Linear Models

Abhishek Swain

2023-09-16

Table of Contents

List of Figures

Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import add_dummy_feature
from sklearn.linear_model import LinearRegression
```

The equations

Standard equation

$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter.

Vectorized equation

$\hat{y} = h_\theta(x) = \theta \cdot x$

- θ here is the model's parameter containing $\theta_0 \dots \theta_n$.
- x is the features containing $x_0 \dots x_n$ where x_0 is always 1.
- $\theta \cdot x$ is the dot product between θ & x. Both are colum vectors.

$$\hat{y} = \theta \cdot x = \theta^T x = [\theta_0 \dots \theta_n] \begin{bmatrix} x_0 \\ \vdots \\ x_n \end{bmatrix} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

where $x_0 = 1$

Training the model

We need a loss function to train this model. Out loss funciton is MSE loss.

$MSE(X, h_\theta) = \frac{1}{m}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2$

Here (i) is the i^{th} training example.

The Normal Equation

Without using any optimization algorithm we also have a direct formula to get the parameters, this formula is the *Normal Equation*

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

- $\hat{\theta}$ is the estimated parameter, $\hat{\theta} \approx \theta$
- y is is the target vector containing $y^{(1)} \dots y^{(m)}$

Examples

```
# Generate dummy data

np.random.seed(42)
m = 100
X = 2 * np.random.randn(m, 1)
y = 3 * X + np.random.randn(m, 1)

# Add bias term to X
X_b = add_dummy_feature(X)
```

```
X_b.shape, y.shape
```

```
((100, 2), (100, 1))
```

```
best_theta = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y # Using the normal eqaution
```

```
best_theta
```

```
array([[0.00742783],
       [2.92837142]])
```

The equation is $y = 3x$ + Gaussian Noise, and you can see that the estimated θ_1 is $2.9 \approx 3$ and $\theta_0 = 0$

```
# Make some predictions

X_new = np.array([[0], [2]])
X_new = add_dummy_feature(X_new)
```

```
y_pred = X_new @ best_theta
```

```
y_pred
```

```
array([[0.00742783],
       [5.86417067]])
```

Linear Regression with `scikit-learn`

```
lr = LinearRegression()
lr.fit(X_b, y)
```

```
LinearRegression()
```

```
lr.coef_
```

```
array([[0.        , 2.92837142]])
```

The `LinearRegression` in `sklearn` computes $X^{+}y$ where X_{+} is the pseudoinverse (Moore - Penrose pseudoinverse). We can compute it directly using `np.linalg.pinv`

```
np.linalg.pinv(X_b) @ y
```

```
array([[0.00742783],
       [2.92837142]])
```

Computational Complexity

- Computing the inverse of $X^T X$ which is (n + 1) x (n + 1) matrix upto $O(n^3)$. The scikit-learn Linear Regression is almost $O(n^2)$
- However, once computed, time to make predictions is very fast. It scales $O(n)$ with the number of predictions to be made

Gradient Descent

- This is an iterative method to find the parameters of the model
- We compute the gradient of the loss function at a particular point and move in the direction of the -ve gradient (the steepest descent)

Ex: You want to go down hill from a the top but it is very foggy, you use your feet to find the next steepest descent point

We are basically searching in the model's parameter space, so more the parameters, the harder the search becomes.

Factors affecting search: - Learning rate determines how big of a step we are taking, too big and we may overshoot - Shape of the function, if the loss function is complex (has lots of highs and toughs) there are chances our GD may get stuck in a *Local Minima* & not the *Global Minima*

Fortunately, MSE is a convex function all we do is go down the slope to the bottom of the bowl

Batch Gradient Descent

To implement GD we need to compute the partial derivative of the Loss function with each parameter of the model

$$MSE(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2$$

$$\frac{\partial MSE(\theta)}{\partial \theta_j} = \frac{2}{m}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})x_j$$

Rather than computing them one-by-one, we vectorize the whole stuff,

$$\nabla_\theta MSE(\theta) = \begin{bmatrix} \frac{\partial MSE(\theta)}{\partial \theta_0} \\ \frac{\partial MSE(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial MSE(\theta)}{\partial \theta_n} \end{bmatrix} = \frac{2}{m} X^T(X\theta - y)$$