Week 35

Exercise 1

I start by writing the Jacobi-matrix which is going to be a 1xn matrix/vector assuming the vectors' length are n, because $\mathbf{b}^T\mathbf{a}$ is a scalar:

$$\frac{\partial(\mathbf{b}^T\mathbf{a})}{\partial\mathbf{a}} = \mathbf{J} = \begin{pmatrix} \frac{\partial(\mathbf{b}^T\mathbf{a})}{\partial a_0} & \dots & \frac{\partial(\mathbf{b}^T\mathbf{a})}{\partial a_{n-1}} \end{pmatrix}$$

I then write the scalar as a sum of the vector elements:

$$= \begin{pmatrix} \frac{\partial \sum_{i=0}^{n-1} b_i a_i}{\partial a_0} & \dots & \frac{\partial \sum_{i=0}^{n-1} b_i a_i}{\partial a_{n-1}} \end{pmatrix}$$

I can then easily differentiate the terms seperately, and all the elements which dont include the a_i-term being differentiated are easily discarded:

$$= \begin{pmatrix} b_0 & b_1 & \dots & b_{n-1} \end{pmatrix} = \underline{\mathbf{b}} \tag{1}$$

We do the same for the next task, because $\mathbf{a}^T\mathbf{A}\mathbf{a}$ is also a scalar:

$$\frac{\partial(\mathbf{a}^T\mathbf{A}\mathbf{a})}{\partial\mathbf{a}} = \mathbf{J} = \begin{pmatrix} \frac{\partial(\mathbf{a}^T\mathbf{A}\mathbf{a})}{\partial a_0} & \dots$$

$$= \begin{pmatrix} \frac{\partial(\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i A_{ij} a_j)}{\partial a_0} & \dots & \underline{\partial(\sum_{i=}^{n-}$$

$$= \begin{pmatrix} \frac{\partial}{\partial a_0}[(a_0(A_{00}a_0 + A_{01}a_1 + \dots + A_{0(n-1)}a_{n-1}) + a_1(A_{10}a_0 + A_{11}a_1 + \dots + A_{1(n-1)}a_{n-1}$$

the difference is that now each vector-element will have two sums after differentiating (this is hard to follow and even harder to write correctly):

$$= \begin{pmatrix} [2a_0A_{00} + a_1A_{01} + \dots + a_{n-1}A_{0(n-1)}] + [a_1A_{10} + \dots + a_{n-1}A_{(n-1)0}] & \dots & [a_0A_{(n-1)0}$$

$$= \begin{pmatrix} a_0A_{00} + \sum_{i=0}^{n-1} a_iA_{1i} + \sum_{j=1}^{n-1} a_jA_{j1} & \dots & a_{n-1}A_{(n-1)}$$

We slump the extra terms into the second sums \begin{gather*} =

$$\begin{pmatrix} \sum_{i=0}^{n-1} a_iA_{1i} + \sum_{j=0}^{n-1} a_jA_{j1} & \dots & \sum_{i=0}^{n-1} a_iA_{(n-1)i} + \sum_{j=0}^{n-1} a_jA_{j(n-1)} \end{pmatrix}$$

\end{gather} here we can see that the $\mathbf{a}$-terms are multiplied into both these two terms for each element, meaning we can identify these sums like shown in example 3 of the weeks lecture notes:

$$= \underline{\mathbf{a}^T(\mathbf{A} + \mathbf{A}^T)} \tag{2}$$

For the next task we recognize that $(\mathbf{x} - \mathbf{A}\mathbf{s})^T(\mathbf{x} - \mathbf{A}\mathbf{s})$ also is a scalar like before, if $\mathbf{A}$

is a quadratic nxn matrix. We start the same:

$$\frac{\partial(\mathbf{x}-\mathbf{A}\mathbf{s})^T(\mathbf{x}-\mathbf{A}\mathbf{s})}{\partial \mathbf{s}} = \mathbf{J} = \begin{pmatrix} \frac{\partial(\mathbf{x}-\mathbf{A}\mathbf{s})^T(\mathbf{x}-\mathbf{A}\mathbf{s})}{\partial s_0} & \cdots & \frac{\partial(\mathbf{x}-\mathbf{A}\mathbf{s})^T(\mathbf{x}-\mathbf{A}\mathbf{s})}{\partial s_{n-1}} \end{pmatrix}$$
$$= \begin{pmatrix} \frac{\partial(\sum_{i=0}^{n-1}(x_i-\sum_{j=0}^{n-1}A_{ij}s_j))^T(\sum_{i=0}^{n-1}(x_i-\sum_{j=0}^{n-1}A_{ij}s_j))}{\partial s_0} & \cdots & \frac{\partial(\sum_{i=0}^{n-1}(x_i-\sum_{j=0}^{n-1}A_{ij}s_j))^T(\sum_{i=0}^{n-1}(x_i-\sum_{j=0}^{n-1}A_{ij}s_j}{\partial s_{n-1}} \end{pmatrix}$$

I stared at this for quite some while, but I really just couldn't figure out the next step and I found it quite difficult to not "lose my tongue".

Exercise 2

Own code:

In [1]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Data
n = 100  # no. data points
x = np.random.rand(n, 1)
y = 2.0 + 5 * x * x + 0.1 * np.random.randn(n, 1)

# Own code
X = np.zeros((n, 3))
X[:, 0] = 1
X[:, 1] = x[:, 0]
X[:, 2] = x[:, 0]**2

beta = np.linalg.inv(X.T @ X) @ X.T @ y
y_tilde = X @ beta

# Metrics
print("OWN CODE\n----------------------")
print(f"MSE: {mean_squared_error(y, y_tilde):g}")
print(f"R2: {r2_score(y, y_tilde):g}")
```

```
OWN CODE
----------------------
MSE: 0.00812078
R2: 0.996574
```

Scikit-learn:

In [2]:
```
linreg = LinearRegression(fit_intercept=False)
linreg.fit(X, y)
y_skl = linreg.predict(X)

# Metrics
print("SCIKIT-LEARN\n---------------------")
print(f"MSE: {mean_squared_error(y, y_skl):g}")
print(f"R2: {r2_score(y, y_skl):g}")
```

```
SCIKIT-LEARN
---------------------
MSE: 0.00812078
R2: 0.996574
```

Exercise 3

Ex3a)

In [17]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression

# Data set
np.random.seed()
n = 100
maxdegree = 6  # 5th degree polynomial plus intercept
x = np.linspace(-3, 3, n).reshape(-1, 1)
y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2) + np.random.normal(0, 0.1, x.shape)

# Create design matrix
X = np.zeros((n, maxdegree))
for degree in range(maxdegree):
    X[:, degree] = x[:, 0]**degree

# Split into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Ex3b)

In [23]:
```
# Linear regression
beta = np.linalg.inv(X_train.T @ X_train) @ X_train.T @ y_train
y_tilde = X_train @ beta
y_predict = X_test @ beta

# Scikit
linreg.fit(X_train, y_train)
y_skl = linreg.predict(X_train)
y_skl_pred = linreg.predict(X_test)


# Metrics
print("OWN CODE\n----------------------")
print(f"Train MSE: {mean_squared_error(y_train, y_tilde):g}")
print(f"Test MSE: {mean_squared_error(y_test, y_predict):g}")
```

```
print("\nSCIKIT-LEARN\n--------------------")
print(f"Train MSE: {mean_squared_error(y_train, y_skl):g}")
print(f"Test MSE: {mean_squared_error(y_test, y_skl_pred):g}")
```

```
OWN CODE
----------------------
Train MSE: 0.25189
Test MSE: 0.303889

SCIKIT-LEARN
--------------------
Train MSE: 0.25189
Test MSE: 0.303889
```

Ex3c)

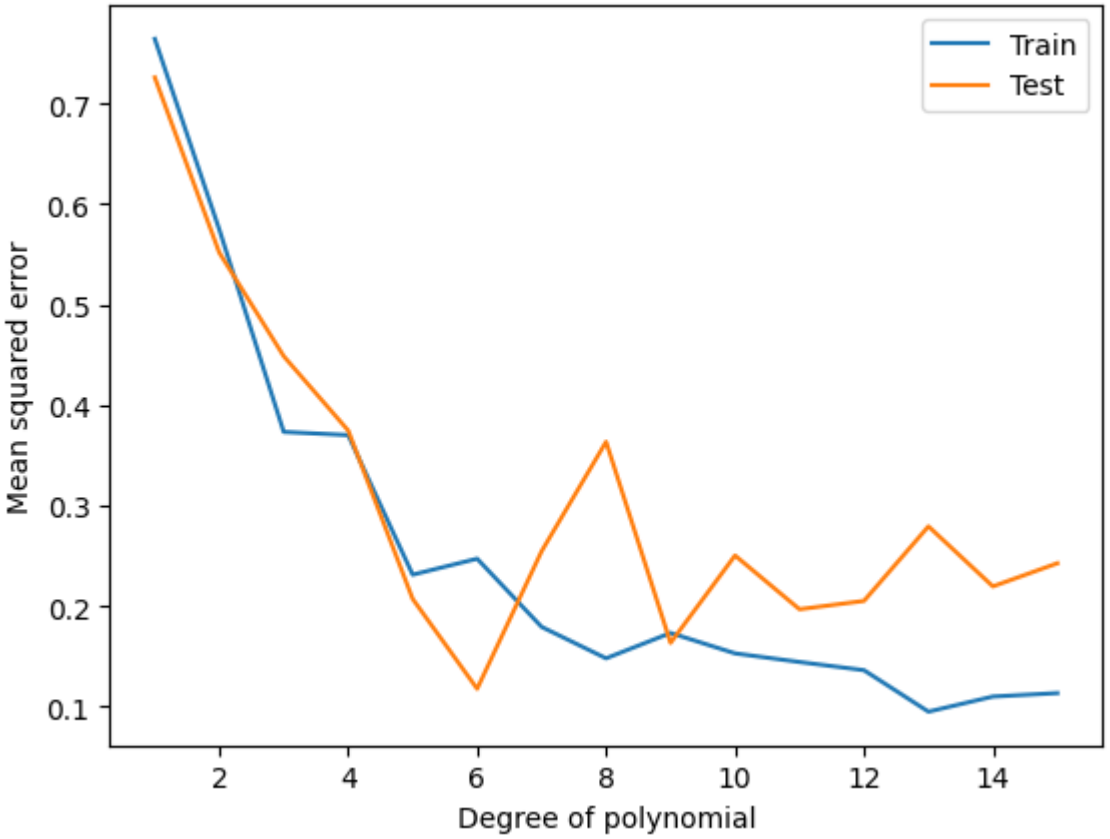
In [32]:
```
maxdegree = 16  # 15th degree polynomial plus intercept
degrees = np.array(range(1, maxdegree)) # loop from 1 to 15 degrees.
train_mse = []
test_mse = []

# Create design matrix first
X = np.zeros((n, maxdegree))
for degree in degrees:
    X[:, degree] = x[:, 0]**degree

# Then loop over the degrees again to do the mean error calculations
for degree in degrees:
    # Split into training and test data
    X_train, X_test, y_train, y_test = train_test_split(X[:, :degree], y, test_s
    linreg.fit(X_train, y_train)
    y_tilde = linreg.predict(X_train)
    y_predict = linreg.predict(X_test)
    train_mse.append(mean_squared_error(y_train, y_tilde))
    test_mse.append(mean_squared_error(y_test, y_predict))

# Plotting
plt.plot(degrees, train_mse, label="Train")
plt.plot(degrees, test_mse, label="Test")
plt.legend()
plt.xlabel("Degree of polynomial")
plt.ylabel("Mean squared error")
```

Out[32]:
```
Text(0, 0.5, 'Mean squared error')
```



This plot shows very different for every time the randomized data is run, but the common overarching feature is that the training MSE and testing MSE diverge with increasing polynomial degree. This is exactly what is shown in the idealized plot figure 2.11 of Hastie et al. Idealized we should see the optimal training MSE at the highest degree since, however in this exact configuration my plot shows it at 13th polynomial degree, and the testing MSE at the 6th degree. Idealized, from figure 2.11, we should see the optimal testing MSE around the point where the two errors diverge, this is a sign of overfitting our data.