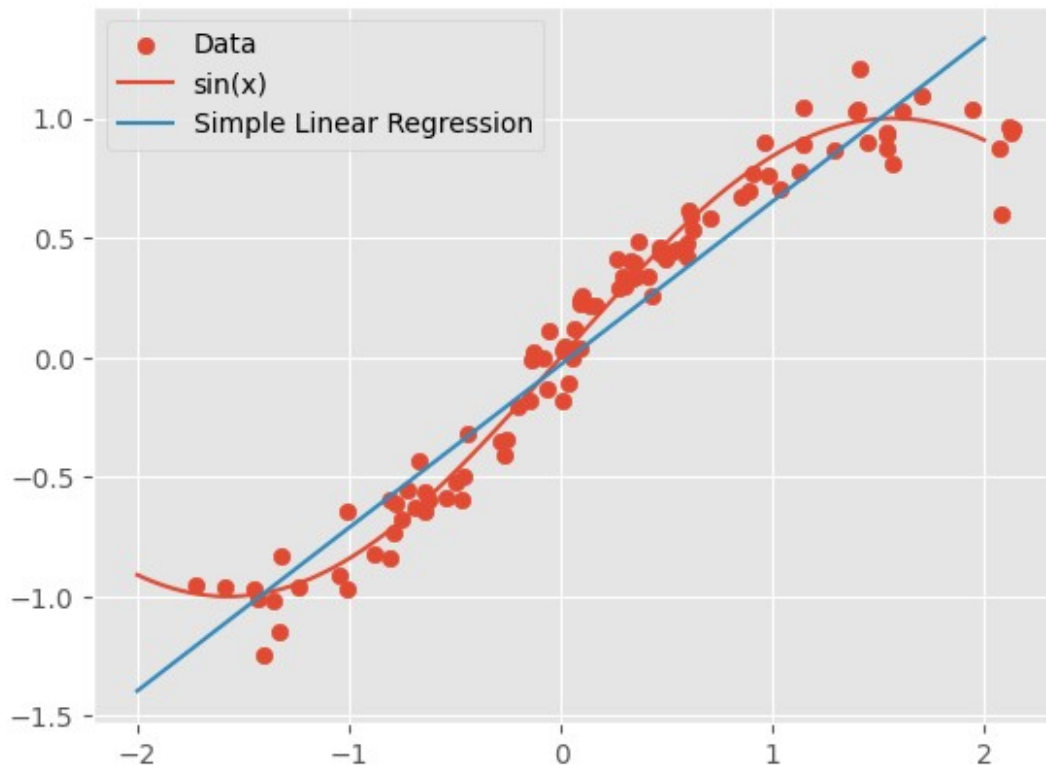
x_1 = X[:, 0].reshape(-1, 1)
X_train, X_test, y_train, y_test = train_test_split(x_1, y,
```

```
test_size=0.2,  
random_state=509)
```

```
lr = LinearRegression()  
  
lr.fit(X_train, y_train)  
  
y_train_pred = lr.predict(X_train)  
y_test_pred = lr.predict(X_test)  
  
train_mse = mean_squared_error(y_train, y_train_pred)  
test_mse = mean_squared_error(y_test, y_test_pred)  
  
print(f"Simple Linear Regression")  
print(f"Train MSE: {train_mse:.4f}\nTest MSE: {test_mse:.4f}")  
print(f"Test error is higher by {((test_mse - train_mse) / train_mse)  
* 100:.2f}%")  
  
Simple Linear Regression  
Train MSE: 0.0313  
Test MSE: 0.0595  
Test error is higher by 90.07%
```

visual inspection

```
plt.scatter(x_1, y, label='Data')  
  
# noiseless DGP  
x_vals = np.linspace(-2, 2, 100)  
y_vals = np.sin(x_vals)  
plt.plot(x_vals, y_vals, label='sin(x)')  
  
# simple linear regression  
plt.plot(x_vals, lr.coef_[0] * x_vals + lr.intercept_,  
         label='Simple Linear Regression')  
plt.legend()  
  
<matplotlib.legend.Legend at 0x1d0b3a0a560>
```



The training error was ~ 0.03 and test error was ~ 0.06 which may indicate that the model is overfitting, but we must first ask ourselves whether or not the results on the training set are actually good.

By plotting the predictions against the actual data we can see that the model lacks complexity and we should first deal with the underfitting problem.

Overfitting

```
# now let's try polynomial regression
poly_degree = 50
poly = PolynomialFeatures(degree=poly_degree)
poly.fit(X_train)

X_train_poly = poly.transform(X_train)
X_test_poly = poly.transform(X_test)

# don't get confused by the model name
# we will be fitting it on the transformed data
poly_reg = LinearRegression()
poly_reg.fit(X_train_poly, y_train)

y_train_pred = poly_reg.predict(X_train_poly)
y_test_pred = poly_reg.predict(X_test_poly)

train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
```

```
print(f"\nPolynomial Regression (degree=7)")
print(f"Train MSE: {train_mse:.4f}\nTest MSE: {test_mse:.4f}")
print(f"Test error is higher by {((test_mse - train_mse) / train_mse)
* 100:.2f}%")
```

```
Polynomial Regression (degree=7)
Train MSE: 0.1385
Test MSE: 145219784.4922
Test error is higher by 104827038582.99%
```

```
plt.scatter(x_1, y, label='Data')
```

```
# noiseless DGP
```

```
x_vals = np.linspace(-2, 2, 100)
```

```
y_vals = np.sin(x_vals)
```

```
plt.plot(x_vals, y_vals, label='sin(x)')
```

```
plt.plot(x_vals, poly_reg.predict(poly.transform(x_vals.reshape(-1,
1))),
```

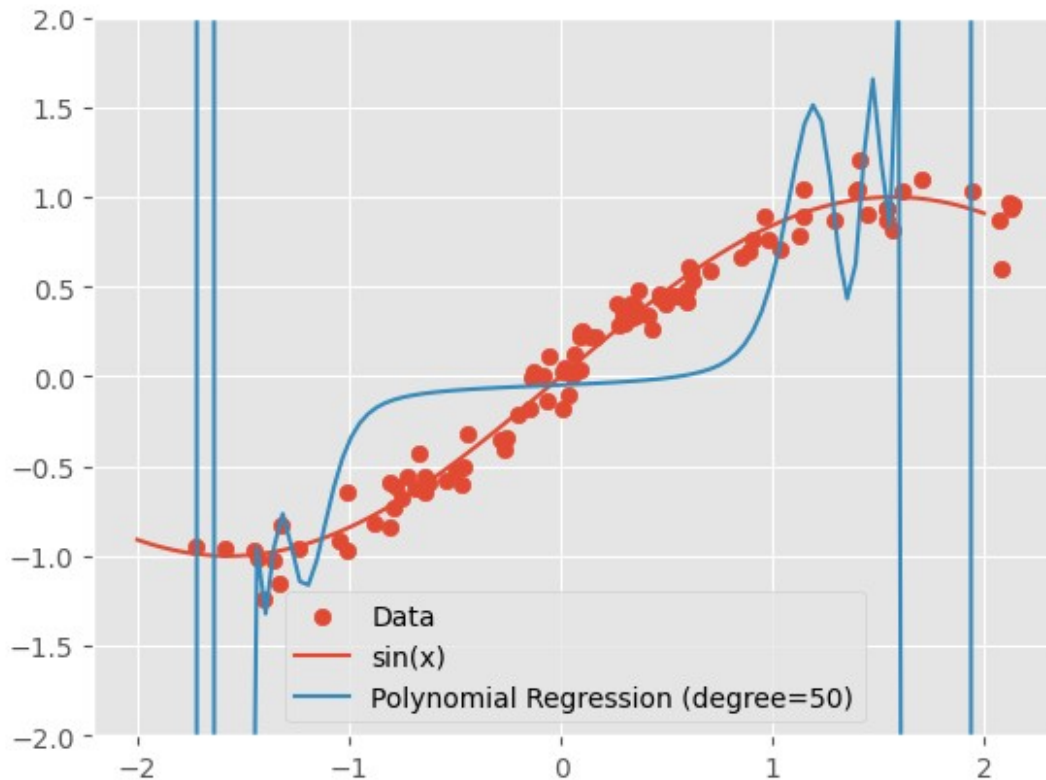
```
label=f'Polynomial Regression (degree={poly_degree})')
```

```
# setting limits to make the plot readable (the model sometimes
predicts extreme values)
```

```
plt.ylim(-2, 2)
```

```
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1d0c6ad9630>
```



In this case both from visual inspection and from looking at the training and test errors we can see that the model is extremely overfitting the data. By looking at the predictions for the points around -1.3 we can nicely see how much the model bends to pass through the training data points, and learns the noise instead of the actual pattern.

The reader may find it interesting to decrease the degree of the polynomial to make the demonstration less extreme, and for degree = 7 you may even notice that the train/test errors alone indicate less overfitting than in the case of simple linear regression.

Also, we encourage the reader to try to find the optimal degree of the polynomial.

L_1 , L_2 and elastic net regularization;

```
from sklearn.linear_model import enet_path

def plot_regularization_paths(X, y, l1_ratio, title):
    alphas, coefs, _ = enet_path(X, y, l1_ratio=l1_ratio)

    plt.figure()
    for coef in coefs:
        plt.plot(alphas, coef)

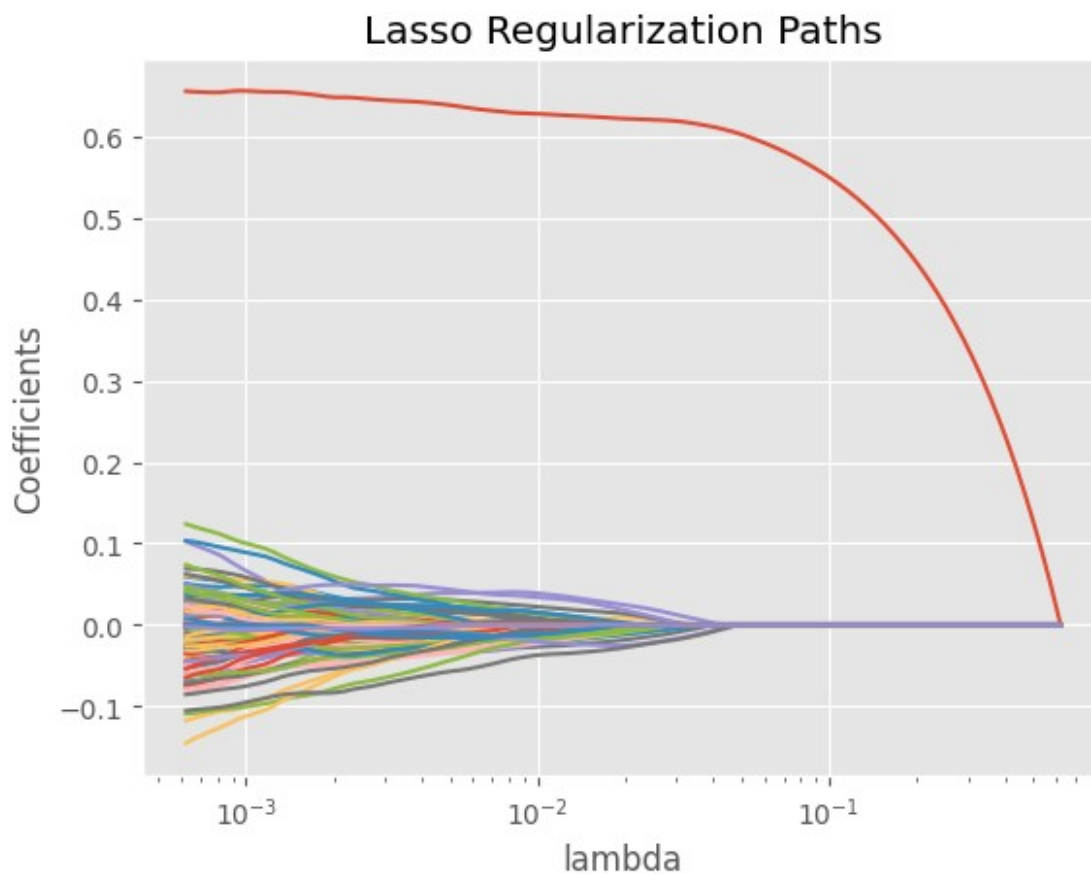
    plt.xscale("log")
    plt.xlabel("lambda")
    plt.ylabel("Coefficients")
```

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

L1 (Lasso)

For weak lambda values we can see that the uninformative covariates do have a non zero coefficients, but by increasing the lambda they eventually go to zero. Although when we penalize the coefficients too strongly the coefficient of x_1 also goes to zero, which is not what we want.

```
plot_regularization_paths(X, y, l1_ratio=1, title="Lasso  
Regularization Paths")
```

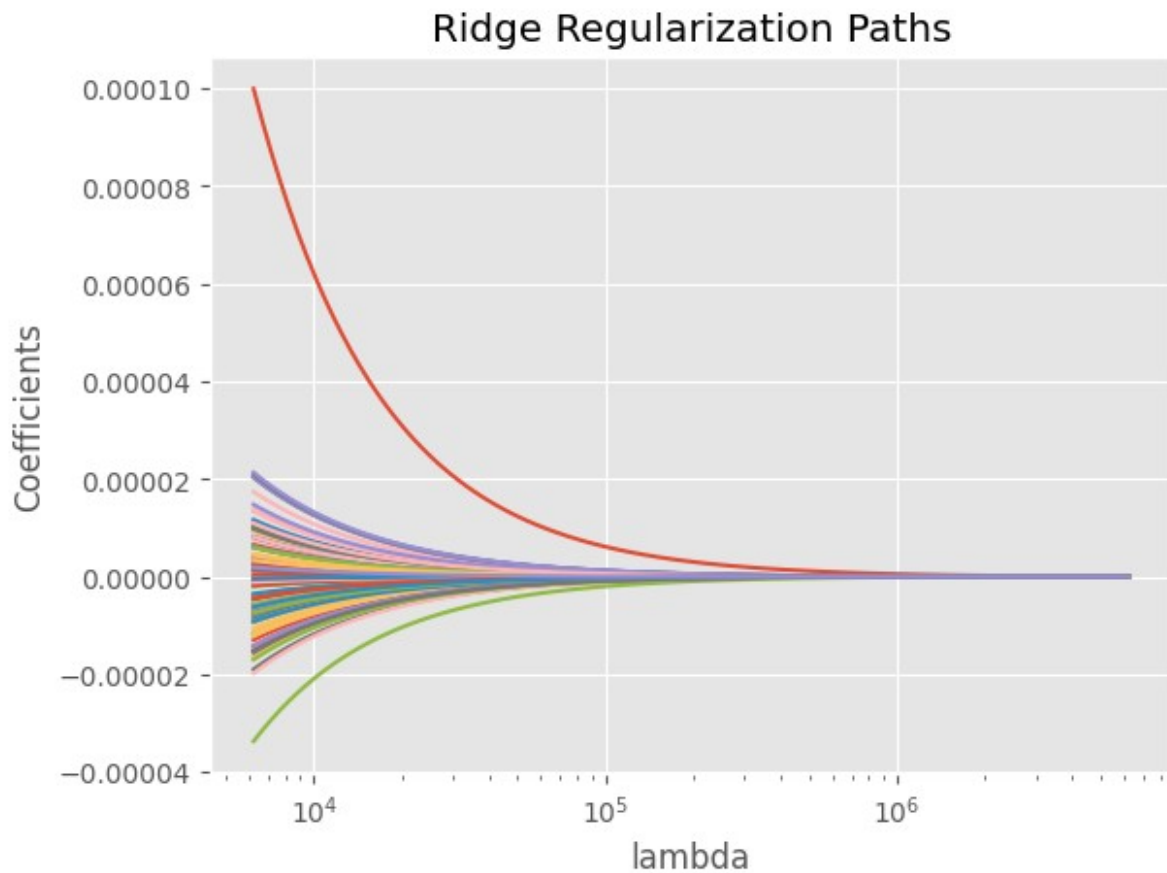


L2 (Ridge)

A few differences from the plot above:

- the lambda values are much bigger here (smallest being around 10^4 instead of 10^{-3})
- as a result we also see that the coefficients are much smaller (0.0001 instead of 0.65)

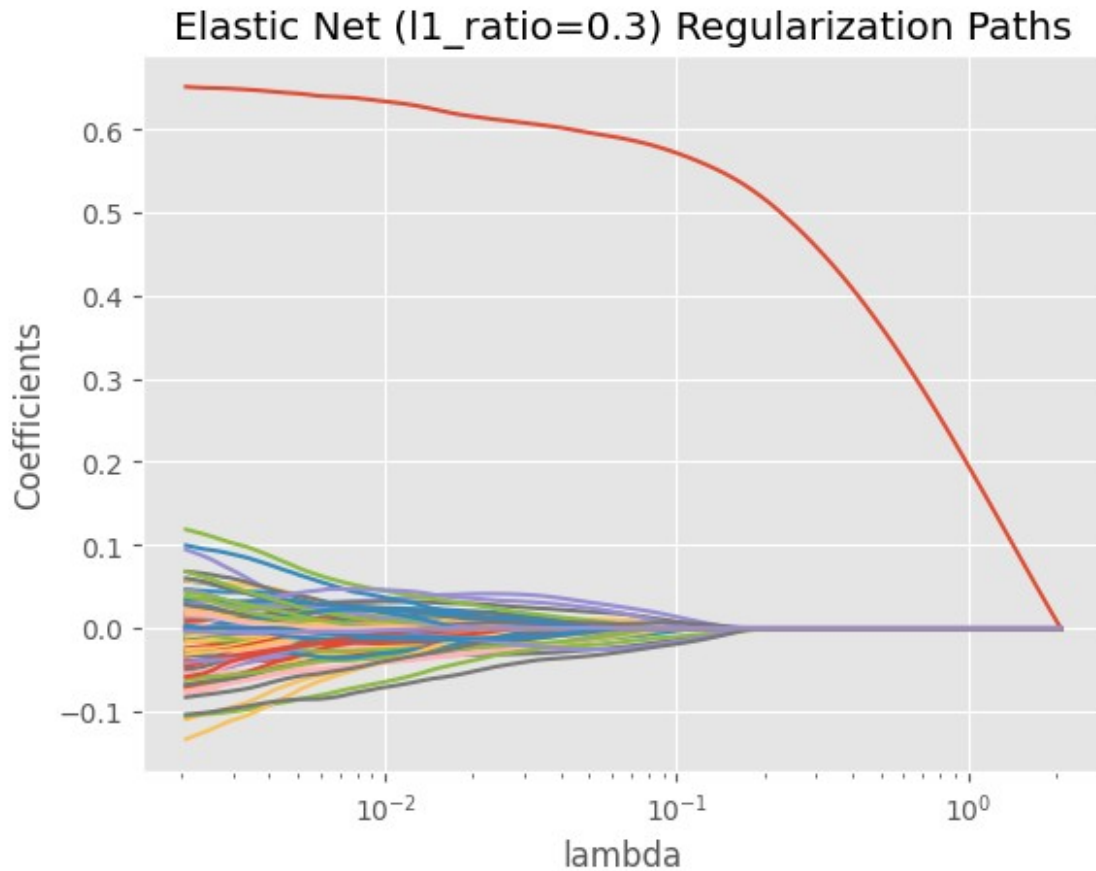
```
plot_regularization_paths(X, y, l1_ratio=1e-7, title="Ridge  
Regularization Paths")
```



Elastic Net (with $l1_ratio = 0.3$)

this one is somehow in-between the two above, lambda's are bigger, but not too big.

```
plot_regularization_paths(X, y, l1_ratio=0.3, title="Elastic Net  
(l1_ratio=0.3) Regularization Paths")
```

The reader may find it interesting to compare the plots for the same λ values. Or maybe that's something that we can add :-)

the underdetermined problem

Note: I struggled a bit to achieve the same result as the R code did, but maybe this type of solution that shows the journey is more useful for the students.

if we have more covariates than observations, then the design matrix is not full rank \Rightarrow the inverse of the design matrix does not exist \Rightarrow we cannot solve the normal equations \Rightarrow we have an underdetermined problem.

```
print(f"X shape: {X.shape}")
```

```
X shape: (100, 101)
```

we have more covariates than samples, but practically this does not imply that we have a zero determinant

```
np.linalg.det(X.T @ X)
```

```
-3.382261897412918e+143
```

```
# let's select even less samples
```

```
X_head = X[:10]
```

```
y_head = y[:10]
```

```
np.linalg.det(X_head.T @ X_head)
```

```
0.0
```

although we see that the determinant is zero, let's try to compute the inverse (although in practice we would instead solve the SLE $X^T X \beta = X^T Y$)

```
np.linalg.inv(X_head.T @ X_head)
```

```
array([[ -6.14560621e+14,  3.23258780e+14,  2.31422320e+14, ...,
         7.36667699e+13, -4.61268529e+14,  1.40062310e+14],
       [ 7.05436025e+13, -1.48872103e+14, -1.05705693e+14, ...,
         2.62952070e+13, -2.74648424e+13, -9.81077912e+12],
       [-5.05638914e+13,  1.31386044e+14,  4.53445891e+13, ...,
         2.75666330e+14,  2.93356907e+14, -3.74015010e+13],
       ...,
       [-1.06616166e+14, -2.16935255e+13, -2.74450882e+13, ...,
         5.07348591e+13, -1.33374666e+14,  1.40038499e+14],
       [ 3.10643252e+13,  4.88795481e+13, -4.46876100e+13, ...,
         1.04166621e+14,  2.10377348e+14, -6.07121343e+13],
       [-2.03686412e+14, -9.96492008e+13, -1.77264094e+14, ...,
         3.12507118e+14, -5.36625860e+13,  2.98007478e+14]])
```

We still get an output. If you're curious about the reasoning behind this, please see [this stackoverflow post](#).

```
np.linalg.solve(X_head.T @ X_head, X_head.T @ y_head)
```

```
array([-1.74285315,  0.8537766 , -0.8110621 , -0.63846306,
        1.09957492,
        0.70411593, -0.81387694,  0.20231228, -0.57019194, -
        0.25694313,
        0.33291916, -0.90583084, -0.1593664 , -0.00836486,
        0.48912791,
        -0.34093805, -0.14619479,  0.26705641, -0.13000239, -
        1.0676898 ,
        -0.75372774, -0.36068403,  1.99980035,  0.70520051, -
        0.78846672,
        -1.1129352 , -1.04849859,  2.44214345,  0.07787901, -
        0.12101252,
        0.13348138,  0.43843487, -0.45808393,  0.65824791, -
        0.33972536,
        0.5243654 , -0.90531215, -1.24247628, -0.76020155,
        0.61623271,
        0.64464659,  0.21378584, -0.56351514,  0.38553472, -
```

```

0.15249909,
    -0.45513269,  0.72338789, -0.1665285 , -0.76297868, -
0.46868489,
    -0.06560666,  0.09467768, -0.23238404, -0.55247066,
0.68868233,
    -0.00795327,  0.17848214,  0.88116131, -1.35698687,
0.30066788,
    1.41038654,  0.09029448, -0.59959995, -0.20006567, -
0.11064867,
    0.52090078,  0.21543092, -0.75079147, -0.68499159,
0.47609125,
    0.49922526, -0.81350002,  0.39819968, -1.12715084, -
0.72515981,
    -1.67792737,  0.34845666, -0.07935231,  0.35450805, -
0.76274803,
    -0.209724 ,  0.39690894, -0.45179311, -0.14240346,
0.10284446,
    0.33539185, -0.24189984, -1.09252272, -0.11627265, -
0.52272444,
    -0.37402145, -0.28182224,  0.46311887, -0.67474195,
0.36903622,
    0.23373493, -1.20457307,  0.69818135, -0.62333609,
0.02863677,
    0.40637021])

```

The code above does produce an output, since from the machine perspective the matrix is not singular, because of the floating point approximations. Let's demonstrate the underdetermined problem with a manually created example.

```

X_singular = np.array([
    [1, 1],
    [1, 1],
], dtype=float)

# y can be anything
y_any = np.array([1, 2])

print("X shape:", X_singular.shape)
print("y shape:", y_any.shape)
print(f"Rank(X) = {np.linalg.matrix_rank(X_singular)} < {X_singular.shape[1]}")

try:
    coefs = np.linalg.solve(X_singular.T @ X_singular, X_singular.T @
y_any)
    print("Direct solution via solve(X^T X, X^T y):", coefs)
except np.linalg.LinAlgError as e:
    print("Error from np.linalg.solve:", e)

```

```
X shape: (2, 2)
y shape: (2,)
Rank(X) = 1 < 2
Error from np.linalg.solve: Singular matrix
```

the bias-variance trade-off

```
x_1 = X[:, 0]
sorted_x_1 = np.sort(x_1)

degrees = [1, 3, 15]

plt.figure()
plt.scatter(x_1, y, alpha=0.3, label='Data')
plt.plot(sorted_x_1, np.sin(sorted_x_1), label='Noiseless sin(x)',
linewidth=2)

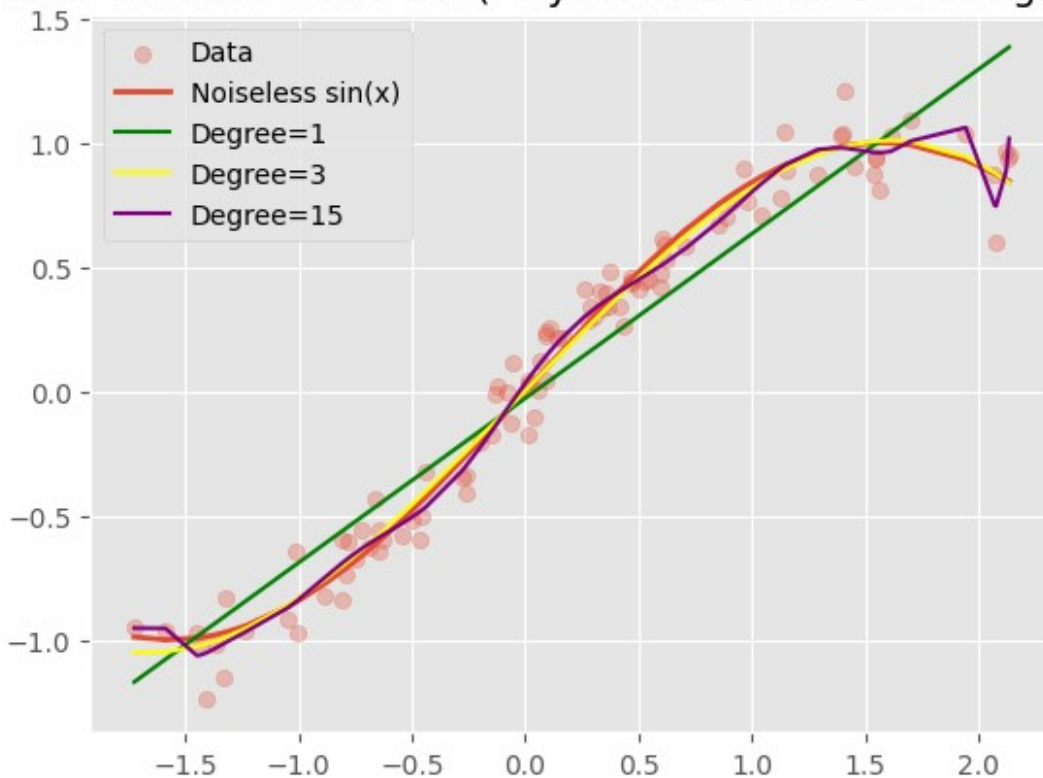
colors = ['green', 'yellow', 'purple', 'blue', 'brown']
for deg, col in zip(degrees, colors):
    poly_tmp = PolynomialFeatures(degree=deg)
    X_poly_tmp = poly_tmp.fit_transform(X[:, [0]])

    model_tmp = LinearRegression().fit(X_poly_tmp, y)
    sorted_x1_poly_tmp = poly_tmp.transform(sorted_x_1.reshape(-1, 1))

    Y_poly_tmp = model_tmp.predict(sorted_x1_poly_tmp)
    plt.plot(sorted_x_1, Y_poly_tmp, label=f"Degree={deg}",
linestyle='--', c=col)

plt.legend()
plt.title("Bias-Variance Trade-Off (Polynomials of different
degrees)")
plt.show()
```

Bias-Variance Trade-Off (Polynomials of different degrees)



we can see that the green line is too simple and will have a high bias, OTOH the purple line is too complex and will have a high variance. The yellow line serves as a good compromise between the two.

We invite the reader to experiment with different degrees of the polynomial, maybe additionally looking at train/test errors and try to find the optimal degree.

early stopping using a simple neural network

tensorflow is used for simplicity, in practice we may often prefer PyTorch

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
```

Build a simple MLP model

```
neural_network = Sequential()
neural_network.add(Dense(units=50, activation='relu',
input_shape=(X.shape[1],)))
neural_network.add(Dense(units=50, activation='relu'))
neural_network.add(Dense(units=1, activation='relu'))

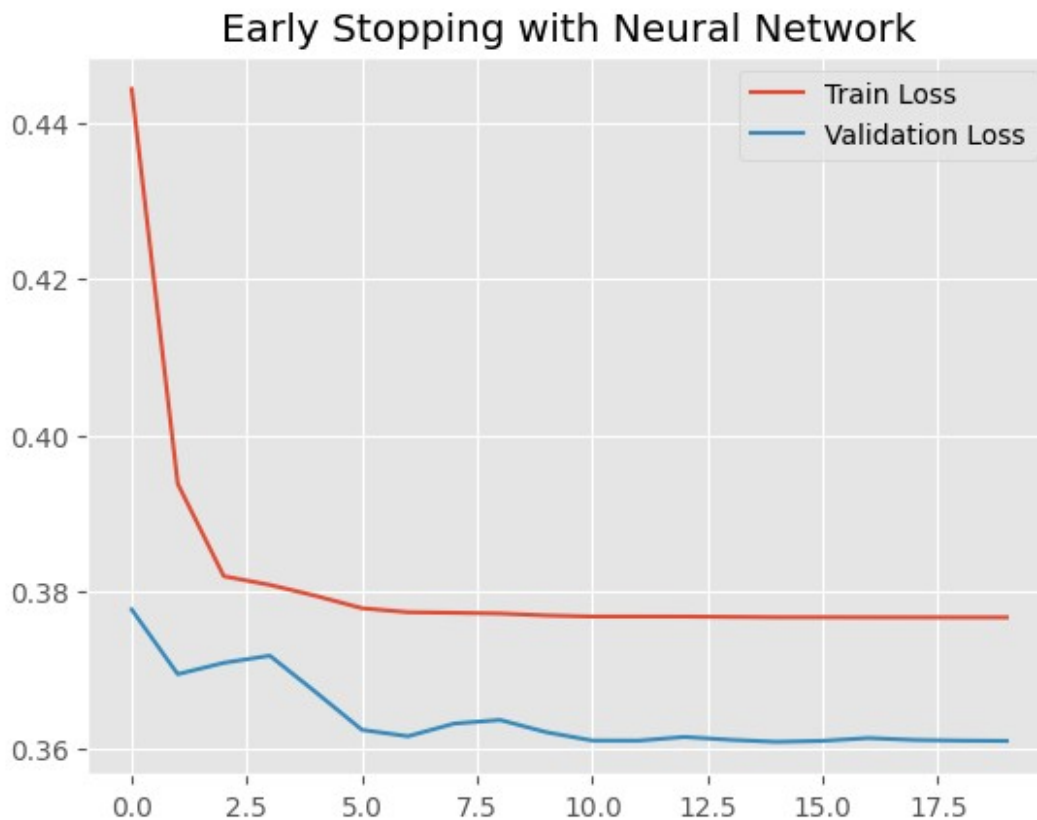
neural_network.compile(
```

```
optimizer='adam',
loss='mse',
metrics=['mse']
)

# Early stopping callback
early_stop = EarlyStopping(patience=5, restore_best_weights=True)

history = neural_network.fit(
    X,
    y,
    batch_size=24,
    epochs=100,
    validation_split=0.2,
    callbacks=[early_stop],
    verbose=0 # Set to 1 if you want to see training logs
)

# Plot the training history
plt.figure()
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.title("Early Stopping with Neural Network")
plt.show()
```



we see that the neural network ran only for ~18 epochs, although we specified 100 epochs. This is because we used early stopping and the optimizer ran for 5 epochs without significant improvement in the validation loss.

We encourage the reader to do split the data and look into train/test errors for the same neural network with and without early stopping. What do you observe regarding the overfitting?