Lecture 2 - Jan 10

Implementing Linear Models and OLS
Data Preprocessing

Recommended Reading

Week 1 Notes in GitHub

Data Mining and Machine Learning

23.1: Linear Regression Model
23.3: Multiple Regression

Elements of Statistical Learning

2.3.1: Linear Models and Least Squares
2.6.2: Supervised Learning

Upcoming Deadlines

Python Exam (Jan 20)
Homework 1 (Jan 27)

Matrix Inverses

Assume A is a square matrix, i.e. $A \in \mathbb{R}^{n \times n}$

A matrix A^{-1} such that $AA^{-1} = A^{-1}A = I_n$ is the **matrix inverse** of A

If A has an inverse, it is **invertible**, otherwise it is **singular**

↳ A being invertible is central to elementary linear algebra (see the invertible matrix theorem)

properties: $(kA)^{-1} = \frac{1}{k} A^{-1}$

$(A^T)^{-1} = (A^{-1})^T$

$(AB)^{-1} = B^{-1}A^{-1}$

Computational Note

Matrix inversion is $O(n^{2.373}) \rightarrow$ tools will do it efficiently

Linear Systems

Let $A \in \mathbb{R}^{n \times d}$, $x, b \in \mathbb{R}^d$, $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$, $b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_d \end{bmatrix}$

$$\begin{matrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1d}x_d = b_1 & \rightarrow \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2d}x_d = b_2 & \rightarrow \\ \vdots \qquad \vdots \qquad \vdots \qquad \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nd}x_d = b_d & \rightarrow \end{matrix} \begin{bmatrix} a_1 \cdot x \\ a_2 \cdot x \\ \vdots \\ a_n \cdot x \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \\ b_d \end{bmatrix}$$

$$\longrightarrow Ax = b$$

system of linear equations can be written compactly as a matrix equation

If A is invertible, we can solve for the vector x:

$$(A^{-1}A)x = A^{-1}b$$

$$Ix = A^{-1}b$$

$$x = A^{-1}b$$

If A is singular, the system does not have a unique solution

OLS Solution

$$\nabla L(\theta) = 2X^TX\theta - 2X^Ty = 0$$

$$\cancel{(X^TX)^{-1}(X^TX)}\theta = (X^TX)^{-1}X^Ty$$

exact solution to the OLS problem $\min_\theta L(\theta)$ where the model $\hat{f}$ is a linear function

$$\theta = \underbrace{(X^TX)}^{-1}X^Ty$$

this inverse exists for $n \geq d+1$ if the columns of X are linearly independent

Recall

[NOTE: replace beta with theta throughout to be consistent with the other notes…]

A regression model

$$Y = f(X_1, \ldots, X_d) + \varepsilon$$

For linear regression, we assume f is linear in betas. In the simplest case,

$$f(x_i) = \beta_0 + \sum_{k=1}^{d} \beta_k x_{ik} = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_d x_{id}$$

Note

It is a common misconception that "linear regression" must fit linear functions to data, but the "linear" part of linear regression refers to the fact that f is linear with respect to $\beta_0, \ldots, \beta_d$, not with respect to x_i, so it is certainly possible to apply some preprocessing to the datapoints, which in effect, fits a nonlinear surface.

For example, if each $x_i \in \mathbb{R}^1$, we can fit a parabola by manipulating each x_i into $x_i^* = (x_i, x_i^2)$ so that our predicted function would be

$$f(x_i) = \beta_0 + \beta_1 x_i + \beta_2 x_i^2$$

While we will discuss only points in the form $x_i = (x_{i1}, \ldots, x_{id})$ for now, keep in mind that we can always create new variables from the data, preprocess the data into different forms, and so on, to consider some (kernel) function of the data $g(x_i)$ as the inputs, as long as the function is differentiable (or at least piecewise differentiable). The main point here is that linear regression can learn to represent functions far beyond simply lines and planes.

Recall, we had a loss function

$$L(\beta) = \sum_{i=1}^{n} (f(x_i) - y_i)^2 = \sum_{i=1}^{n} \left(x_i^T \beta - y_i\right)^2,$$

where

$$X = \begin{pmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1d} \\ 1 & x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix} \qquad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \qquad \beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_d \end{pmatrix}$$

Solving the optimization problem

$$\min_{\beta} \; L(\beta)$$

We derived the parameters (betas) to be

$$\beta = (X^T X)^{-1} X^T y.$$

This is the solution to the **ordinary least squares (OLS) regression** problem.

In some sources, it is called the **normal equation** approach to solving the OLS problem.

Ordinary Least Squares Code

Before we write some code for ordinary least squares, let's import some packages.

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split

# increase the width of boxes in the notebook file (this is only cosmetic)
np.set_printoptions(linewidth=250)
```

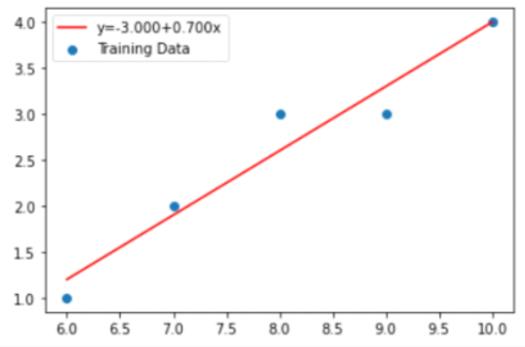
Let's create a class for using ordinary least squares in this way to fit the model and to predict outputs for unknown inputs. We will use the scikit-learn pattern.

```python
class OrdinaryLeastSquaresExact:

    # fit the model to the data
    def fit(self, X, y):
        # save the training data
        self.data = np.hstack((np.ones([X.shape[0],1]), X))

        # save the training labels
        self.outputs = y

        # find the beta values that minimize the sum of squared errors
        X = self.data
        self.beta = np.linalg.inv(X.T @ X) @ X.T @ y

    # predict the output from input (testing) data
    def predict(self, X):
        # initialize an empty matrix to store the predicted outputs
        yPredicted = np.empty([X.shape[0],1])

        # append a column of ones at the beginning of X
        X = np.hstack((np.ones([X.shape[0],1]), X))

        # apply the function f with the values of beta from the fit function to each testing datapoint
        for row in range(X.shape[0]):
            yPredicted[row] = self.beta @ X[row,]

        return yPredicted
```

1D Example

```python
X = np.array([[6], [7], [8], [9], [10]])
y = np.array([1, 2, 3, 3, 4])

model = OrdinaryLeastSquaresExact()
model.fit(X,y)
predictions = model.predict(X)

# print the predictions
print('The predicted y values are', predictions.T[0])

# print the real y values
print('The real y values are', y)

# print the beta values
parameters = model.beta
print('The beta values are', parameters)

# plot the training points
plt.scatter(X, y, label = 'Training Data')

# plot the fitted model with the training data
xModel = np.linspace(6,10,100)
yModel = parameters[0] + parameters[1]*xModel
lineFormula = 'y={:.3f}+{:.3f}x'.format(parameters[0], parameters[1])
plt.plot(xModel, yModel, 'r', label = lineFormula)

# add a legend
plt.legend()

# return quality metrics
print('The r^2 score is', r2_score(y, predictions))
print('The mean squared error is', mean_squared_error(y, predictions))
print('The mean absolute error is', mean_absolute_error(y, predictions))
```

```
The predicted y values are [1.2 1.9 2.6 3.3 4. ]
The real y values are [1 2 3 3 4]
The beta values are [-3.   0.7]
The r^2 score is 0.9423076923076923
The mean squared error is 0.06
The mean absolute error is 0.20000000000000054
```



Training and Testing a Regression Model

If we have a dataset of labeled data, a very common approach is to randomly split the dataset into two parts: the training set and the testing set. We remove the labels from the test set, "train" the model with the training set, use the resulting model to attempt to make predictions for the test set, and then measure performance.

There are no strict rules here, but it is common to use 60% train 40% test, although folks like Andrew Ng have argued that a much smaller test sets are reasonable if the dataset is very large--e.g. datasets with millions of datapoints are not uncommon in some fields. This way, we can measure the success of our regression model on data it has never seen (the testing set). Once we become confident our model works well in this way, we can be more confident that it will **generalize** well in the real world.

My preferred approach is to use the train_test_split function from the scikit-learn.model_selection library to randomly assign a specified percentage (usually 50-75%) of the dataset to the training set and the rest in the test set.

Hyperparameters and Dev Sets

In some models, there are **hyperparameters** to tune. These are settings the user specifies before running the algorithms which may have an impact on performance. In this case, the test set is frequently split in half into "dev" and "test" sets. The dev set is used for tuning the hyperparameters before testing the model on unknown test sets. More on that later.

Performance Metrics for Regression

Let's consider a few performance metrics in common usage for regression.

With linear regression, we generally have the unfortunate situation that the model we construct is not perfect even on the training set. Therefore, we first need to consider the performance on the training data to which it was fit just to see how well the model fits to the training data. The formulas for the common metrics are not particularly interesting, so we just state what they represent and what value they should ideally have:

- **Coefficient of determination** R^2 - the fraction of the variation in the data explained by the model. It is, in a sense, a measure of the strength of the linear relationship between the variables. Ideally, it will be near 1.
- **Sum of squared error (SSE)** - the loss function we have used. Ideally, it will be as small as possible.
- **Mean squared error (MSE)** is simply the SSE divided by the number of examples we test. It is frequently better because it makes little sense to measure success in a way that is so depedent on the dataset size. Ideally, it should of course be small.
- **Mean absolute error (MAE)** - the mean of the absolute errors between the points and the fitted function. Ideally, it will be as small as possible.

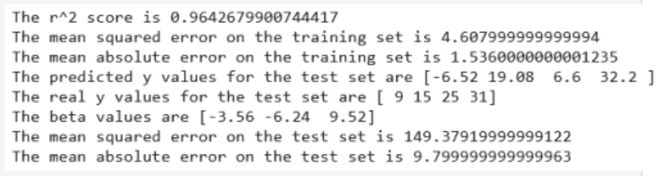
If these values are far from their ideal values for the training set, the model does not even fit the training data well, so it probably will not fit the testing data well. The ordinary least squares solution finds the optimal parameters for a linear fit, so poor performance on the training set means the data do not have a strong linear relationship.

Some preprocessing of the data might make it work better. For example, you can apply a logarithm to a variable if there's a linear relationship with that variable on a log scale. See the (free) classic book *Elements of Statistical Learning* by Hastie, et. al., section 2.6.3 for an introduction on linear basis expansions.

Second, we need to consider the performance on the testing data. We generally should consider the MSE or MAE.

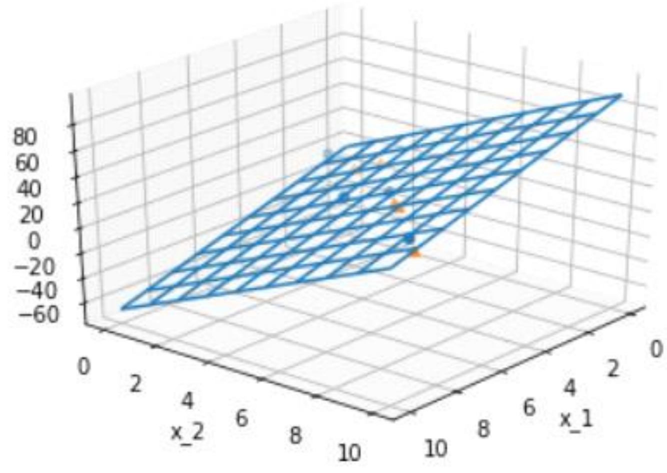
2D Example

```python
trainX = np.array([[2, 2], [2, 3], [5, 6], [6, 7], [9, 10]])
trainY = np.array([3, 13, 19, 29, 35])

testX = np.array([[2, 1], [4, 5], [6, 5], [8, 9]])
testY = np.array([9, 15, 25, 31])

# instantiate an OLS model
model = OrdinaryLeastSquaresExact()

# fit the model to the training data (find the beta parameters)
model.fit(trainX, trainY)

# return the predicted outputs for the datapoints in the training set
trainPredictions = model.predict(trainX)

# print the coefficient of determination r^2
print('The r^2 score is', r2_score(trainY, trainPredictions))

# print quality metrics
print('The mean squared error on the training set is', mean_squared_error(trainY, trainPredictions))
print('The mean absolute error on the training set is', mean_absolute_error(trainY, trainPredictions))

# return the predicted outputs for the datapoints in the test set
predictions = model.predict(testX)

# print the predictions
print('The predicted y values for the test set are', predictions.T[0])

# print the real y values
print('The real y values for the test set are', testY)

# print the beta values
print('The beta values are', model.beta)

# print quality metrics
print('The mean squared error on the test set is', mean_squared_error(testY, predictions))
print('The mean absolute error on the test set is', mean_absolute_error(testY, predictions))
```



```
The r^2 score is 0.9642679900744417
The mean squared error on the training set is 4.607999999999994
The mean absolute error on the training set is 1.5360000000001235
The predicted y values for the test set are [-6.52 19.08  6.6  32.2 ]
The real y values for the test set are [ 9 15 25 31]
The beta values are [-3.56 -6.24  9.52]
The mean squared error on the test set is 149.37919999999122
The mean absolute error on the test set is 9.799999999999963
```

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

n = 100

# scatter plot of the test data
x1 = testX[:,0]
x2 = testX[:,1]
y = testY
ax.scatter(x1, x2, y, marker = 'o')

# scatter plot of the training data
x1 = trainX[:,0]
x2 = trainX[:,1]
y = trainY
ax.scatter(x1, x2, y, marker = '^')

# plot the plane we fit to the data
beta = model.beta

# surface plot
x1 = np.linspace(0,10,10)
x2 = np.linspace(0,10,10)

X1, X2 = np.meshgrid(x1,x2)
#Y = beta[0]*X1 + beta[1]*X2
Y = beta[0] + beta[1]*X1 + beta[2]*X2

surf = ax.plot_wireframe(X1, X2, Y)
ax.view_init(10, 50)

# add axis labels
ax.set_xlabel('x_1')
ax.set_ylabel('x_2')
ax.set_zlabel('y')
```



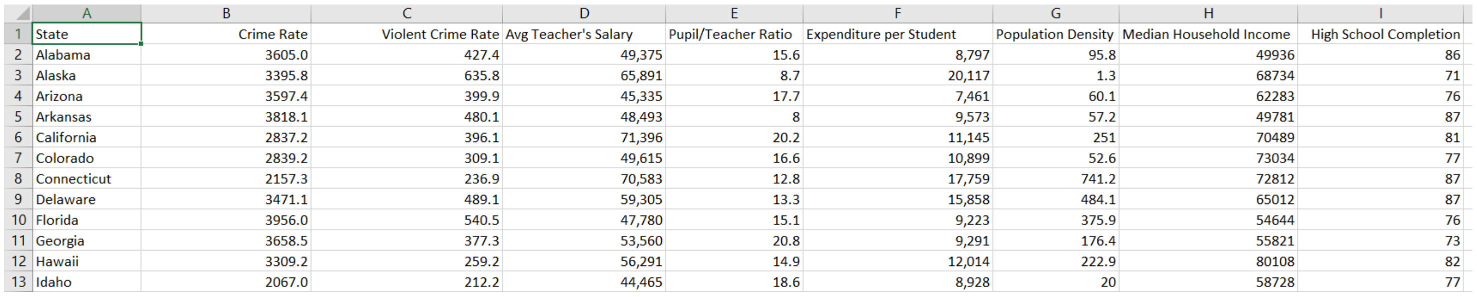
Small Dataset Example

Example: High School Graduation Rates in US States

Let's try to use ordinary least squares on a real dataset. The CSV file in '/data/US_State_data.csv' contains data from each U.S. state.

We would like to predict the output variable included, the high school graduation rate, from some input variables: including the crime rate (per 100,000 persons), the violent crime rate (per 100,000 persons), average teacher salary, student-to-teacher ratio, education expenditure per student, population density, and median household income.

This means we have 50 examples (one for each state), 7 input (predictor) variables, and one output (response) variable. In order to use the formula we derived above to attack the problem with ordinary least squares, we need to find the matrices $\boldsymbol{X}$ and $\boldsymbol{y}$.



	A	B	C	D	E	F	G	H	I
1	State	Crime Rate	Violent Crime Rate	Avg Teacher's Salary	Pupil/Teacher Ratio	Expenditure per Student	Population Density	Median Household Income	High School Completion
2	Alabama	3605.0	427.4	49,375	15.6	8,797	95.8	49936	86
3	Alaska	3395.8	635.8	65,891	8.7	20,117	1.3	68734	71
4	Arizona	3597.4	399.9	45,335	17.7	7,461	60.1	62283	76
5	Arkansas	3818.1	480.1	48,493	8	9,573	57.2	49781	87
6	California	2837.2	396.1	71,396	20.2	11,145	251	70489	81
7	Colorado	2839.2	309.1	49,615	16.6	10,899	52.6	73034	77
8	Connecticut	2157.3	236.9	70,583	12.8	17,759	741.2	72812	87
9	Delaware	3471.1	489.1	59,305	13.3	15,858	484.1	65012	87
10	Florida	3956.0	540.5	47,780	15.1	9,223	375.9	54644	76
11	Georgia	3658.5	377.3	53,560	20.8	9,291	176.4	55821	73
12	Hawaii	3309.2	259.2	56,291	14.9	12,014	222.9	80108	82
13	Idaho	2067.0	212.2	44,465	18.6	8,928	20	58728	77

```python
# import the data from the csv file to an numpy array
data = pd.read_csv('data/US_State_Data.csv', sep=',').to_numpy()

# select the data and the labels
X = np.array(data[:,1:7], dtype=float)
y = np.array(data[:,8], dtype=float)

# split the data into training and test sets
(trainX, testX, trainY, testY) = train_test_split(X, y, test_size = 0.25, random_state = 1)

# run the model (same code as above)

# instantiate an OLS model
model = OrdinaryLeastSquaresExact()

# fit the model to the training data (find the beta parameters)
model.fit(trainX, trainY)

# return the predicted outputs for the datapoints in the training set
trainPredictions = model.predict(trainX)

# print the coefficient of determination r^2
print('The r^2 score is', r2_score(trainY, trainPredictions))

# print quality metrics
print('The mean squared error on the training set is', mean_squared_error(trainY, trainPredictions))
print('The mean absolute error on the training set is', mean_absolute_error(trainY, trainPredictions))

# return the predicted outputs for the datapoints in the test set
predictions = model.predict(testX)

# print the predictions
print('The predicted y values for the test set are', np.round(predictions.T[0],0))
# print the real y values
print('The real y values for the test set are      ', testY)

# print the beta values
print('The beta values are', model.beta)

# print quality metrics
print('The mean squared error on the test set is', mean_squared_error(testY, predictions))
print('The mean absolute error on the test set is', mean_absolute_error(testY, predictions))
```

```
The r^2 score is 0.3921626378110835
The mean squared error on the training set is 19.59820392039355
The mean absolute error on the training set is 3.796456104985723
The predicted y values for the test set are [80. 80. 89. 89. 80. 81. 85. 93. 75. 84. 84. 79. 83.]
The real y values for the test set are      [70. 83. 83. 81. 76. 87. 89. 89. 78. 78. 84. 80. 79.]
The beta values are [ 1.16286460e+02 -6.09008374e-03  3.98286852e-03 -1.63697058e-04 -2.97328939e-01 -
4.74750536e-04  1.09164594e-02]
The mean squared error on the test set is 28.314919261803976
The mean absolute error on the test set is 4.643935838705472
```