Week 35

Kjersti Stangeland, 2025

Exercise 1 - Finding the derivative of Matrix-Vector expressions

a) Consider the expression

$$\frac{\partial(\boldsymbol{a}^T\boldsymbol{x})}{\partial \boldsymbol{x}},$$

Where $\boldsymbol{a}$ and $\boldsymbol{x}$ are column-vectors with length n.

What is the *shape* of the expression we are taking the derivative of?

What is the *shape* of the thing we are taking the derivative with respect to?

What is the *shape* of the result of the expression?

b) Show that

$$\frac{\partial(\boldsymbol{a}^T\boldsymbol{x})}{\partial \boldsymbol{x}} = \boldsymbol{a}^T,$$

Answer a & b:

We have $\boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix} (n,1)$, $\boldsymbol{a}^T = \begin{pmatrix} a_0 & a_1 & \dots & a_{n-1} \end{pmatrix} (1,n)$ and $\boldsymbol{x} = \begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_{n-1} \end{pmatrix} (n,1)$. Thus, the expression we're taking the derivative of is

$$\boldsymbol{a}^T\boldsymbol{x} = \begin{pmatrix} a_0 & a_1 & \dots & a_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} a_0x_0 + a_1x_1 + \dots + a_{n-1}x_{n-1} \end{pmatrix}$$ or more neatly $\boldsymbol{a}^T\boldsymbol{x} = \sum_{i=0}^{n-1} a_i x_i$ and is a scalar with shape $(1,1)$.

The *thing* we are taking the derivative with respect to, $\boldsymbol{x} = \begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_{n-1} \end{pmatrix}$ has shape $(n, 1)$.

The expression is taking the derivative of a scalar $(1, 1)$ with respect to a vector $(n, 1)$. That is, how does the scalar change when each component of x changes. Writing it out:

$\frac{\partial(\boldsymbol{a}^T\boldsymbol{x})}{\partial \boldsymbol{x}} = \frac{\partial}{\partial x_j}(\sum_{i=0}^{n} a_i x_i)$ where $\frac{\partial}{\partial x_j}$ is the partial derivative and $j = 0, 1, 2, \dots, n-1$. Further writing it out, $\frac{\partial}{\partial x_j}(\sum_{i=0}^{n-1} a_i x_i) = \sum_{i=0}^{n-1} a_i \frac{\partial x_i}{\partial x_j}$ where we have if $i = j$, $\frac{\partial x_i}{\partial x_j} = 1$ or $i \neq j$, $\frac{\partial x_i}{\partial x_j} = 0$. The result is then $\frac{\partial(\boldsymbol{a}^T\boldsymbol{x})}{\partial \boldsymbol{x}} = \frac{\partial}{\partial x_j}(\sum_{i=0}^{n} a_i x_i) = \sum_{i=0}^{n} a_i \frac{\partial x_i}{\partial x_j} = a_j$ and $\frac{\partial(\boldsymbol{a}^T\boldsymbol{x})}{\partial \boldsymbol{x}} = \boldsymbol{a}^T$.

Here I would argue that as you can write both $= a_j$ and a_i that the shape of the resulting expression depends on which method you use. Therefore both $(1, n)$ and $(n, 1)$ are valid shapes of the result.

c) Show that

$$\frac{\partial(\boldsymbol{a}^T\boldsymbol{A}\boldsymbol{a})}{\partial \boldsymbol{a}} = \boldsymbol{a}^T(\boldsymbol{A} + \boldsymbol{A}^T)$$

Answer c :

I assume that:

- $\boldsymbol{a}^T$ has shape $(1, n)$
- $\boldsymbol{A}$ has shape (n, n)
- $\boldsymbol{a}$ has shape $(n, 1)$

Starting with

$$\boldsymbol{a}^T\boldsymbol{A}\boldsymbol{a} = \begin{pmatrix} a_0 & a_1 & \dots & a_n - 1 \end{pmatrix} \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & \cdots & a_{1n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-10} & a_{n-11} & \cdots & a_{n-1n-1} \end{bmatrix}$$

We can define the scalar $\alpha = \boldsymbol{a}^T\boldsymbol{A}\boldsymbol{a}$ which written out, component-wise looks like: $\alpha = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i A_{ij} a_j$. Taking the derivative of α with respect to each component of $\boldsymbol{a}$,

$$\frac{\partial \alpha}{\partial a_k} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} \frac{\partial}{\partial a_k}(a_i A_{ij} a_j)$$

If $i = k$ or $j = k$ the derivative is non-zero. We can write this using the Kronecker delta:

$$\frac{\partial \alpha}{\partial a_k} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} (\delta_{ik} A_{ij} a_j + \delta_{jk} A_{ij} a_i)$$

such that if $i = k$ or $j = k$, $\delta = 1$. And as $\bolda$ is a 1-dimensional vector, the components of the transposed equal the untransposed, $a_i = a_j^T$. This leaves us with:

$$\frac{\partial \alpha}{\partial a_k} = \sum_{i=0}^{n-1} A_{ik} a_i + \sum_{j=0}^{n-1} A_{kj} a_j = \sum_{j=0}^{n-1} (A_{kj} + A_{jk}) a_j$$

and lastly back to matrix form:

$$\frac{\partial \alpha}{\partial \bolda} = \bolda^T (\boldsymbol{A} + \boldsymbol{A}^T)$$

Exercise 2 - Deriving the expression for OLS

The ordinary least squares method finds the parameters $\boldsymbol{\theta}$ which minimizes the squared error between our model $\boldsymbol{X\theta}$ and the true values $\boldsymbol{y}$.

To find the parameters $\boldsymbol{\theta}$ which minimizes this error, we take the derivative of the squared error expression with respect to $\boldsymbol{\theta}$, and set it equal to 0.

a) Very briefly explain why the approach above finds the parameters $\boldsymbol{\theta}$ which minimizes this error.

Answer a:

The approach find the parameters $\boldsymbol{\theta}$ which minimizes error because we find the minimum of the error function with respect to the parameters. The parameters are in our function to reproduce the true values. As the error function compares the true values with modelled values, which depend on the parameters, minimizing the function gives us the optimal parameters. This works as the squared error is convex, and thus finding the minimum yields the lowest error.

We typically write the squared error as

$$||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}||^2$$

which we can rewrite in matrix-vector form as

$$(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})$$

b) If $\boldsymbol{X}$ is invertible, what is the expression for the optimal parameters $\boldsymbol{\theta}$? (**Hint:** Don't compute any derivatives, but solve $\boldsymbol{X}\boldsymbol{\theta}=\boldsymbol{y}$ for $\boldsymbol{\theta}$)

Answer b:

If $\boldsymbol{X}$ is invertible we can directly solve $\boldsymbol{X}\boldsymbol{\theta}=\boldsymbol{y}$ as $\boldsymbol{\theta}=\boldsymbol{X}^{-1}\boldsymbol{y}$

c) Show that

$$\frac{\partial(\boldsymbol{x}-\boldsymbol{A}\boldsymbol{s})^T(\boldsymbol{x}-\boldsymbol{A}\boldsymbol{s})}{\partial \boldsymbol{s}}=-2(\boldsymbol{x}-\boldsymbol{A}\boldsymbol{s})^T\boldsymbol{A},$$

Answer c: First lets expand the numerator:

$$(\boldsymbol{x}-\boldsymbol{A}\boldsymbol{s})^T(\boldsymbol{x}-\boldsymbol{A}\boldsymbol{s})=\boldsymbol{x}^T\boldsymbol{x}-\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{s}-\boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{x}+\boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{A}$$

Since $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{s}$ is a scalar it is equal to its transposed. That is: $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{s}=(\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{s})^T=\boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{x}$. This leaves us with

$$(\boldsymbol{x}-\boldsymbol{A}\boldsymbol{s})^T(\boldsymbol{x}-\boldsymbol{A}\boldsymbol{s})=\boldsymbol{x}^T\boldsymbol{x}-2\boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{x}+\boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{s}$$

Now we can derivate each term with respect to $\boldsymbol{s}$. The first term drops out as it is independent of $\boldsymbol{s}$.

$$-2\frac{\partial}{\partial \boldsymbol{s}}(\boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{x})+\frac{\partial}{\partial \boldsymbol{s}}(\boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{s})=-2(\boldsymbol{A}^T\boldsymbol{x})^T+2\boldsymbol{s}$$

d) Using the expression from **c)**, but substituting back in $\boldsymbol{\theta}$, $\boldsymbol{y}$ and $\boldsymbol{X}$, find the expression for the optimal parameters $\boldsymbol{\theta}$ in the case that $\boldsymbol{X}$ is not invertible, but $\boldsymbol{X}^T\boldsymbol{X}$ is, which is most often the case.

$$\hat{\boldsymbol{\theta}}_{\boldsymbol{OLS}}=\dots$$

Answer d: Inserting into the expression above:

$$\hat{\boldsymbol{\theta}}_{\boldsymbol{OLS}}=\frac{\partial(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}=-2(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T\boldsymbol{X}=0$$

Then we can solve for $\hat{\boldsymbol{\theta}}$:

$$(\boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\theta}})^T \boldsymbol{X} = 0$$

As $\boldsymbol{X}$ is not invertible, but $\boldsymbol{X^T X}$ is, we can transpose the equation, solve the paranthesis and invert $\boldsymbol{X^T X}$ to find a solution.

$$\boldsymbol{X^T}(\boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\theta}}) = 0$$

$$\boldsymbol{X^T y} - \boldsymbol{X^T X}\,\hat{\boldsymbol{\theta}} = 0$$

$$\boldsymbol{X^T X}\,\hat{\boldsymbol{\theta}} = \boldsymbol{X^T y}$$

$$\hat{\boldsymbol{\theta}}_{OLS} = (\boldsymbol{X^T X})^{-1} \boldsymbol{X^T y}$$

Exercise 3 - Creating feature matrix and implementing OLS using the analytical expression

With the expression for $\hat{\boldsymbol{\theta}}_{\boldsymbol{OLS}}$, you now have what you need to implement OLS regression with your input data and target data $\boldsymbol{y}$. But before you can do that, you need to set up you input data as a feature matrix $\boldsymbol{X}$.

In a feature matrix, each row is a datapoint and each column is a feature of that data. If you want to predict someones spending based on their income and number of children, for instance, you would create a row for each person in your dataset, with the montly income and the number of children as columns.

We typically also include an intercept in our models. The intercept is a value that is added to our prediction regardless of the value of the other features. The intercept tries to account for constant effects in our data that are not dependant on anything else. In our current example, the intercept could account for living expenses which are typical regardless of income or childcare expenses.

We calculate the optimal intercept by including a feature with the constant value of 1 in our model, which is then multplied by some parameter θ_0 from the OLS method into the optimal intercept value (which will be θ_0). In practice, we include the intercept in our model by adding a column of ones to the start of our feature matrix.

In [54]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [55]:
```
n = 20
income = np.array([116., 161., 167., 118., 172., 163., 179., 173., 162.,
children = np.array([5, 3, 0, 4, 5, 3, 0, 4, 4, 3, 3, 5, 1, 0, 2, 3, 2, 1
spending = np.array([152., 141., 102., 136., 161., 129.,  99., 159., 160.
```

a) Create a feature matrix $\boldsymbol{X}$ for the features income and children, including an

intercept column of ones at the start.

In [56]:
```
X = np.zeros((n, 3))
X[:, 0] = 1
X[:, 1] = income
X[:, 2] = children
```

b) Use the expression from **3d)** to find the optimal parameters $\hat{\boldsymbol{\beta}}_{\boldsymbol{OLS}}$ for predicting spending based on these features. Create a function for this operation, as you are going to need to use it a lot.

$$\hat{\boldsymbol{\beta}}_{OLS} = (\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}$$

In [57]:
```
def OLS_parameters(X, y):
    X_T = np.transpose(X)
    X_T_X = X_T @ X

    return np.linalg.inv(X_T_X) @ X_T @ y

beta = OLS_parameters(X, spending)
```

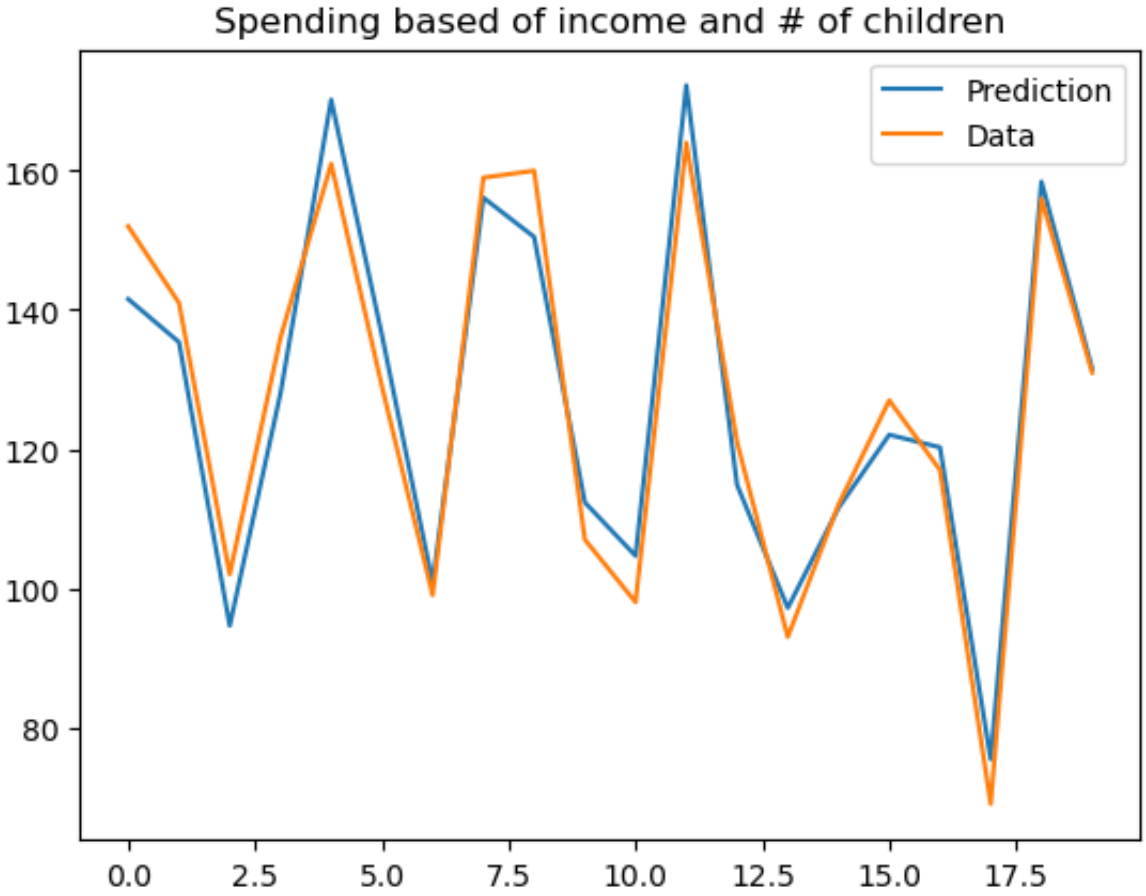
In [58]:
```
beta
```

Out[58]:
```
array([ 9.12808583,  0.5119025 , 14.60743095])
```

In [59]:
```
predict = X @ beta
```

In [60]:
```
plt.plot(predict, label='Prediction')
plt.plot(spending, label='Data')
plt.title('Spending based of income and # of children')
plt.legend()
```

Out[60]:
```
<matplotlib.legend.Legend at 0x16a16c410>
```



Exercise 4 - Fitting a polynomial

In this course, we typically do linear regression using polynomials, though in real world applications it is also very common to make linear models based on measured features like you did in the previous exercise.

When fitting a polynomial with linear regression, we make each polynomial degree($x, x^2, x^3, \ldots, x^p$) its own feature.

In [61]:
```
n = 100
x = np.linspace(-3, 3, n)
y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2) + np.random.normal(0, 0.1)
```

a) Create a feature matrix $\boldsymbol{X}$ for the features x, x^2, x^3, x^4, x^5, including an intercept column of ones at the start. Make this into a function, as you will do this a lot over the next weeks.

In [62]:
```
def polynomial_features(x, p):
    n = len(x)
    X = np.zeros((n, p + 1))
    X[:, 0] = 1
    for i in range(1, p+1):
        X[:, i] = x**i
```

```
    return X

X = polynomial_features(x, 5)
```

b) Use the expression from **3d)** to find the optimal parameters $\hat{\beta}_{OLS}$ for predicting y based on these features. If you have done everything right so far, this code will not need changing.

In [63]:
```
beta = OLS_parameters(X, y)
```

c) Like in exercise 4 last week, split your feature matrix and target data into a training split and test split.

In [64]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

d) Train your model on the training data(find the parameters which best fit) and compute the MSE on both the training and test data.

In [65]:
```
beta_train = OLS_parameters(X_train, y_train)

train_predict = X_train @ beta_train
test_predict = X_test @ beta_train
```

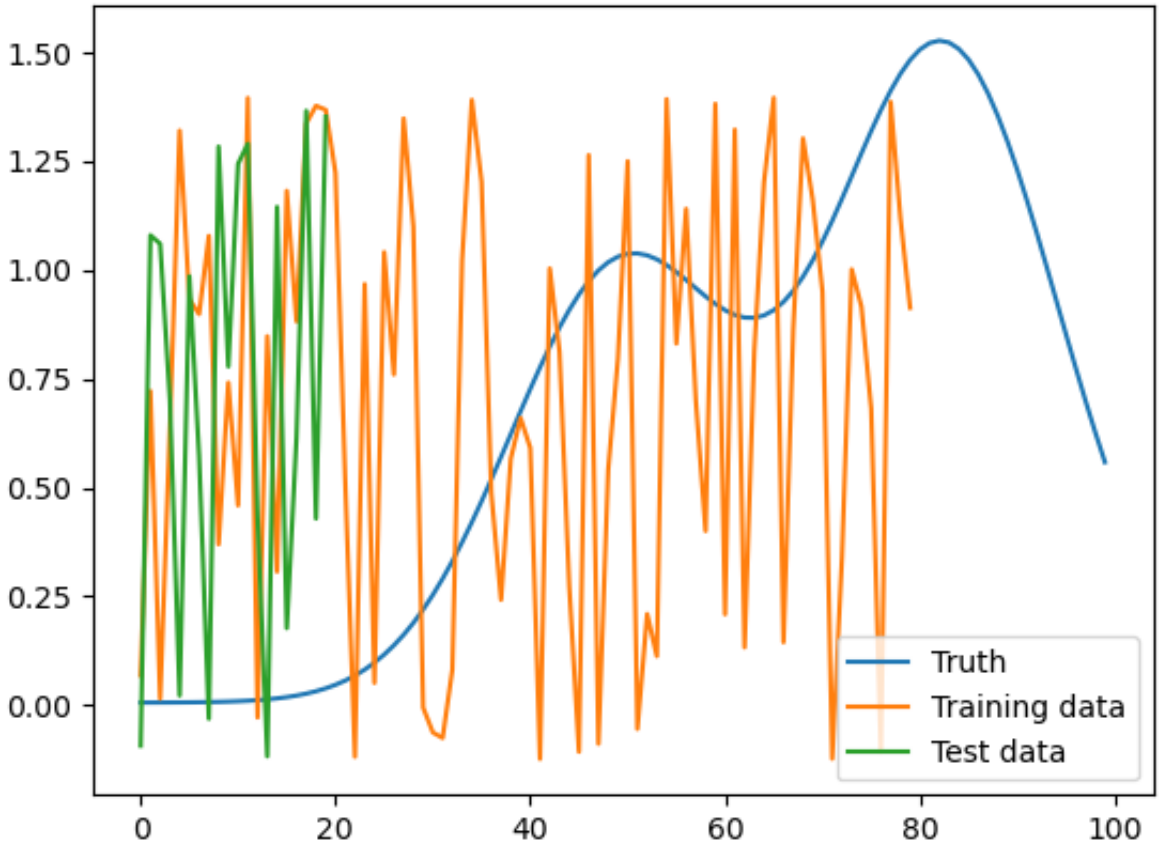
In [66]:
```
def MSE(y_data, y_pred):
    return np.mean((y_data - y_pred)**2)
```

In [67]:
```
mse_train = MSE(y_train, train_predict)
mse_test = MSE(y_test, test_predict)

print(f'Training data prediction MSE: {mse_train}')
print(f'Test data prediciton MSE: {mse_test}')
```

```
Training data prediction MSE: 0.01435969965684041
Test data prediciton MSE: 0.011731037657083932
```

In [68]:
```
plt.plot(y, label='Truth')
plt.plot(train_predict, label='Training data')
plt.plot(test_predict, label='Test data')
plt.legend()
```

Out[68]:
```
<matplotlib.legend.Legend at 0x16a1f0190>
```



Well that sucked!

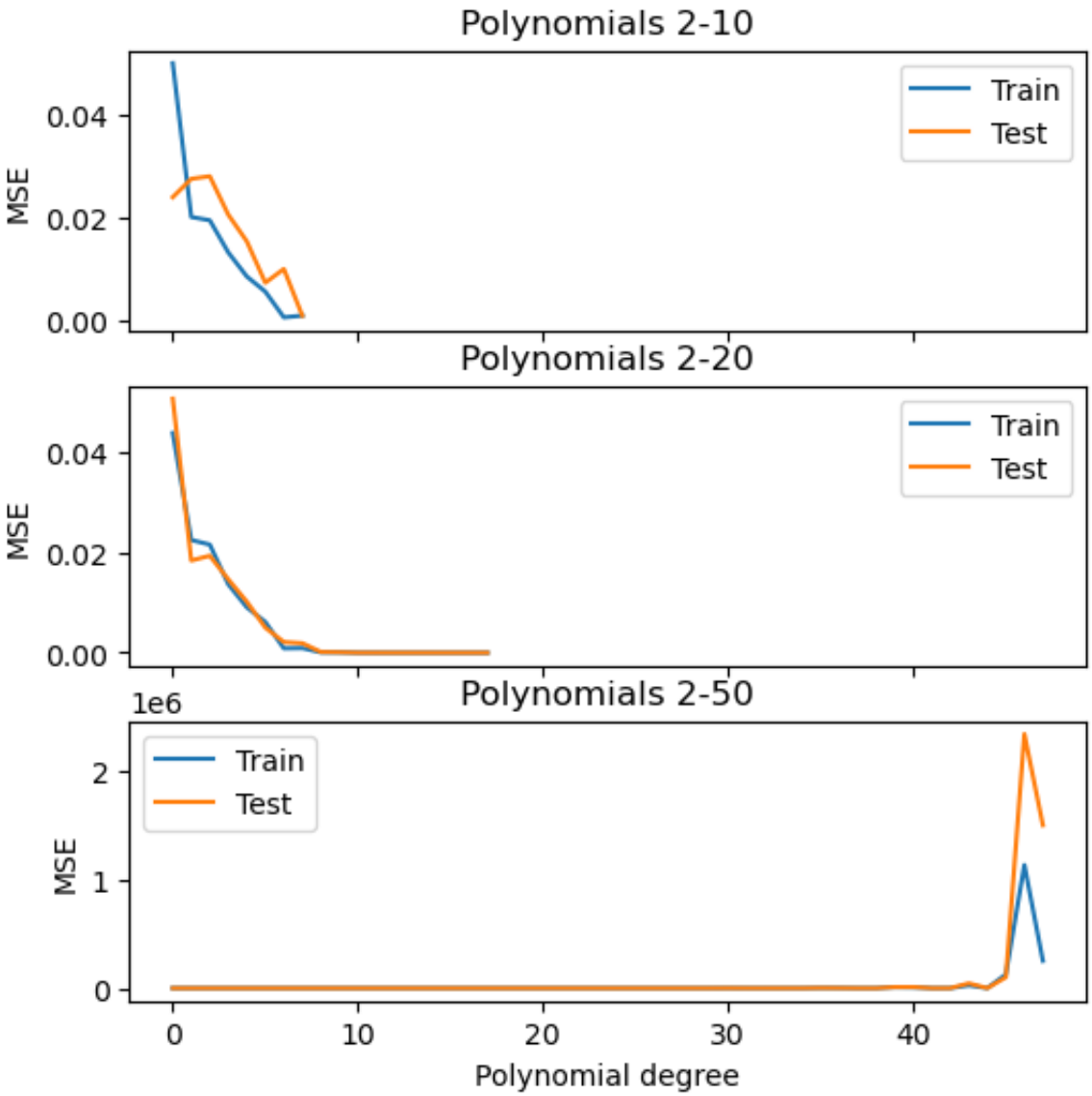
e) Do the same for each polynomial degree from 2 to 10, and plot the MSE on both the training and test data as a function of polynomial degree. The aim is to reproduce Figure 2.11 of Hastie et al. Feel free to read the discussions leading to figure 2.11 of Hastie et al.

In [ ]:
```
def model_complexity_test(s):
    degrees = np.arange(2, s+1)

    mse_trains = []
    mse_tests = []

    for d in degrees:
        # Feature matrix with of a given poly. degree
        X_p = polynomial_features(x, d)

        # Splitting data
        X_train, X_test, y_train, y_test = train_test_split(X_p, y, test_

        # Training parameters
        beta_train = OLS_parameters(X_train, y_train)

        # Training the model
        train_predict = X_train @ beta_train
        test_predict = X_test @ beta_train
```

```
        # Computing MSE and storing it in a list
        mse_trains.append(MSE(y_train, train_predict))
        mse_tests.append(MSE(y_test, test_predict))

    return mse_trains, mse_tests
```

In [82]:
```
mse_trains_10, mse_tests_10 = model_complexity_test(10)  # MSE for polyno
mse_trains_20, mse_tests_20 = model_complexity_test(20)  # MSE for polyno
mse_trains_50, mse_tests_50 = model_complexity_test(50)  # MSE for polyno
```

In [92]:
```
fig, ax = plt.subplots(3, 1, figsize=(6, 6), sharex=True)
degrees = np.arange(2, 51)

ax[0].plot(mse_trains_10, label='Train')
ax[0].plot(mse_tests_10, label='Test')
ax[0].set_title('Polynomials 2-10')

ax[1].plot(mse_trains_20, label='Train')
ax[1].plot(mse_tests_20, label='Test')
ax[1].set_title('Polynomials 2-20')

ax[2].plot(mse_trains_50, label='Train')
ax[2].plot(mse_tests_50, label='Test')
ax[2].set_title('Polynomials 2-50')

ax[2].set_xlabel('Polynomial degree')

for axs in ax:
    axs.set_ylabel('MSE')
    axs.legend()
```



f) Interpret the graph. Why do the lines move as they do? What does it tell us about model performance and generalizability?

I freestyled a bit as I didnt really see a similar response as in fig 2.11 in Hastie et al. when only trying order 2-10.

MSE decreases with increasing degree of the polynomial to a certain point. The region where the test error is still decreasing is the optimal model range for the given number of data points. As the test prediction has not seen the training data, it does not know of the outliers or noises in the dataset and thus the error increases (???). The training error shows a similar behaviour, where at a certain point over-fitting occurs with a high degree of model complexity.

Low degree polynomials under-fit the data, aka it is too simple to fit the data. A too high degree polynomial results in over-fitting, as the model is trained on very precisely capturing the training data and thus fails to fit unseen data in, as seen in the testing error.

Exercise 5 - Comparing your code with sklearn

When implementing different algorithms for the first time, it can be helpful to double check your results with established implementations before you go on to add more complexity.

a) Make sure your `polynomial_features` function creates the same feature matrix as sklearns PolynomialFeatures.

(https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html)

In [93]:
```
from sklearn.preprocessing import PolynomialFeatures
```

In [121…
```
X_mine = polynomial_features(x, 3)

pol_feat = PolynomialFeatures(3)
X_sklearn = pol_feat.fit_transform(x.reshape(-1, 1))

print(f'Are they the same? {np.allclose(X_mine, X_sklearn)}')
```

```
Are they the same? True
```

b) Make sure your `OLS_parameters` function computes the same parameters as sklearns LinearRegression with fit_intercept set to False, since the intercept is included in the feature matrix. Use `your_model_object.coef_` to extract the computed parameters.

(https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)

In [122…
```
from sklearn.linear_model import LinearRegression
```

In [123…
```
my_beta = OLS_parameters(X_mine, y)

sk_model = LinearRegression(fit_intercept=False)
sk_model.fit(X_mine, y)
beta_sklearn = sk_model.coef_

# Compare
print("Are the parameters equal?", np.allclose(my_beta, beta_sklearn))
```

```
Are the parameters equal? True
```