Module-4.6-Stochastic Gradient Descent

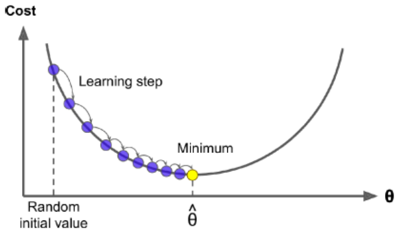
Presented by Yasin Ceran

Table of Contents

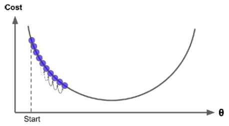
Introduction to Gradient Descent

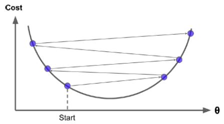
- Tweak parameters iterative to minimize a cost function, $L(\theta)$
- Initialize θ_0 with random values (random initialization)
- Then update θ in steps until it converges to a local minimum

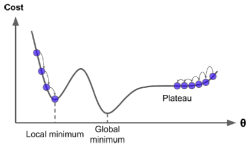
$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta_i \frac{d}{d\theta} L(\theta^{(i)})$$



Gradient Descent Convergence

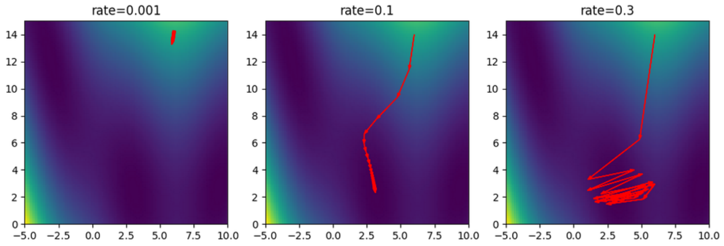






$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta_i \frac{d}{d\theta} L(\theta^{(i)})$$

Pick a Learning Rate



$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta_i \frac{d}{d\theta} L(\theta^{(i)})$$

Batch Gradient Descent

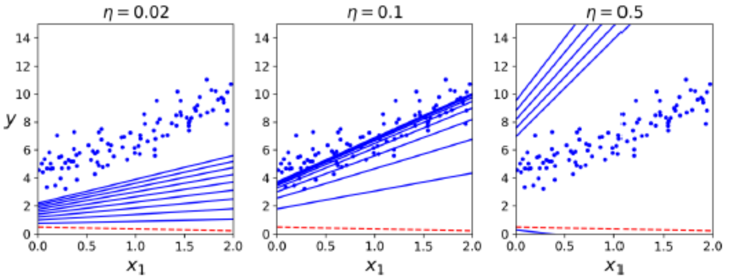
- Compute the gradient of the cost function wrt each model parameter θ_j–> Partial derivative
- $\frac{\partial MSE(\theta)}{\partial \theta_j} = \frac{2}{n} \sum_{i=1}^{m} \left(\theta^T x^{(i)} - y^{(i)}\right) x_j^{(i)}$
- Instead of computing partial derivatives individually, we can compute the gradient vector:

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

- This formula involves calculations over the full training set X, at each Gradient Descent step, hence Batch Gradient Descent.
- $\theta^{(next\ \ step)} = \theta - \eta \Delta_{\theta MSE(\theta)}$

Batch Gradient Descent Example

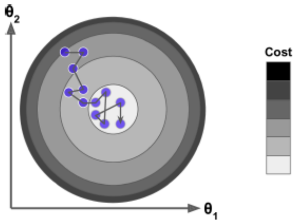
```python
eta = 0.1 # learning rate
n_iterations = 1000
n = 100
theta = np.random.randn(2,1) # random initialization
for iteration in range(n_iterations):
    gradients = 2/n * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```



Stochastic Gradient Descent

- picks a random instance at every step
- much faster but less regular

```python
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters
def learning_schedule(t):
    return t0 / (t + t1)
theta = np.random.randn(2,1) # random initialization
for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```



SGDClassifier and SGDRegressor

- you can use the SGDRegressor class, which defaults to optimizing the squared error cost function

```python
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())

sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

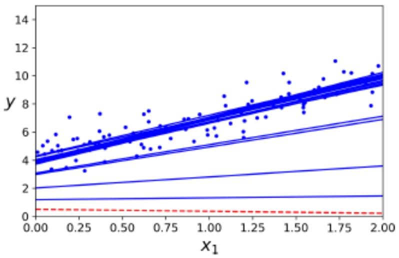


Table of Contents

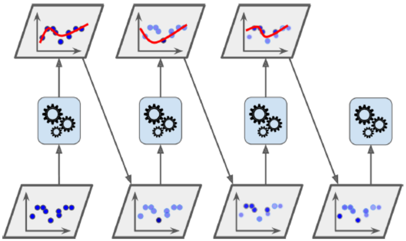
Boosting

- Boosting refers to any Ensemble method that can combine several weak learners into a strong learner.
- Train predictors sequentially, each trying to correct its predecessor.
- AdaBoost (Adaptive Boosting), Gradient Boosting, XGBoosting, ...

AdaBoost

- One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted.
- This results in new predictors focusing more and more on the hard cases.
- It cannot be parallelized

```python
from sklearn.ensemble import AdaBoostClassifier
ada_clf = AdaBoostClassifier(
      DecisionTreeClassifier(max_depth=1), n_estimators=200,
      algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```



Gradient Boosting

- Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor
- Instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the residual errors made by the previous predictor

$$f_1(x) \approx y$$

$$f_2(x) \approx y - f_1(x)$$

$$f_3(x) \approx y - f_1(x) - f_2(x)$$

```python
from sklearn.tree import DecisionTreeRegressor
tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)

y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)

y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)

y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

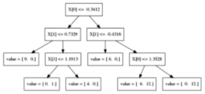
Gradient Boosting, Cont.

$$f_1(x) \approx y$$

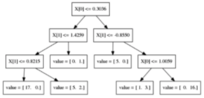
$$f_2(x) \approx y - \gamma f_1(x)$$

$$f_3(x) \approx y - \gamma f_1(x) - \gamma f_2(x)$$

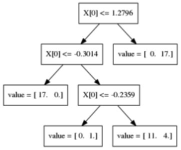
$y \approx \gamma$



$+ \gamma$



$+ \gamma$



$+ \ldots$

Learning rate $\gamma, i.e. 0.1$

Gradient Boosting is Gradient Descent

Linear regression

$$L(\mathbf{x}_i, y_i, \mathbf{w}, b) = \sum_i (y_i - \hat{y}_i)^2$$

$$= \sum_i (y_i - w^T \mathbf{x}_i - b)^2$$

optimize:

$$\min_{w \in \mathbb{R}^p, b \in \mathbb{R}} \sum_{i=1}^{n} (y_i - w^T \mathbf{x}_i - b)^2$$

gradient descent:

$$w_{j+1} = w_j - \gamma \frac{\partial L(\mathbf{x}_i, y_i, \mathbf{w}, b)}{\partial \mathbf{w}}$$

Gradient Boosting

$$L(y_i, \hat{y}_i) = \sum_i (y_i - \hat{y}_i)^2$$

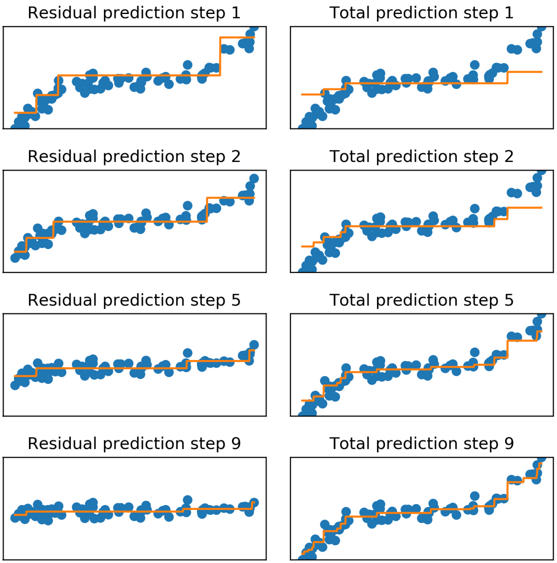
optimize:

$$\min_{\hat{y} \in \mathbb{R}^n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

gradient descent:

$$\hat{y}_{j+1} = \hat{y}_j - \gamma \frac{\partial L(y_i, \hat{y}_i)}{\partial \hat{y}}$$

Gradient Boosting Regressor



Summary

When to use tree-based models:

- Model non-linear relationships
- Doesn't care about scaling, no need for feature engineering
- Single tree: very interpretable (if small)
- Random forests very robust, good benchmark