Week35

August 29, 2025

Exercise 1a) We have the expression $f(x) = \mathbf{a^T x}$ where a and x are column vectors of length n. $f(x)$ is therefore a scalar. x has dimensions (n, 1) The result (gradient) inherit transposed x dimentions, (1, n)

Exercise 1b) We write $\mathbf{a^T x} = \sum_{j=0}^{n-1} a_j x_j$. The partial derivative with respect to x_i is

$$\frac{\partial}{\partial x_i}\left(\sum_{j=0}^{n-1} a_j x_j\right) = a_i$$

Since only the $i = j$ term contributes, this holds for all components i, so the gradient is $(a_1, a_2, ..., a_n)$ which is $\mathbf{a}^T$.

Exercise 1c) $\mathbf{a^T A a} = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} a_j A_{ij} a_i$ For $i = k$: $A_{kj} a_j$ contributes For $j = k$: $a_i A_{ik}$ contributes Taking $\frac{\partial}{\partial a_k}$, we get

$$\frac{\partial}{\partial a_k}\left(\mathbf{a^T A a}\right) = \sum_{j=0}^{n-1} A_{kj} a_j + \sum_{i=0}^{n-1} a_i A_{ik}$$

The first sum is the k th component of $(\mathbf{A}\boldsymbol{a})$, and the second sum is the k th component of $(\boldsymbol{a}^T \mathbf{A})^T = (\mathbf{A}^T \boldsymbol{a})$.

$$(\mathbf{A a})^T + \left(\mathbf{A}^T \mathbf{a}\right)^T = \mathbf{a}^T \mathbf{A}^T + \mathbf{a}^T \mathbf{A} = \mathbf{a}^T \left(\mathbf{A} + \mathbf{A}^T\right)$$

Exercise 2a) Taking the derivative of the error with respect to $\boldsymbol{\theta}$ and setting it to zero finds the minimum because we are dealing with a convex quadratic cost function. The squared error $||\mathbf{y} - \mathbf{X}\boldsymbol{\theta}||^2$ has a single global minimum. Setting the gradient to zero gives the critical point, which for a convex function is the global minimum. In short, solving $\nabla_\theta ||y - X\theta||^2 = 0$ gives the $\boldsymbol{\theta}$ that minimizes the error.

Exercise 2b) If $\mathbf{X}$ is square and invertible, the optimal solution is the one that gives zero error by exactly solving $\mathbf{X}\boldsymbol{\theta} = \mathbf{y}$. In that case, we can multiply both sides by $\mathbf{X}^{-1}$ to get

$$\boldsymbol{\theta} = \mathbf{X}^{-1}\mathbf{y}$$

Exercise 2c)

$$f(s) = (\mathbf{x} - \mathbf{A s})^T (\mathbf{x} - \mathbf{A s})$$

We differentiate $f(s)$ with respect to $\boldsymbol{s}$. This expression is the squared norm of the error vector $\boldsymbol{e} = \boldsymbol{x} - \mathbf{A}\boldsymbol{s}$.

$$\frac{\partial}{\partial s}\left(\mathbf{e}^T \mathbf{e}\right) = 2\mathbf{e}^T \frac{\partial e}{\partial s}$$

Here $\boldsymbol{e} = \boldsymbol{x} - \mathbf{A}\boldsymbol{s}$, so $\frac{\partial \boldsymbol{e}}{\partial \boldsymbol{s}} = -\mathbf{A}$

$$\frac{\partial}{\partial s}\left(\mathbf{e}^T\mathbf{e}\right) = 2\mathbf{e}^T\left(-\mathbf{A}\right) = -2\left(\mathbf{x} - \mathbf{A}\mathbf{s}\right)^T\mathbf{A}$$

Exercise 2d) In OLS, the error vector is $\boldsymbol{y} - \mathbf{X}\boldsymbol{\theta}$. Using the formula above, and substituting $\mathbf{x} = \mathbf{y}$, $\mathbf{A} = \mathbf{X}$ and $\mathbf{s} = \theta$, we get

$$-2\left(\mathbf{y} - \mathbf{X}\theta\right)^T\mathbf{X}.$$

Optimizing this gives

$$\left(\mathbf{y} - \mathbf{X}\theta\right)^T\mathbf{X} = 0^T$$
$$\mathbf{X}^T\left(\mathbf{y} - \mathbf{X}\theta\right) = 0$$
$$\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{X}\theta = 0$$
$$\mathbf{X}^T\mathbf{y} = \mathbf{X}^T\mathbf{X}\theta$$

Assuming $\mathbf{X}^T\mathbf{X}$ is invertible, we multiply by $(\mathbf{X}^T\mathbf{X})^{-1}$ to get

$$\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Exercise 3a)

[ ]:
```
import numpy as np
```

[ ]:
```
n = 20
income = np.array([116., 161., 167., 118., 172., 163., 179., 173., 162.,
                   116., 101., 176., 178., 172., 143., 135., 160., 101., 149.,␣
 ↪125.])
children = np.array([5, 3, 0, 4, 5, 3, 0, 4, 4, 3, 3, 5, 1, 0, 2, 3, 2, 1, 5,␣
 ↪4])
spending = np.array([152., 141., 102., 136., 161., 129.,  99., 159., 160.,
                     107.,  98., 164., 121.,  93., 112., 127., 117.,  69., 156.
 ↪, 131.])
```

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

[ ]:
```
print("Controlling the feature matrix X:")
print(X[:5])
print("Shape of X:", X.shape)
```

```
Controlling the feature matrix X:
[[  1. 116.   5.]
 [  1. 161.   3.]
 [  1. 167.   0.]
 [  1. 118.   4.]
 [  1. 172.   5.]]
Shape of X: (20, 3)
```

Exercise 3b)

```
[ ]: def OLS_parameters(X, y):
         return np.linalg.inv(X.T @ X) @ (X.T @ y)
     beta = OLS_parameters(X, spending)
     print("Optimal OLS parameters beta:", beta)
```

```
Optimal OLS parameters beta: [ 9.12808583  0.5119025  14.60743095]
```

The model would then be spending $\approx 9.13 + 0.512 \cdot \text{income} + 14.61 \cdot \text{children}$

Exercise 4a)

```
[ ]: 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)
```

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

     X_test = polynomial_features(x, 5)
     print("X_test shape:", X_test.shape)
     print("X_test first row:", X_test[0])
```

```
X_test shape: (100, 6)
X_test first row: [   1.   -3.    9.  -27.   81. -243.]
```

Exercise 4b)

```
[ ]: beta_test = OLS_parameters(X_test, y)
     print("Beta coefficients for polynomial degree 5:", beta_test)
```

```
Beta coefficients for polynomial degree 5: [ 0.84188633  0.27464654 -0.02326439
0.05342623 -0.0034652  -0.0087781 ]
```

Exercise 4c)

```
[ ]: from sklearn.model_selection import train_test_split
     X_train, X_test, y_train, y_test = train_test_split(X_test, y, test_size=0.2,␣
      ↪random_state=42)
     print("Training set size:", X_train.shape[0], "Test set size:", X_test.shape[0])
```

```
Training set size: 80 Test set size: 20
```

Exercise 4d)

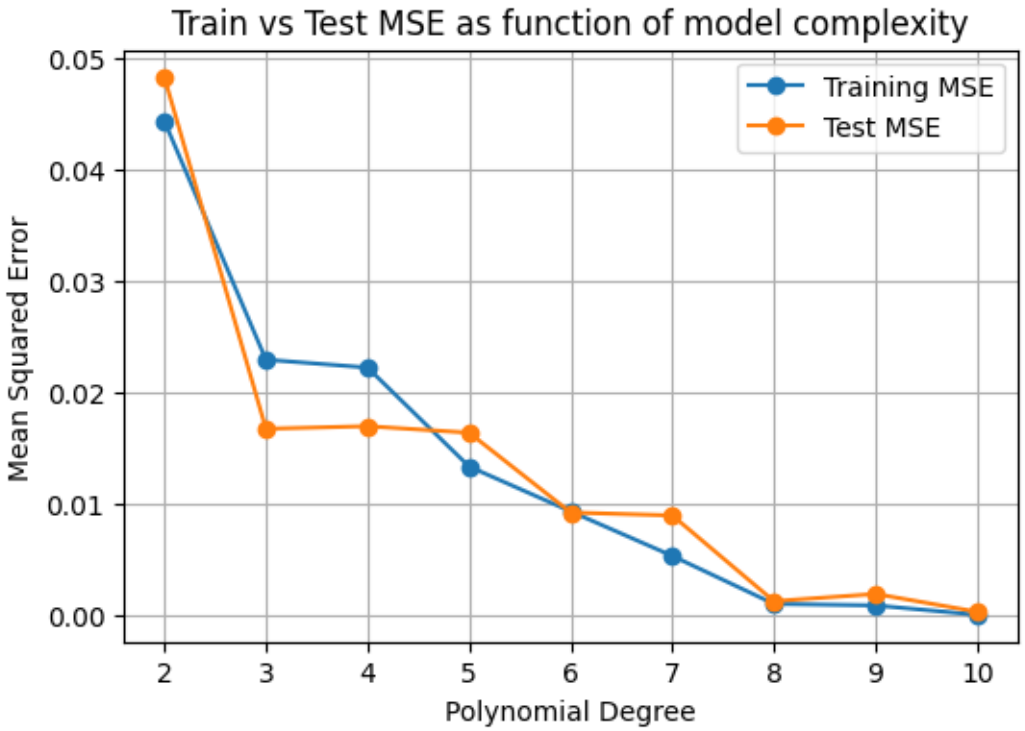
[ ]:
```
beta_train = OLS_parameters(X_train, y_train)

# Predictions for training and test sets
y_train_pred = X_train @ beta_train
y_test_pred  = X_test @ beta_train

# Calculate Mean Squared Error for train and test
MSE_train = np.mean((y_train - y_train_pred)**2)
MSE_test  = np.mean((y_test  - y_test_pred)**2)
print("Training MSE :", MSE_train)
print("Test MSE     :", MSE_test)
```

```
Training MSE : 0.013358669300733142
Test MSE     : 0.016393199898149562
```

Exercise 4e)

[ ]:
```
import matplotlib.pyplot as plt
degrees = range(2, 11)
train_errors = []
test_errors = []

# We'll reuse the same train/test indices for all degrees for consistency
idx_all = np.arange(n)
idx_train, idx_test = train_test_split(idx_all, test_size=0.2, random_state=42)

for p in degrees:
    Xp = polynomial_features(x, p)
    Xp_train = Xp[idx_train]
    Xp_test  = Xp[idx_test]
    y_train_split = y[idx_train]
    y_test_split  = y[idx_test]
    beta_p = OLS_parameters(Xp_train, y_train_split)
    train_err = np.mean((y_train_split - Xp_train @ beta_p)**2)
    test_err  = np.mean((y_test_split  - Xp_test  @ beta_p)**2)
    train_errors.append(train_err)
    test_errors.append(test_err)



plt.figure(figsize=(6,4))
plt.plot(list(degrees), train_errors, marker='o', label='Training MSE')
plt.plot(list(degrees), test_errors,  marker='o', label='Test MSE')
plt.xlabel("Polynomial Degree")
plt.ylabel("Mean Squared Error")
plt.legend()
plt.title("Train vs Test MSE as function of model complexity")
plt.xticks(list(degrees))
```

```
plt.grid(True)
plt.show()
```



Exercise 4f)

The training data decreases steadily with polynomial degree, as expected. The test data initially drops quickly, then levels off. Up to degree 10 there is no strong sign of overfitting. This suggests that the data set is relatively smooth and the amount of noise is low. The models of higher degree are not yet "too flexible" for the available sample size, so they still generalize well. The bias is reduced as degree increases, and the variance is not to high to dominate. To observe the "U-shape" that appears in Hastie et al. in test error, we would likely need to extend the experiment to much higher degrees, where overfitting to noise becomes more pronounced.

Exercise 5a)

[ ]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
poly = PolynomialFeatures(degree=5, include_bias=True)
X_skl = poly.fit_transform(x.reshape(-1, 1))
X_custom = polynomial_features(x, 5)
print("Matrices equal:", np.allclose(X_custom, X_skl))
```

```
Matrices equal: True
```

Exercise 5b)

[ ]:
```
X_new = polynomial_features(x, 5)
y_new = y

beta_manual = OLS_parameters(X_new, y_new)

model = LinearRegression(fit_intercept=False)
model.fit(X_new, y_new)
beta_sklearn = model.coef_

print("Our OLS beta coefficients:     ", np.round(beta_manual, 6))
print("Sklearn LinearRegression beta:", np.round(beta_sklearn, 6))
print("Difference (our - sklearn):   ", np.round(beta_manual - beta_sklearn,␣
 ↪12))
```

```
Our OLS beta coefficients:      [ 0.841886  0.274647 -0.023264  0.053426
-0.003465 -0.008778]
Sklearn LinearRegression beta: [ 0.841886  0.274647 -0.023264  0.053426
-0.003465 -0.008778]
Difference (our - sklearn):    [ 0. -0. -0.  0.  0. -0.]
```