Multiple Liear Regression

Rafiq Islam

2024-08-29

Table of contents

## Multiple Linear Regression

The multiple linear regression takes the form

$$y=β\_{0}+β\_{1}x\_{1}+β\_{2}x\_{2}+\cdots +β\_{d}x\_{d}+ξ=\vec{x}⋅\vec{β}+ξ$$

with $\{β\_{i}\}\_{i=0}^{d}\in R$ constants or parameters of the model. In vector notation, $\vec{β}\in R^{d+1}$,

$$
\vec{\beta}=\begin{pmatrix}\beta\_0\\ \beta\_1\\ \vdots \\ \beta\_d \end{pmatrix};\hspace{4mm}\vec{x}=\begin{pmatrix}1\\ x\_1\\ x\_2\\ \vdots\\ x\_d\end{pmatrix}
$$

For $n$ data points, in matrix algebra notation, we can write $y=X\vec{β}+ξ$ where $X\in M\_{n×\left(d+1\right)}$ and $y\in R^{d+1}$ with

$$
X=\begin{pmatrix}1&x\_{11}&x\_{12}&\cdots&x\_{1d}\\1&x\_{21}&x\_{22}&\cdots&x\_{2d}\\ \vdots& \vdots &\vdots&\ddots &\vdots\\1&x\_{n1}&x\_{n2}&\cdots&x\_{nd} \end{pmatrix};\hspace{4mm} y=\begin{pmatrix}y\_1\\y\_2\\ \vdots\\ y\_n\end{pmatrix};\hspace{4mm} \xi=\begin{pmatrix}\xi\_1\\ \xi\_2\\ \vdots\\ \xi\_n\end{pmatrix}
$$

We fit the $n$ data points with the objective to minimize the loss function, mean squared error

$$MSE\left(\vec{β}\right)=\frac{1}{n}\sum\_{i=1}^{n}\left(y\_{i}−f\_{\vec{β}}\left(\vec{x}\_{i}\right)\right)^{2}=\frac{1}{n}\left|\vec{y}−X\vec{β}\right|^{2}$$

## Ordinary Least Square Method

The scikit-learn library uses ***Ordinary Least Squares (OLS)*** method to find the parameters. This method is good for a simple and relatively smaller dataset. Here is a short note on this method. However, when the dimension is very high and the dataset is bigger, scikit-learn uses another method called ***Stochastic Gradient Descent*** for optimization which is discussed in the next section.

The goal of OLS is to find the parameter vector $\hat{β}$ that minimizes the sum of squared errors (SSE) between the observed target values $y$ and the predicted values $\hat{y}$:

$$SSE=\sum\_{i=1}^{n}\left(y\_{i}−\hat{y}\_{i}\right)^{2}=\sum\_{i=1}^{n}\left(y\_{i}−X\_{i}β\right)^{2}$$

This can be expressed in matrix form as:

$$SSE=\left(y−Xβ\right)^{T}\left(y−Xβ\right)$$

To minimize the SSE, let’s first expand the expression:

Since $β^{T}X^{T}y$ is a scalar (a 1x1 matrix), it is equal to its transpose. That is

and therefore,

$$SSE=y^{T}y−2β^{T}X^{T}y+β^{T}X^{T}Xβ$$

To find the minimum of the SSE, we take the derivative with respect to $β$ and set it to zero:

$$\frac{∂SSE}{∂β}=−2X^{T}y+2X^{T}Xβ=0$$

Now, solve for $β$:

$$X^{T}Xβ=X^{T}y$$

To isolate $β$, we multiply both sides by $\left(X^{T}X\right)^{−1}$ (assuming $X^{T}X$ is invertible):

$$β=\left(X^{T}X\right)^{−1}X^{T}y$$

The vector $\hat{β}=\left(X^{T}X\right)^{−1}X^{T}y$ gives the estimated coefficients that minimize the sum of squared errors between the observed target values $y$ and the predicted values $\hat{y}=X\hat{β}$. This method is exact and works well when $X^{T}X$ is invertible and the dataset size is manageable. This method is very efficient for small to medium-sized datasets but can become computationally expensive for very large datasets due to the inversion of the matrix $X^{T}X$.

## Iterative Method

### Gradient Descent

 GIF Credit: gbhat.com
 Gradient Descent is an optimization algorithm used to minimize the cost function. The cost function $f\left(β\right)$ measures how well a model with parameters $β$ fits the data. The goal is to find the values of $β$ that minimize this cost function. In terms of the iterative method, we want to find $β\_{k+1}$ and $β\_{k}$ such that $f\left(β\_{k+1}\right)<f\left(β\_{k}\right)$. For a small change in $β$, we can approximate $f\left(β\right)$ using Taylor series expansion

$$f\left(β\_{k+1}\right)=f\left(β\_{k}+Δβ\_{k}\right)≈f\left(β\_{k}\right)+∇f\left(β\_{k}\right)^{T}Δβ\_{k}+higher-order terms$$

The update rule for vanilla gradient descent is given by:

$$β\_{k+1}=β\_{k}−η∇f\left(β\_{k}\right)$$

Where:

* $β\_{k}$ is the current estimate of the parameters at iteration $k$.
* $η$ is the learning rate, a small positive scalar that controls the step size.
* $∇f\left(β\_{k}\right)$ is the gradient of the cost function $f$ with respect to $β$ at the current point $β\_{k}$.

The update rule comes from the idea of moving the parameter vector $β$ in the direction that decreases the cost function the most.

1. **Gradient**: The gradient $∇f\left(β\_{k}\right)$ represents the direction and magnitude of the steepest ascent of the function $f$ at the point $β\_{k}$. Since we want to minimize the function, we move in the opposite direction of the gradient.
2. **Step Size**: The term $η∇f\left(β\_{k}\right)$ scales the gradient by the learning rate $η$, determining how far we move in that direction. If $η$ is too large, the algorithm may overshoot the minimum; if it’s too small, the convergence will be slow.
3. **Iterative Update**: Starting from an initial guess $β\_{0}$, we repeatedly apply the update rule until the algorithm converges, meaning that the changes in $β\_{k}$ become negligible, and $β\_{k}$ is close to the optimal value $β^{\*}$.

### Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent is a variation of the vanilla gradient descent. Instead of computing the gradient using the entire dataset, SGD updates the parameters using only a single data point or a small batch of data points at each iteration. The later one we call it mini batch SGD.

Suppose our cost function is defined as the average over a dataset of size $n$:

$$f\left(β\right)=\frac{1}{n}\sum\_{i=1}^{n}f\_{i}\left(β\right)$$

Where $f\_{i}\left(β\right)$ represents the contribution of the $i$-th data point to the total cost function. The gradient of the cost function with respect to $β$ is:

$$∇f\left(β\right)=\frac{1}{n}\sum\_{i=1}^{n}∇f\_{i}\left(β\right)$$

Vanilla gradient descent would update the parameters as:

$$β\_{k+1}=β\_{k}−η∇f\left(β\_{k}\right)$$

Instead of using the entire dataset to compute the gradient, SGD approximates the gradient by using only a single data point (or a small batch). The update rule for SGD is:

$$β\_{k+1}=β\_{k}−η∇f\_{i\_{k}}\left(β\_{k}\right)$$

Where:

* $i\_{k}$ is the index of a randomly selected data point at iteration $k$.
* $∇f\_{i\_{k}}\left(β\_{k}\right)$ is the gradient of the cost function with respect to the parameter $β\_{k}$, evaluated only at the data point indexed by $i\_{k}$.

## Python Execution

### Synthetic Data

import numpy as np
from sklearn.linear\_model import LinearRegression
X=np.random.randn(1000,2)
y=3\*X[:,0]+2\*X[:,1]+1+np.random.randn(1000)

So for this project, our known relationship is $y=1+3x\_{1}+2x\_{2}+ξ$.

### Fit the data: Using scikit-learn Library

mlr=LinearRegression()
mlr.fit(X,y)
coefficients=mlr.coef\_.tolist()
slope=mlr.intercept\_.tolist()

So the model parameters: slope $β\_{0}=$ 0.9776 and coefficients $β\_{1}=$ 3.0198, and $β\_{2}=$ 2.0666

### Fit the data: Using Custom Library OLS

First we create our custom NewLinearRegression using the OLS formula above and save this python class as mlreg.py

import numpy as np

class NewLinearRegression:
 def \_\_init\_\_(self) -> None:
 self.beta = None

 def fit(self, X, y):
 X = np.concatenate([np.ones((len(X), 1)), X], axis=1)
 X\_transpose\_X = np.dot(X.transpose(), X)
 X\_transpose\_X\_inverse = np.linalg.inv(X\_transpose\_X)
 X\_transpose\_y = np.dot(X.transpose(), y)
 self.beta = np.dot(X\_transpose\_X\_inverse, X\_transpose\_y)

 def predict(self, X):
 X = np.concatenate([np.ones((len(X), 1)), X], axis=1)
 return np.dot(X, self.beta)

 def coeff\_(self):
 return self.beta[1:].tolist()

 def interceptt\_(self):
 return self.beta[0].tolist()

Now it’s time to use the new class

from mlreg import NewLinearRegression
mlr1 = NewLinearRegression()
mlr1.fit(X,y)
coefficients1=mlr1.coeff\_()
slope1=mlr1.interceptt\_()

So the model parameters: slope $β\_{0}=$ 0.9776 and coefficients $β\_{1}=$ 3.0198, and $β\_{2}=$ 2.0666

### Fit the data: Using Gradient Descent

We create the class

class GDLinearRegression:
 def \_\_init\_\_(self, learning\_rate=0.01, number\_of\_iteration=1000) -> None:
 self.learning\_rate = learning\_rate
 self.number\_of\_iteration = number\_of\_iteration
 self.weights = None
 self.bias = None

 def fit(self, X, y):
 num\_of\_samples, num\_of\_features = X.shape
 self.weights = np.zeros(num\_of\_features)
 self.bias = 0

 for \_ in range(self.number\_of\_iteration):
 y\_predicted = np.dot(X, self.weights) + self.bias

 d\_weights = (1 / num\_of\_samples) \* np.dot(X.T, (y\_predicted - y))
 d\_bias = (1 / num\_of\_samples) \* np.sum(y\_predicted - y)

 self.weights -= self.learning\_rate \* d\_weights
 self.bias -= self.learning\_rate \* d\_bias

 def predict(self, X):
 y\_predicted = np.dot(X, self.weights) + self.bias
 return y\_predicted

 def coefff\_(self):
 return self.weights.tolist()

 def intercepttt\_(self):
 return self.bias

Now we use this similarly as before,

from mlreg import GDLinearRegression
mlr2= GDLinearRegression(learning\_rate=0.008)
mlr2.fit(X,y)
coefficients2=mlr2.coefff\_()
slope2=mlr2.intercepttt\_()

So the model parameters: slope $β\_{0}=$ 0.9771 and coefficients $β\_{1}=$ 3.0184, and $β\_{2}=$ 2.066

### Fit the data: Using Stochastic Gradient Descent

First we define the class

class SGDLinearRegression:
 def \_\_init\_\_(self, learning\_rate=0.01, num\_iterations=1000, batch\_size=1) -> None:
 self.learning\_rate = learning\_rate
 self.num\_iterations = num\_iterations
 self.batch\_size = batch\_size
 self.theta = None
 self.mse\_list = None # Initialize mse\_list as an instance attribute

 def \_loss\_function(self, X, y, beta):
 num\_samples = len(y)
 y\_predicted = X.dot(beta)
 mse = (1/num\_samples) \* np.sum(np.square(y\_predicted - y))
 return mse

 def \_gradient\_function(self, X, y, beta):
 num\_samples = len(y)
 y\_predicted = X.dot(beta)
 grad = (1/num\_samples) \* X.T.dot(y\_predicted - y)
 return grad

 def fit(self, X, y):
 # Adding the intercept term (bias) as a column of ones
 X = np.concatenate([np.ones((len(X), 1)), X], axis=1)
 num\_features = X.shape[1]
 self.theta = np.zeros((num\_features, 1))

 self.mse\_list = np.zeros(self.num\_iterations) # Initialize mse\_list

 for i in range(self.num\_iterations):
 # Randomly select a batch of data points
 indices = np.random.choice(
 len(y), size=self.batch\_size, replace=False)
 X\_i = X[indices]
 y\_i = y[indices].reshape(-1, 1)

 # Compute the gradient and update the weights
 gradient = self.\_gradient\_function(X\_i, y\_i, self.theta)
 self.theta = self.theta - self.learning\_rate \* gradient

 # Calculate loss for the entire dataset (optional)
 self.mse\_list[i] = self.\_loss\_function(X, y, self.theta)

 return self.theta, self.mse\_list

 def predict(self, X):
 # Adding the intercept term (bias) as a column of ones
 X = np.concatenate([np.ones((len(X), 1)), X], axis=1)
 return X.dot(self.theta)

 def coef\_(self):
 # Return the coefficients (excluding the intercept term)
 return self.theta[1:].flatten().tolist()

 def intercept\_(self):
 # Return the intercept term
 return self.theta[0].item()

 def mse\_losses(self):
 # Return the mse\_list
 return self.mse\_list.tolist()

Now

import matplotlib.pyplot as plt
from mlreg import SGDLinearRegression
mlr3=SGDLinearRegression(learning\_rate=0.01, num\_iterations=1000, batch\_size=10)
theta, \_ = mlr3.fit(X, y)

So the model parameters: slope $β\_{0}=$ array([0.98750421]) and coefficients $β\_{1}=$ array([3.01288453]), and $β\_{2}=$ array([2.05965667])

Up next [knn regression](../../dsandml/knn/index.qmd)

**Share on**

**You may also like**