Московский государственный технический
университет им. Н.Э. Баумана

Факультет «Информатика и системы управления»
Кафедра ИУ5 «Системы обработки информации и управления»

Курс «Парадигмы и конструкции языков программирования»
Отчет по Домашнему заданию
«Методы оптимизации градиентного спуска»

Выполнил:
студент группы ИУ5-33Б
Громов Владислав

Проверил:
преподаватель каф. ИУ5
Гапанюк Ю.Е.

Москва, 2024 г.

Постановка задачи

Реализовать несколько методов оптимизации градиентного спуска на линейной модели (SGD, SAG, momentum gradient, adaptive gradient)

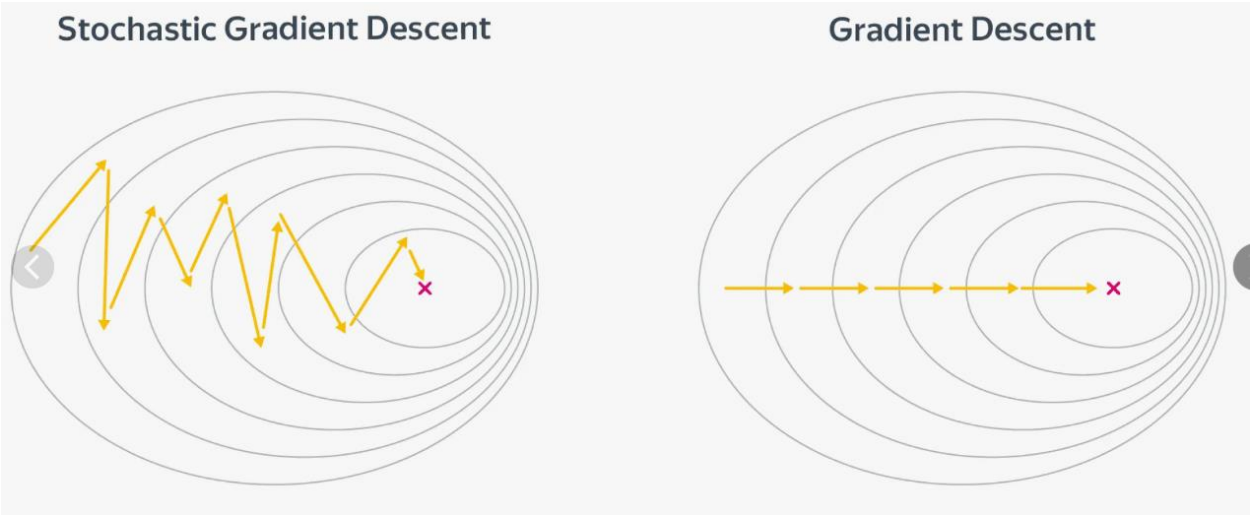
Для начала, немного подготовим данные:

```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.base import *
import numpy as np
import pandas as pd
from math import asin, cos, sqrt, radians
from sklearn.preprocessing import StandardScaler
from typing import Optional, List
from sklearn.metrics import mean_absolute_error, r2_score
import matplotlib.pyplot as plt



data = pd.read_csv('csv/data.csv')


seed = 25
np.random.seed(seed)


test_size = 0.2
data_train, data_test, Y_train, Y_test = train_test_split(
    data[data.columns.drop("Sale_Price")],
    np.array(data["Sale_Price"]),
    test_size=test_size,
    random_state=seed)
```

```
target_column = "Sale_Price"
continuous_columns = [key for key in data.keys() if data[key].dtype in ("int64", "float64")]
continuous_columns.remove(target_column)



def root_mean_squared_logarithmic_error(y_true, y_pred, a_min=1.):
    assert np.all(y_true >= 0)
    y_pred = np.maximum(y_pred, a_min)
    return np.sqrt(np.mean((np.log(y_true) - np.log(y_pred))**2))



class SmartDataPreprocessor(TransformerMixin):
    def __init__(self, needed_columns: Optional[List[str]]=None):
        self.center_coords = (42.025333, -93.633801)
        self.needed_columns = needed_columns
        self.scaler = StandardScaler()
        self.median_lot_frontage = None

    # формула гаверсина для нахождения расстояние
    def distance(self, lat1, long1, lat2, long2):
        r = 6371
        long1r = radians(long1)
        lat1r = radians(lat1)
        long2r = radians(long2)
        lat2r = radians(lat2)
        d = 2 * r * asin(sqrt((1-cos(lat2r-lat1r) + cos(lat1r) * cos(lat2r) * (1 - cos(long2r - long1r)))/2))
        return round(d, 2)
```

```
    def fit(self, X):
        self.median_lot_frontage = np.median(X['Lot_Frontage'])
        X['Distance_to_the_center'] = X.apply(lambda row:
self.distance(row['Latitude'], row['Longitude'], self.center_coords[0],
self.center_coords[1]), axis=1)
        self.scaler.fit(data[self.needed_columns])
        return self


    def transform(self, X):
        X = X[self.needed_columns]
        X['Lot_Frontage'] = X['Lot_Frontage'].replace({0:
np.NaN}).fillna(self.median_lot_frontage)
        X = self.scaler.transform(X)
        return X


preprocessor = SmartDataPreprocessor(needed_columns=continuous_columns)


X_train = preprocessor.fit_transform(data_train)
X_test = preprocessor.transform(data_test)
```

Так как, нахождение градиента на всей выборке очень затратная операция придумали метод в котором на каждой итерации берется градиент либо одного элемента либо батча из некоторого количества элементов (32, 64, 128), эксперимент показал, что хотя спуск к минимуму не будет таким плавным, тем не менее матожидание стохастического градиентного спуска является несмещенной оценкой полного градиента.



Теперь реализуем SGDlinearregressor с L2-регуляризацией

$$\vec{w}, b = argmin_{\vec{w},b}(L); L = \frac{1}{N}\sum(y_i - \hat{y_i})^2 + \overrightarrow{w^T}\vec{w}$$

$$\nabla_b L = \frac{2}{N}\sum(X\vec{w} + b - \vec{y})$$

$$\nabla_{\vec{w}} L = \frac{2}{N}X^T(X\vec{w} + b - \vec{y}) + 2\vec{w}$$

Класс должен инициализироваться со следующими гиперпараметрами:

a. lr — learning rate. Длина шага градиентного спуска

b. regularization — коэффициент λ из формулы выше

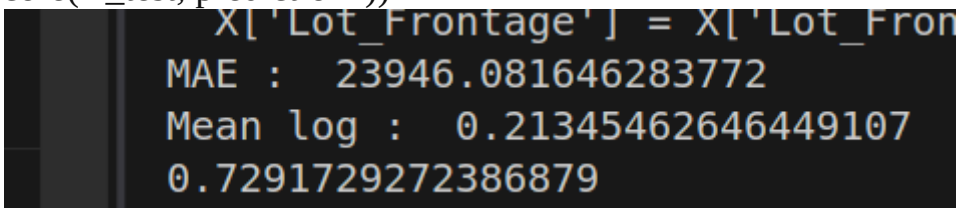
c. delta_converged — устанавливает условие окончание обучение. В тот момент когда норма разности весов на соседних шагах градиентного спуска меньше чем delta_converged алгоритм перкращает обновлять веса

d. max_steps — максимальное число шагов градиентного спуска

e. batch_size — размер батча

```
class SGDLinearRegressor(RegressorMixin):
    def __init__(self,
                 lr=0.01, regularization=1., delta_converged=1e-3, max_steps=1000,
                 batch_size=64, p=0.32):
        self.lr = lr
        self.regularization = regularization
        self.max_steps = max_steps
        self.delta_converged = delta_converged
        self.batch_size = batch_size
            self.p = p
        self.W = None
        self.b = None

    def fit(self, X, Y):

        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)
        self.b = 0
        self.SGD(n_samples, n_features, X, Y)

    def SGD(self, n_samples, n_features, X, Y):
        res = [self.w.copy()]
        for step in range(self.max_steps):
            indices = np.arange(n_samples)
            np.random.shuffle(indices) # перемешиваем индексы
            X_shuffled = X[indices]
            Y_shuffled = Y[indices]
            # создание батчей
            for i in range(0, n_samples, self.batch_size):
                end_inx = i + self.batch_size
                X_batch = X_shuffled[i:end_inx]
                Y_batch = Y_shuffled[i:end_inx]
                # предсказание
                Y_pred = np.dot(X_batch, self.w) + self.b
                f = Y_pred - Y_batch
                # вычисление градиентов
                grad_w = (2 / self.batch_size) * np.dot(X_batch.T, f) + 2 *
self.regularization * self.w
                grad_b = (2 / self.batch_size) * np.sum(f)
```

```
            # обновление весов
            self.w -= self.lr * np.power((1 / (1 + step)), self.p) * grad_w
            self.b -= self.lr * np.power((1 / (1 + step)), self.p) * grad_b

            res.append(self.w.copy())

        if np.linalg.norm(self.lr * grad_w) < self.delta_converged:
            break


    # предсказание
    def predict(self, X):
        return np.dot(X, self.w) + self.b

model = SGDLinearRegressor()нианcы
model.fit(X_train, Y_train)
prediction = model.predict(X_test)
print("MAE : ", mean_absolute_error(Y_test, prediction2))
print("Mean log : ", root_mean_squared_logarithmic_error(Y_test, prediction2))
print(r2_score(Y_test, prediction2))
```



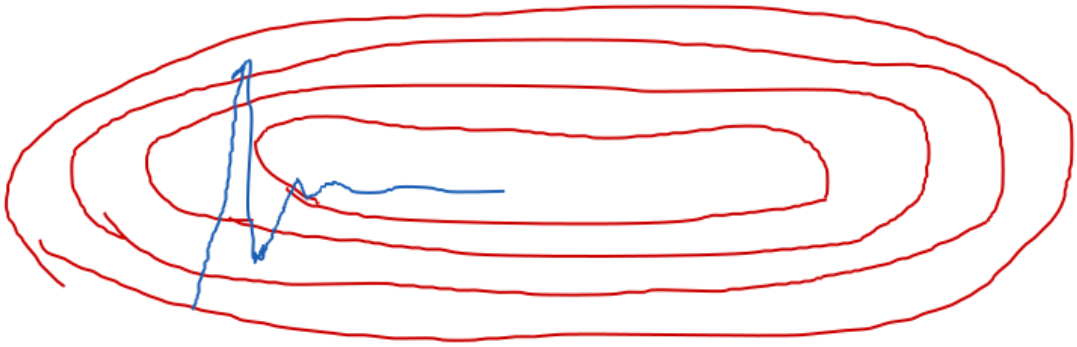
Реализация SAG похожа на SGD, но есть некоторые нюансы.
Во-первых сначала вычислим средний градиент на всей выборке . Теперь на каждой итерации мы будем обновлять градиент рандомного одного объекта и пересчитывать средний градиент. Практика показывает, что такой метод тоже будет сходиться к минимуму.

```
class SAGLinearRegressor(RegressorMixin):
    def __init__(self,
                 lr=0.01, delta_converged=1e-5, max_steps=1000):
        self.lr = lr
        self.max_steps = max_steps
```

```
        self.delta_converged = delta_converged
        self.W = None

    def fit(self, X, Y):
        X = np.hstack([X, np.ones([X.shape[0], 1])])
        n_samples, n_features = X.shape
        self.W = np.zeros(n_features)
        res = [self.W.copy()]
        self.SAG(n_samples, n_features, X, Y)

    def SAG(self, n_samples, n_features, X, Y):
        # ищем средний градиент по всей выборке
        memgrad = np.zeros((n_samples, n_features))
        for i in range(n_samples):
            Y_pred = np.dot(X[i], self.W)
            f = Y_pred - Y[i]
            memgrad[i] = 2 * X[i] * f
            # memgrad[i] = 2 * X[i] * (X[i].dot(self.w) - Y[i])
        avg_memgrad = memgrad.mean(axis=0)

        for step in range(self.max_steps):
            # пересчитываю градиент случайного элемента
            i = np.random.randint(0, n_samples)
            Y_pred = np.dot(X[i], self.W)
            f = Y_pred - Y[i]
            memgrad[i] = 2 * X[i] * f
            avg_memgrad = memgrad.mean(axis=0)
            self.W -= self.lr * avg_memgrad
            if np.linalg.norm(self.lr * avg_memgrad) < self.delta_converged:
                break
    # предсказание
    def predict(self, X):
        X = np.hstack([X, np.ones([X.shape[0], 1])])
        return np.dot(X, self.W)

model2 = SAGLinearRegressor()
model2.fit(X_train, Y_train)
prediction2 = model.predict(X_test)
print("MAE : ", mean_absolute_error(Y_test, prediction2))
print("Mean log : ", root_mean_squared_logarithmic_error(Y_test, prediction2))
print(r2_score(Y_test, prediction2))
```

Перейдем к momentum grad. Рассмотрим одну проблему: если у нас линии уровня очень вытянуты (а это очень часто бывает) и я делаю град спуск на этих линиях уровня. Т.к градиент ортогонален линиям уровня

Процедура сходимости будет осциллировать. По оси Y у нас осциллирующее движение, а по оси X у нас стабильное. Появилась идея устранить осцилляцию путем сложения или усреднения векторов. Для этого введем дополнительную переменную h0 = 0
Сначала обновим вектор инерции $h_k = ah_{k-1} + lr\nabla_{\vec{w}}Q\left(w^{(k-1)}\right)$
Потом обновим вес $w^{(k)} = w^{(k-1)} - h_k$
hk — это затухающее среднее градиентов с прошлых шагов



Реализация:

```
        class MomentumGradient(RegressorMixin):
    def __init__(self, learning_rate=0.01, momentum=0.9, max_steps=1000,
delta_converged=1e-5):
        self.learning_rate = learning_rate  # Скорость обучения
        self.momentum = momentum  # Параметр момента
        self.max_steps = max_steps
        self.delta_converged = delta_converged
        self.w = None
        self.b = None
        self.v = None  # Вектор инерции


    def fit(self, X, y):
        X = np.hstack([X, np.ones([X.shape[0], 1])])
        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)
        self.b = 0
        self.v = np.zeros(n_features)  # Вектор скорости для коэффициентов и
свободного члена
        for i in range(self.max_steps):

            gradients = self._compute_grad(X, y)  # Вычисление градиентов
            self.v = self.momentum * self.v + self.learning_rate * gradients  #
Обновление вектора скорости
            self.w -= self.v

            # Проверка сходимости
            if np.linalg.norm(self.v) < self.delta_converged:
                break
```

```
        return self

    def _compute_grad(self, X, y):
        n_samples = X.shape[0]
        y_pred = np.dot(X, self.w)  # Предсказание
        error = y_pred - y  # Ошибка
        gradients = np.dot(X.T, error) / n_samples  # Средний градиент
        return gradients


    def predict(self, X):
        X = np.hstack([X, np.ones([X.shape[0], 1])])
        return np.dot(X, self.w) # Предсказание
```

Адаптивный шаг.

Представим что у нас обучающая выборка где какие-то признаки очень разрежены. Если мы делаем SGD, то мы по очереди высчитываем градиент каждого элемента и по мере прохождения наша длина шага постепенно уменьшается.
Частная производная по какому-то весу не нулевая только если признак соответствующий не нулевой.
И допустим только на 100 шаге я впервые обновлю вес какого-то разреженного признака, а до этого я уже 100 раз обновил длину шага и она стала маленькой и получается, что этот вес плохо обучится. Возникает идея сделать длину шага свою для каждого веса модели.
Для этого используем новую переменную G0 = 0

Обновляем $G_{kj} = G_{k-1,j} + \left(\nabla Q\left(w^{(k-1)}\right)\right)_j^2$

Обновляем вес $w_j^{(k)} = w_j^{(k-1)} - \frac{lr}{\sqrt{(G_{kj}+epsilon)}} \nabla Q * \left(w^{(k-1)}\right)$

Реализация:

```
class AdaGradRegressor(RegressorMixin):
    def __init__(self, lr=0.1, epsilon=1e-8, max_steps=2000,
delta_converged=1e-3):
        self.lr= lr
```

```
        self.epsilon = epsilon
        self.max_steps = max_steps
        self.delta_converged = delta_converged
        self.w = None
        self.b = None
        self.G = None


    def fit(self, X, y):
        X = np.hstack([X, np.ones([X.shape[0], 1])])
        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)
        self.G = np.zeros(n_features)
        self.sum_squared_gradients = np.zeros(n_features)
        for _ in range(self.max_steps):
            gradients = self._compute_grad(X, y)
            self.G += gradients**2  # Обновляем накопленную сумму
квадратов градиентов
            adaptive_lr = self.lr / (np.sqrt(self.G) + self.epsilon)
#адаптивная скорость обучения
            self.w -= adaptive_lr * gradients
            if np.linalg.norm(self.lr * gradients) < self.delta_converged:
                break

    def _compute_grad(self, X, y):
        y_pred = np.dot(X, self.w)
        error = y_pred - y
        gradients = np.dot(X.T, error) / X.shape[0]
        return gradients

    def predict(self, X):
        X = np.hstack([X, np.ones([X.shape[0], 1])])
        return X @ self.w
```