Chapter 4: Training Models

Warren Alphonso

March 2019

In this chapter, we explore a Linear Regression model, and two different ways to train it:

1. Using an equation to directly compute the model parameters that best fit the model to training set.
2. Using an iterative optimization process, called Gradient Descent.

1 Linear Regression

A linear model makes a prediction by simply computing a weighted sum of the input features and adding a constant called the *bias term*.

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

Figure 1: Linear Regression model prediction

- $\hat{y}$ 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.

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

Figure 2: Vectorized form of Linear Regression model prediction

- θ is the model's *parameter vector*, contianing bias term θ_0 and the feature weights θ_1 to θ_n.
- x is the instance's *feature vector*, containing x_0 to x_n, with x_0 always equal to 1.
- h_θ is the hypothesis function, using the model parameters θ.

We need to measure how well the model fits the training data. The most common performance measure of a regression model is the Root Mean Square Error (RMSE). Though, in practice, it is simpler to minimize Mean Square Error (MSE) which leads to the same result.

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

Figure 3: MSE cost function for Linear Regression model

1.1 The Normal Equation

To find the value of θ that minimizes the cost function, we use the *Normal Equation*, aka Least-Squares regression.

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

Figure 4: Normal Equation

- $\hat{\theta}$ is the value of θ that minimizes the cost function.
- y is the vector of target values containing y^1 to y^m.

Using `np.linalg.lstsq` does this for us. It computes $\hat{\theta} = X^+ y$, where X^+ is the *pseudoinverse* of X.
The psuedoinverse is computed using a matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose training set matrix X into the matrix multiplication of three matrices $U\Sigma V^T$. The pseudoinverse is computed as $X^+ = V\Sigma^+ U^T$. To compute the matrix Σ^+, the algorithm takes Σ and sets to zero all values smaller than a tiny threshold value, then it replaces all the non-zero values with their inverse, and finally it transposes the resulting matrix. This is more efficient than computing the Normal Equation, plus it can handle edge cases while Normal Equation won't work if matrix $X^T X$ is not invertible. The pseudoinverse is always defined.

1.2 Computational Complexity

The Normal Equation computes the inverse of $X^T X$ which is an $(n+1)(n+1)$ matrix where n is the number of features. The computational complexity of inverting this matrix is about $O(n^{2.4})$ to $O(n^3)$. SVD approach is about $O(n^2)$. Both Normal Equation and SVD approach get very slow when the number of features grows large, but both are linear with regards to number of instances in training set, so they handle large training sets efficiently.

2 Gradient Descent

Gradient Descent measures the local gradient of the error function with regards to the parameter vector θ, and it goes in the direction of descending gradient. Once gradient becomes zero, we're at the minimum. Concretely, we start by filling θ with random values (random initialization) and then improve it gradually until convergence. *learning rate* is an important parameter. If the learning rate is too small, then the algorithm will have to go through many iterations to convergence. If it's too high, algorithm diverges with larger and larger values. There are two main challenges with Gradient Descent:

1. a plateau might cause convergence to take a long time
2. getting trapped in a local minimum which prevents us from reaching the global minimum

Fortunately, MSE for Linear Regression models happens to be a convex function so there are no local minima. It's also a continuous function whose slope never changes abruptly. These two things mean Gradient Descent is guaranteed to approach the global minimum. It is *vital* when using Gradient Descent to ensure that all features have a similar scale (using Scikit-Learn's `StandardScalar` class), or it will take much longer to converge.

2.1 Batch Gradient Descent

To implement Gradient Descent, we need to compute the partial derivative of the cost function with regards to each model parameter θ_j.

$$\Delta_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial\theta_0} MSE(\theta) \\ \frac{\partial}{\partial\theta_1} MSE(\theta) \\ \dots \\ \frac{\partial}{\partial\theta_n} MSE(\theta) \end{pmatrix} = \frac{2}{m} X^T (X\theta - y)$$

Figure 5: Gradient vector of the cost function

Note that this formula does calculations over the *full* training set X at each Gradient Descent step, hence the name batch Gradient Descent. It is very slow on large training sets. However, Gradient Descent scales well with number of features, so training a Linear Regression model with hundreds of thousands of features is much faster using Gradient Descent as opposed to the Normal Equation or SVD.

One we we have the gradient vector, which points uphill, we just go in the opposite direction. This means we subtract $\Delta_\theta MSE(\theta)$ from θ. This is where learning rate η comes into play. Multiply the gradient vector by η to determine size of downhill step.

$$\theta^{(next)} = \theta - \eta \Delta_\theta MSE(\theta)$$

Figure 6: Gradient Descent step

To find a good learning rate, you can use a grid search. We set a large number of iterations but interrupt the algorithm when the gradient vector becomes tiny.

2.2 Stochastic Gradient Descent

The main problem with Batch Gradient Descent is it uses the entire training set to compute gradients at every step. At the opposite extreme Stochastic Gradient Descent picks a random instance in the training set at every step and computes the gradient based only on that instance. This makes it possible to train on huge training sets since only one instance needs to be in memory at each iteration. Due to its stochastic (random) nature, this algorithm is much less regular than BGD. Instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. This means the algorithm is more likely to jump out of local minima.
Ultimately, randomness is good to escape local minima but bad because the algorithm can never settle at the minima. One solution is to gradually reduce the learning rate. The step size starts out large so we can escape local minima, but gets smaller which allows the algorithm to settle at the global minimum.

2.3 Mini-batch Gradient Descent

Instead of computing the gradients based on the full training set (BGD) or based on just one instance (SGD), Mini-batch Gradient Descent computes gradients on small random sets of instances called *mini-batches*. The main advantage to MGD is it allows a performance boost from hardware optimization of matrix operations, especially when using GPUs.

3 Polynomial Regression

Surprisingly, we can actually use a linear model to fit *nonlinear* data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This is known as *Polynomial Regression*.

4 Learning Curves

High-degree Polynomial Regression will likely overfit the training data. As an example, suppose we have a dataset that follows a quadratic model. A linear model will clearly underfit, but a 300-degree polynomial will severely overfit.
Previously, we used cross-validation to estimate a model's generalization. Cross-validation says a model that fit the training data well, but performs poorly on testing data, then the model is overfitting.
Another way to estimate generalization is through *learning curves*: plots of the model's performance on training set and validation set as a function of training

set size.
The learning curve of an **underfitting** model shows that training set loss and validation set loss are close together and fairly high. The learning curve of an **overfitting** model has low error on training data but high error for validation, with a large gap between the two accuracies.

The Bias/Variance Tradeoff A model's generalization error can be expressed as the sum of three very different errors:

- *Bias*: this part of generalization error is due to wrong assumptions, such as assuming the data is linear when it is actually quadratic. A high-bias model is most likely to underfit training data.
- *Variance*: this part is due to the model's excessively sensitivity to small variations in the training data. A model with many degrees of freedom is likely to have high variance and thus overfit the training data.
- *Irreducible error*: this part is due to the noisiness of the data itself. The only way to reduce this is to clean up the data.

Increasing a model's complexity typically increases variance and reduces bias. On the other hand, reducing a model's complexity decreases variance and increases bias.

5 Regularized Linear Models

A good way to reduce overfitting is to regularize the model: the fewer degrees of freedom, the harder it is to overfit the data. Regularization is achieved by constraining the weights of the model.

5.1 Ridge Regression

Ridge Regression is a regularized version of Linear Regression. We add a *regularization term* equal to $\alpha \sum_{i=1}^{n} \theta_i^2$ to the cost function. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. Note the regularization term should only be added to the cost function during training. The hyperparameter α controls how much you want to regularize the model. If α is very large, then all weights end up close to zero.

$$J(\theta) = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

Figure 7: Ridge Regression cost function

Note that bias term θ_0 is not regularized since the sum starts at i=1, not 0. It is also crucial to scale the data before performing Ridge Regression because it is sensitive to the scale of the input features. This is true of most regularized models.

5.2 Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression (Lasso Regression) also adds a regularization term to the cost function, but it uses the ℓ_1 norm of the weight vector instead of half of the square of the ℓ_2 norm.

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} \mid \theta_i \mid$$

Figure 8: Lasso Regression cost function

Lasso Regression tends to completely eliminate the weights of the least important features (ie set them to zero). In other words, Lasso Regression automatically performs feature selection and outputs a *sparse model*.

5.3 Elastic Net

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The Elastic Net cost function has both ℓ_1 and ℓ_2 norm.
When should you use regression? It is almost always preferable to have regularization. You should prefer Lasso or Elastic Net since they tend to reduce useless features' weights down to zero. In general, Elastic Net is preferred over Lasso since Lasso may behave erratically when number of features is greater than number of training instances.

6 Logistic Regression

Logistic Regression is commonly used to estimate probability that an instance belongs to a particular class. Just like Linear Regression, a Logistic Regression model computes a weighted sum of the input features plus a bias term, but instead of outputting the result directly like Linear Regression, it outputs the *logistic* of this result.

$$\hat{p} = h_\theta(x) = \sigma(\theta^T x)$$

Figure 9: Logistic Regression model estimated probability

The logistic - noted σ - is a *sigmoid function*.

$$\sigma(t) = \frac{1}{1 + exp(-t)}$$

Figure 10: Logistic function

6.1 Training and Cost Function

The model should estimate high probabilities for positive instances and low probabilities for negative instances.

$$c(\theta) = \begin{cases} -log(\hat{p}) & \text{if } y = 1 \\ -log(1-\hat{p}) & \text{if } y = 0 \end{cases}$$

This makes sense: $-log(t)$ grows very large when t approaches 0, so cost function is large is model estimates a probability close to 0 for positive instance and will also be large if model estimates a probability close to 1 for a negative instance. There is no known closed-form equation to compute the value of θ that minimizes this cost function. However, since this cost function is convex, Gradient Descent is guaranteed to find the global minimum.

6.2 Softmax Regression

The Logistic Regression model can be generalized to support multiple classes directly. This is called *Softmax Regression*. When given an instance x, the Softmax Regression model first computes a score $s_k(x)$ for each class k, then estimates the probability of eachc class by applying the *softmax function*, also known as the normalized exponential.

$$s_k(x) = (\theta^{(k)})^T x$$

Figure 11: Softmax score for class k

Each class has its own dedicated parameter vector $\theta^{(k)}$. All these vectors are stored as rows in a parameter matrix Θ.
Once we have computed the score for every class for the instance x, we can estimate probability $\hat{p}_k$ that the instance belongs to class k by running the score through the softmax function.

$$\hat{p}_k = \sigma(s(x))_k = \frac{exp(s_k(x))}{\sum_{j=1}^{K} exp(s_j(x))}$$

Figure 12: Softmax function

- K is the number of classes.
- $s(x)$ is a vector containing scores of each class for the instance x.
- $\sigma(s(x))_k$ is the estimated probability that the instance x belongs to class k given the scores of each class for that instance.

Softmax Regression is just brute forcing each class to see which outputs the highest probability. Note that Softmax can only output one class at a time.