Session2 - solutions

January 23, 2019

In [1]: import numpy as np
   import pandas as pd
   import sklearn.linear_model as skl_lm
   import matplotlib.pyplot as plt

1  2.1 Problem 1.1 using matrix multiplications

Implement the linear regression problems from Exercises 1.1(a), (b), (c), (d) and (e) in Python using matrix multiplications. A matrix

\[
X = \begin{bmatrix}
1 & 2 \\
1 & 3
\end{bmatrix}
\]

can be constructed with numpy as \( X = \text{np.array([[1, 2], [1, 3]]) } \) (Make sure that numpy has been imported. Here it is imported as np). The commands for matrix multiplication and transpose in numpy are \( @ \) or \( \text{np.matmul} \) and \( .T \) or \( \text{np.transpose()} \) respectively. A system of linear equations \( Ax = b \) can be solved using \( \text{np.linalg.solve(A,b)} \). A \( k \times k \) unit matrix can be constructed with \( \text{np.eye(k)} \).

1.1 (a)

In [2]: # Construct the data matrix
   X = np.array([[1, 2], [1, 3]])
   y = np.array([-1, 1])

   # Solve the normal equations
   beta = np.linalg.solve(X.T@X, X.T@y)

   # Print the solution
   print(beta)

   # Compute prediction for \( x = 4 \)
   yhat = beta@np.array([1,4])

   # Print the prediction
   print(yhat)

[-5.  2.]
2.9999999999999964
1.2  (b)

In [3]: # Construct the data matrices
   X = np.array([[1, 2], [1, 3], [1, 4]])
   y = np.array([-1, 1, 2])

   # Compute the solution using the normal equation
   beta = np.linalg.solve(X.T@X, X.T@y)

   # Print the solution
   print(beta)

   # Compute prediction for x = 5
   yhat = beta@np.array([1, 5])

   # Print the prediction
   print(yhat)

([-3.83333333  1.5])
3.666666666666668

1.3  (c)

In [4]: # Construct the data matrices
   # Reshape the array to 2-dim so we can use np.linalg.solve()
   X = np.array([2, 3, 4]).reshape(-1, 1)
   y = np.array([-1, 1, 2]).reshape(-1, 1)

   # Compute the solution
   beta = np.linalg.solve(X.T@X, X.T@y)

   # Print the solution
   print(beta)

   # Compute prediction for x = 5
   yhat = beta@np.array([5])

   # Print the prediction
   print(yhat)

[[0.31034483]]
[1.55172414]

1.4  (d)

In [5]: # Use the solution to the Ridge Regression problem
   # Construct the data matrices
\[ X = \text{np.array([[1, 2], [1, 3], [1, 4]])} \]
\[ y = \text{np.array([-1, 1, 2])} \]
\[ \text{lambda} = 1 \]

\# Compute the solution
\[ \text{beta} = \text{np.linalg.solve}(X.T@X + \text{np.eye}(2), X.T@y) \]

\# Print the solution
\[ \text{print(beta)} \]

\# Compute prediction for \( x = 5 \)
\[ \text{yhat} = \text{beta@np.array([1, 5])} \]

\# Print the prediction
\[ \text{print(yhat)} \]

\[-0.53846154 0.46153846\]
1.7692307692307692

\subsection{1.5 (e)}

\textbf{In [6]:} \# e)

\# Construct the data matrices
\[ X = \text{np.array([[1, 2], [1, 3], [1, 4]])} \]
\[ Y = \text{np.array([[1, 0], [1, 2], [2, -1]])} \]

\# Compute the solution
\[ \text{beta} = \text{np.linalg.solve}(X.T@X, X.T@Y) \]

\# Print the solution
\[ \text{print(beta)} \]

\[ \begin{bmatrix} -3.83333333 & 1.83333333 \\ 1.5 & -0.5 \end{bmatrix} \]

\section{2 2.2 Problem 1.1 using the linear_model.LinearRegression() command}

Implement the linear regression problem from Exercises 1.1(b) and (c) using the command
\text{LinearRegression()} from \text{sklearn.linear_model}.

\subsection{2.1 (b)}

\textbf{In [7]:} \# 2b)

\[ X = \text{np.array([2, 3, 4])}.\text{reshape(-1, 1)} \]
\[ y = \text{np.array([-1, 1, 2])}.\text{reshape(-1, 1)} \]
# Learn the model using the skl_lml( ) command
model = skl_lml.LinearRegression()
model.fit(X, y)

# Print the solution
print('The coefficient beta_1 for X is : ', model.coef_)
print('The offset beta_0 is: ', model.intercept_)

# Plot the data and the model
plt.plot(X, y, 'o'
 prediction = model.predict(X)
plt.plot(X, prediction)

The coeficient beta_1 for X is: [[1.5]]
The offset beta_0 is: [-3.83333333]

Out[7]: [matplotlib.lines.Line2D at 0x22c1b988da0]

2.2  (c)

In [8]: # 2c)

# Learn the model using the skl_lml( ) command without intercept
model = skl_lml.LinearRegression(fit_intercept=False)
model.fit(X, y)
# Print the solution
print('The coefficient for X is : ', model.coef_)
print('The offset is: ', model.intercept_)

# Plot the data and the model
plt.plot(X, y, 'o')
prediction = model.predict(X)
plt.plot(X, prediction)

The coefficient for X is :  
[0.31034483]
The offset is:  0.0

Out[8]:  [<matplotlib.lines.Line2D at 0x22c1b9c69e8>]

3  2.3 The Auto data set

3.1  (a)

Load the dataset 'Data/Auto.csv'. Familiarize yourself with the dataset using Auto.info(). The dataset:

Description: Gas mileage, horsepower, and other information for 392 vehicles.
Format: A data frame with 392 observations on the following 9 variables.

- mpg: miles per gallon
• cylinders: Number of cylinders between 4 and 8
• displacement: Engine displacement (cu. inches)
• horsepower: Engine horsepower
• weight: Vehicle weight (lbs.)
• acceleration: Time to accelerate from 0 to 60 mph (sec.)
• year: Model year (modulo 100)
• name: Vehicle name

The original data contained 408 observations but 16 observations with missing values were removed.

In [9]: # Set seed to get reproducible results
    np.random.seed(1)

    # a) load library and familiarize with the data
    # The null values are '?' in the dataset. `na_values='?'` recognize the null values.
    # There are null values that will mess up the computation. Easier to drop them by 'dropna'.
    Auto = pd.read_csv('Data/Auto.csv', na_values='?').dropna()
    print(Auto.shape)
    Auto.info()

    (392, 9)
    <class 'pandas.core.frame.DataFrame'>
    Int64Index: 392 entries, 0 to 396
    Data columns (total 9 columns):
    mpg 392 non-null float64
    cylinders 392 non-null int64
    displacement 392 non-null float64
    horsepower 392 non-null float64
    weight 392 non-null int64
    acceleration 392 non-null float64
    year 392 non-null int64
    origin 392 non-null int64
    name 392 non-null object
    dtypes: float64(4), int64(4), object(1)
    memory usage: 30.6+ KB

3.2 (b)

Divide the data set randomly into two approximately equally sized subsets, train and test by generating the random indices using np.random.choice().

In [10]: print(Auto.shape) #(No. of rows, No. of columns)
    trainI = np.random.choice(Auto.shape[0], size=200, replace=False)
    trainIndex = Auto.index.isin(trainI)
    train = Auto.iloc[trainIndex]
    test = Auto.iloc[-trainIndex]

    (392, 9)
3.3  (c)

Perform linear regression with mpg as the output and all other variables except name as input. How well (in terms of root-mean-square-error) does the model perform on test data and training data, respectively?

```python
# Ignore RuntimeWarning: internal gelsd driver lwork query error. Harmless

# Linear regression
model = skl_lm.LinearRegression(fit_intercept = True)  # Add an offset
X_train = train[['cylinders', 'displacement', 'horsepower', 'weight',
                 'acceleration', 'year', 'origin']]
Y_train = train['mpg']
model.fit(X_train, Y_train)
print(model)

# Evaluate on training data
train_predict = model.predict(X_train)
train_RMSE = np.sqrt(np.mean((train_predict - train.mpg)**2))
print('Train RMSE: %2.6f' % train_RMSE)

## Evaluate on test data
X_test = test[['cylinders', 'displacement', 'horsepower', 'weight',
               'acceleration', 'year', 'origin']]
test_predict = model.predict(X_test)
test_RMSE = np.sqrt(np.mean((test_predict - test.mpg)**2))
print('Test RMSE: %2.6f' % test_RMSE)
```

LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
Train RMSE: 3.282090
Test RMSE: 3.336977

3.4  (d)

Now, consider the input variable origin. What do the different numbers represent? By running `Auto.origin.sample(30)` we see the 30 samples of the variable and that the input variables is quantitative. Does it really makes sense to treat it as a quantitative input? Use `np.get_dummies()` to split it into dummy variables and do the linear regression again.

```python
# Examples of the origin variable
print('Auto origin:')
print(Auto.origin.sample(30).tolist())

X_train = pd.get_dummies(train, columns=['origin'])
print('X after transformation (origin has been split in three dummy variables):')
print(X_train.head())

# Pick out the input variables
X_train = X_train[['cylinders', 'displacement', 'horsepower', 'weight',
```
'acceleration', 'year', 'origin_1', 'origin_2', 'origin_3']

X_test = pd.get_dummies(test, columns=['origin'])
X_test = X_test[['cylinders', 'displacement', 'horsepower', 'weight',
                 'acceleration', 'year', 'origin_1', 'origin_2', 'origin_3']]

# look at how sci-kit learn transforms the qualitative input
print(X_train.sample(5))

# Repeat c) create and evaluate the model now using encoded categorical data
model1 = skl_lm.LinearRegression()
model1.fit(X_train, Y_train)
print(model1)

# Evaluate on training data
train_predict = model1.predict(X_train)
train_RMSE = np.sqrt(np.mean((train_predict - train.mpg)**2))
print('Train RMSE: %2.6f' % train_RMSE)

## Evaluate on test data
test_predict = model1.predict(X_test)
test_RMSE = np.sqrt(np.mean((test_predict - test.mpg)**2))
print('Test RMSE: %2.6f' % test_RMSE)

Auto origin:
[1, 1, 1, 1, 1, 2, 1, 3, 1, 3, 2, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3]
X after transformation (origin has been split in three dummy variables):

mpg cylinders displacement horsepower weight acceleration year
0 18.0 8 307.0 130.0 3504 12.0 70
4 17.0 8 302.0 140.0 3449 10.5 70
5 15.0 8 429.0 198.0 4341 10.0 70
6 14.0 8 454.0 220.0 4354 9.0 70
8 14.0 8 455.0 225.0 4425 10.0 70

name origin_1 origin_2 origin_3
0 chevrolet chevelle malibu 1 0 0
4 ford torino 1 0 0
5 ford galaxie 500 1 0 0
6 chevrolet impala 1 0 0
8 pontiac catalina 1 0 0

cylinders displacement horsepower weight acceleration year
98 6 250.0 100.0 3278 18.0 73
285 8 305.0 130.0 3840 15.4 79
236 4 140.0 89.0 2755 15.8 77
213 8 350.0 145.0 4055 12.0 76
373 4 140.0 92.0 2865 16.4 82

origin_1 origin_2 origin_3
3.5 (e)

Try obtain a better RMSE on test data by removing some inputs (explore what happens if you remove, e.g., year, weight and acceleration)

In [13]: # First write a function that takes the prediction model, training and test data # and computes RMSE to simplify the process

    def computeRMSE(model, X, Y):
        Y_predict = model.predict(X)
        RMSE = np.sqrt(np.mean((Y_predict - Y)**2))
        return RMSE

    # The following function streamlines the procedure of testing with dropping # different variables. It is optional. But if you want to skip the function # keep in mind that when you declare e.g. X=X.drop(columns=column_name), # you are manipulating the original X
    def RMSE_with_drop_col(model, X, Y, X_test, Y_test, drop_col):
        # drop_col takes a list of string or strings
        print("Results without the variable \"%s\":" % drop_col)
        X = X.drop(columns=drop_col)
        model.fit(X, Y)
        train_RMSE = computeRMSE(model, X, Y)
        print('Train RMSE %.6f' % train_RMSE)

        X_test = X_test.drop(columns=drop_col)
        test_RMSE = computeRMSE(model, X_test, Y_test)
        print('Test RMSE %.6f' % test_RMSE)
        print()

    # Test output (has not been declared)
    Y_test = test.mpg

    # Remove weight
    model2 = skl_lm.LinearRegression()
    RMSE_with_drop_col(model2, X_train, Y_train, X_test, Y_test, \
                      ['weight', 'acceleration'])
# Remove year
model3 = skl_lm.LinearRegression()
RMSE_with_drop_col(model3, X_train, Y_train, X_test, Y_test, ['year'])

# Remove acceleration
model4 = skl_lm.LinearRegression()
RMSE_with_drop_col(model4, X_train, Y_train, X_test, Y_test, ['acceleration'])

Results without the variable ['weight', 'acceleration']:
Train RMSE 3.749640
Test RMSE 3.835552

Results without the variable ['year']:
Train RMSE 4.171414
Test RMSE 4.097517

Results without the variable ['acceleration']:
Train RMSE 3.264964
Test RMSE 3.312246

3.6 (f)
Try to obtain a better RMSE on test data by adding some transformations of inputs, such as $\log(x)$, $\sqrt{x}$, $x_1x_2$ etc.

In [14]: # A small function to simplify the process
def RMSE_with_cols(model, X, Y, cols):
    print("RMSE with the variables '%s':" % cols)
    X = X[cols]
    model.fit(X, Y)
    RMSE = computeRMSE(model, X, Y)
    print('RMSE %2.6f' % RMSE)

# horsepower*acceleration
print('comp = horsepower * acceleration')
model = skl_lm.LinearRegression()
X_train_copy = X_train.copy() # A hard copy to avoid manipulating the original X
X_train_copy['comp'] = X_train_copy.horsepower * X_train_copy.acceleration
cols = ['cylinders', 'displacement','comp', 'origin_1', 'origin_2', 'origin_3']
RMSE_with_cols(model, X_train_copy, Y_train, cols)

print()

# sqrt(horsepower) and weight^2
print('sqrt(horsepower) and weight^2')
model = skl_lm.LinearRegression()
\[
\begin{align*}
X_{\text{train}}_{\text{copy}} &= X_{\text{train}}.\text{copy()} \\
X_{\text{train}}_{\text{copy}}[\text{'sqrt\_horsepower']} &= \text{np.sqrt}(X_{\text{train}}_{\text{copy}}.\text{horsepower}) \\
X_{\text{train}}_{\text{copy}}[\text{'weight\_sqr']} &= X_{\text{train}}_{\text{copy}}.\text{weight}^2 \\
cols &= [\text{'cylinders'}, \text{'displacement'}, \text{'sqrt\_horsepower'}, \text{'weight\_sqr'}, \text{'origin\_1'}, \text{'origin\_2'}, \text{'origin\_3']} \\
\text{RMSE} &\text{with}_{\text{cols}}(\text{model}, X_{\text{train}}_{\text{copy}}, Y_{\text{train}}, \text{cols})
\end{align*}
\]

comp = horsepower * acceleration
RMSE with the variables ['cylinders', 'displacement', 'comp', 'origin\_1', 'origin\_2', 'origin\_3']
RMSE 4.187345

\[
\begin{align*}
\text{sqrt\_horsepower}\text{ and weight}^2 \\
\text{RMSE} &\text{with}_{\text{cols}}(\text{model}, X_{\text{train}}_{\text{copy}}, Y_{\text{train}}, \text{cols})
\end{align*}
\]

RMSE 4.182440

### 4 2.4 Nonlinear transformations of input variables

Start by running the following code to generate your training data

\[
\begin{align*}
\text{np}\.\text{random}\.\text{seed}(1) \\
x_{\text{train}} &= \text{np}\.\text{random}\.\text{uniform}(0, 10, 100) \\
y_{\text{train}} &= .4 - .6 * x_{\text{train}} + 3. * \text{np}\.\text{sin}(x_{\text{train}} - 1.2) + \text{np}\.\text{random}\.\text{normal}(0, 0.1, 100)
\end{align*}
\]

In [15]: np.random.seed(1)
    x_train = np.random.uniform(0, 10, 100)
    y_train = .4 - .6 * x_train + 3. * np.sin(x_train - 1.2) + np.random.normal(0, 0.1, 100)

4.1 (a)

Plot the training output y_train versus the training input x_train.

In [16]: # a) Plot
    plt.plot(x_train, y_train, 'o')
    plt.xlabel('Input')
    plt.ylabel('Output')

Out[16]: Text(0,0.5,'Output')
4.2 (b)

Learn a model on the form
\[ y = a + bx + csin(x + \phi) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, 1^2) \]  \hspace{1cm} (2.1)

where all parameters \( a, b, c \) and \( \phi \) are to be learned from the training data \( x_{\text{train}} \) and \( y_{\text{train}} \). Refrain from using the \texttt{linear_model()} command, but implement the normal equations yourself as in problem 2.1. Hint: Even though (2.1) is not a linear regression model, you can use the fact that \( csin(x + \phi) = ccos(\phi)sin(x) + csin(\phi)cos(x) \) to transform it into one.

In [17]: # b) Do linear regression
   
   X_train = np.column_stack([np.repeat(1, 100), x_train, np.cos(x_train), np.sin(x_train)])
   
   y_train = np.array(y_train).reshape(-1, 1)
   
   beta = np.linalg.solve(X_train.T @ X_train, X_train.T @ y_train)
   
   beta
   
   Out[17]: array([[ 0.42117995],
                   [-0.60266039],
                   [-2.78869453],
                   [ 1.08808499]])

4.3 (c)

Construct 100 test inputs \( x_{\text{test}} \) in the span from 0 to 10 by using the \texttt{np.linspace()} function. Predict the outputs corresponding to these inputs and plot them together with the training data.
In [18]: # c) Do prediction
    x_test = np.random.uniform(0, 10, 100)
    X_test = np.column_stack([np.repeat(1, 100), x_test, np.cos(x_test), np.sin(x_test)])
    yhat = X_test@beta
    plt.plot(x_test, yhat, 'o')
    plt.show()

4.4 (d)

Do a least squares fit by instead using the `linear_model()` function in Python. Check that you get the same estimates as in (b).

In [19]: # d) Linear regression
    model = skl_lm.LinearRegression()
    model.fit(X_train, y_train)
    prediction = model.predict(X_test)

    plt.plot(x_train, y_train, 'o', label='data')
    plt.plot(x_test, yhat, 'o', label='Lin. Reg. (normal equations)')
    plt.plot(x_test, prediction, 'o', label='Lin. Reg (built-in function)')

    plt.legend()
    plt.show()
5 2.5 Regularization

In this exercise we will apply Ridge regression and Lasso for fitting a polynomial to a scalar data set. We will have a setting where we first generate synthetic training data from

\[ y = x^3 + 2x^2 + 6 + \epsilon, \quad (2.2) \]

and later try to learn model for the data.

5.1 (a)

Write a function that implements the polynomial (2.2), i.e., takes \( x \) as argument and returns \( x^3 + 2x^2 + 6 \).

In [20]: # a)
def f(x):
    return x**3 + 2*x**2 + 6

5.2 (b)

Use `np.random.seed()` to set the random seed. Use the function `np.linspace()` to construct a vector \( x \) with \( n = 12 \) elements equally spaced from \(-2.3\) to \(1\). Then use your function from (a) to construct a vector \( y = [y_1, ..., y_n]^T \) with 12 elements, where \( y = x^3 + 2x^2 + 6 + \epsilon \), with \( \epsilon \sim \mathcal{N}(0, \infty) \). This is our training data.
5.3 (c)

Plot the training data $T = \{x_i, y_i\}_{i=1}^{12}$ together with the "true" function.

5.4 (d)

Fit a straight line to the data with $y$ as output and $x$ as input and plot the predicted output $\hat{y}$, for densely spaced $x*$ values between $-2.3$ and $1$. Plot these predictions in the same plot window.
prediction = model.predict(x_test.reshape(-1,1))

# Plots
plt.plot(x_train, y_train, 'o', label='data')
plt.plot(x_test, y_test, label='true function')
plt.plot(x_test, prediction, label='linear regression')
plt.legend()
plt.show()

5.5 (e)
Fit a 11th degree polynomial to the data with linear regression. Plot the corresponding predictions.

In [24]: # e)
   # Add X^n, n=1, 2...11, to x_train and x_test
   x_train_ext = x_train.reshape(-1,1)
   x_test_ext = x_test.reshape(-1,1)

   for i in range(10):
       x_train_ext = np.column_stack([x_train_ext, x_train.reshape(-1,1)**(i+2)])
       x_test_ext = np.column_stack([x_test_ext, x_test.reshape(-1,1)**(i+2)])

   # verify
   print(x_train_ext[0,:])
   print(x_test_ext[0,:])
5.6 (f)

Use the function `sklearn.linear_model.Ridge` and `sklearn.linear_model.Lasso` to fit a 11th degree polynomial. Also inspect the estimated coefficients. Try different values of penalty term $\alpha$. What do you observe?

In [26]: # f)
    # With Ridge
model = skl_lm.Ridge(alpha=1)
model.fit(x_train_ext, y_train)
print('Model coefficients: ')
print(model.coef_)

prediction = model.predict(x_test_ext)

# Plots
plt.plot(x_train, y_train, 'o', label='data')
plt.plot(x_test, y_test, label='true function')
plt.plot(x_test, prediction, label='linear regression with Ridge')

plt.legend()
plt.show()

Model coefficients:
[ 0.24047501  0.75707966 -0.0506247  0.79748062 -0.04197945  0.50957368
  0.38463181  0.01286643  0.56377651  0.47223915  0.0970541 ]

In [27]: # With Lasso
model = skl_lm.Lasso(alpha=0.1)
model.fit(x_train_ext, y_train)
print('Model coefficients: ')
print(model.coef_)
prediction = model.predict(x_test_ext)

# Plots
plt.plot(x_train, y_train, 'o', label='data')
plt.plot(x_test, y_test, label='true function')
plt.plot(x_test, prediction, label='linear regression with Lasso')
plt.legend()
plt.show()

Model coefficients:
[ 1.75687912e-01  2.20069018e+00  0.00000000e+00  1.99502387e-01
  4.20853300e-01  0.00000000e+00  1.85824779e-02  3.57481443e-03
 -1.73710840e-03  1.49116638e-03 -4.59040376e-04]

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\coordinate_descent.py:491: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations. Fitting data with very small alpha may cause precision problems.
  ConvergenceWarning)