AI and Machine Learning, Homework2

Author: Ying Yiwen
Number: 12210159

Contents

1 Code Implementation

1.1 MSE Loss

use the formula:

$$l(\omega) = \frac{(y_{pred} - y_{true})^2}{2N} \tag{1}$$

```
def calculate_loss(self, y, y_pred):
    # MSE loss
    N = y.size
    return np.sum((y_pred - y) ** 2) / (2 * N)
```

1.2 Update Method

train function with BGD method:

```
def train_BGD(self, X_train, y_train, X_test, y_test):
    # BGD
    self.W = 10*np.random.rand(X_train.shape[1] + 1)
    X_train, X_test = self.preprocess_data_X(X_train), self.preprocess_data_X(
        X_test)
    for _ in range(self.n_iter):
        # compute loss
        y_pred = self.predict(X_train)
        train_loss = self.calculate_loss(y_train, y_pred)
        self.train_loss.append(train_loss)
        y_pred_test = self.predict(X_test)
        test_loss = self.calculate_loss(y_test, y_pred_test)
        self.test_loss.append(test_loss)
        # gradient descent
        self.W -= self.lr * self.gradient(X_train, y_train, y_pred)
```

train function with SGD method:

```
def train_SGD(self, X_train, y_train, X_test, y_test):
    # SGD
    self.W = 10*np.random.rand(X_train.shape[1] + 1)
    X_train, X_test = self.preprocess_data_X(X_train), self.preprocess_data_X(
        X_test)
    N = y_train.size
    for _ in range(self.n_iter):
        # shuffle the data
        indices = np.random.permutation(N)
        for i in indices:
            x_i = X_train[i:i+1]
            y_i = y_train[i:i+1]
            # compute loss
            y_pred = self.predict(x_i)
            train_loss = self.calculate_loss(y_i, y_pred)
            self.train_loss.append(train_loss)
            y_pred_test = self.predict(X_test)
            test_loss = self.calculate_loss(y_test, y_pred_test)
            self.test_loss.append(test_loss)
            # compute gradient
```

```
            grad = self.gradient(x_i, y_i, y_pred)
            self.W -= self.lr * grad
```

train function with MBGD method:

```
def train_MBGD(self, X_train, y_train, X_test, y_test):
    # MBGD
    self.W = 10*np.random.rand(X_train.shape[1] + 1)
    X_train, X_test = self.preprocess_data_X(X_train), self.preprocess_data_X(
        X_test)
    N = y_train.size
    for _ in range(self.n_iter):
        # shuffle the data
        indices = np.random.permutation(N)
        for start in range(0, N, self.batch_size):
            end = min(start + self.batch_size, N)  # not out of range
            # get batch data
            batch_indices = indices[start:end]
            X_batch = X_train[batch_indices]
            y_batch = y_train[batch_indices]
            # predict and compute loss
            y_pred = self.predict(X_batch)
            train_loss = self.calculate_loss(y_batch, y_pred)
            self.train_loss.append(train_loss)
            y_pred_test = self.predict(X_test)
            test_loss = self.calculate_loss(y_test, y_pred_test)
            self.test_loss.append(test_loss)
            # compute gradient
            grad = self.gradient(X_batch, y_batch, y_pred)
            self.W -= self.lr * grad
```

1.3 Normalization

There should be normalization function for x and inverse normalization for W. As follows:
min-max normalization:

```
def min_max_normalization(self, x):
    # normalize the data by min-max normalization
    _min = np.min(x, axis=0)
    _max = np.max(x, axis=0)
    _range = _max - _min
    normalized = (x - _min) / _range
    return normalized, _min, _max

def inverse_min_max_weight(self, W, _min, _max):
    # inverse min-max normalization
    W[:1] -= W[1:] * _min / (_max - _min)
    W[1:] /= (_max - _min)
    return W
```

mean normalization:

```
def mean_normalization(self, x):
    # normalize the data by mean normalization
    mu = np.mean(x, axis=0)
```

```
    sigma = np.std(x, axis=0)
    normalized = (x - mu) / sigma
    return normalized, mu, sigma

def inverse_mean_weight(self, W, mu, sigma):
    # inverse mean normalization
    W[:1] -= W[1:] * mu / sigma
    W[1:] /= sigma
    return W
```

1.4 Random Split

The input data should be split into train set and test set randomly. Shuffle the data also prevent the fitting pay importance to too few features.

```
def random_split(self, x, y, test_size=0.1, random_state=None):
    # shuffle the data
    if random_state is not None:
    np.random.seed(random_state)
    indices = np.arange(len(x))
    np.random.shuffle(indices)
    # split the data
    split_index = int(len(x) * (1-test_size))
    train_indices, test_indices = indices[:split_index], indices[split_index:]
    xtrain, ytrain = x[train_indices], y[train_indices]
    xtest, ytest = x[test_indices], y[test_indices]
    return xtrain, ytrain, xtest, ytest
```

1.5 Main Function

With all the function written, here's the main function:

```
if __name__ == "__main__":
    # record the time
    start_time = time.time()

    # generate data
    X = np.arange(100).reshape(100,1)
    a, b = 1, 10
    y = a * X + b + np.random.normal(0, 5, size=X.shape)
    y = y.reshape(-1)

    # set parameters
    n_iter, lr = 30000, 5e-4
    model = LinearRegression(n_iter=n_iter, lr=lr)

    # normalize the data
    method = 'min_max'
    X_normalized, param1, param2 = model.normalize(X, method=method)

    # split the data
    X_train, y_train, X_test, y_test = model.random_split(X_normalized, y,
        test_size=0.2)
```

```python

    # train the model
    model.train(X_train, y_train, X_test, y_test, method='MBGD')
    model.plot_loss()
    end_time = time.time()

    # compute loss
    train_loss = model.calculate_loss(y_train, model.predict(model.preprocess_data_X(X_train)))
    test_loss = model.calculate_loss(y_test, model.predict(model.preprocess_data_X(X_test)))

    # inverse normalization
    if method == 'min_max':
    W_original = model.inverse_min_max_weight(model.W, param1, param2)
    elif method == 'mean':
    W_original = model.inverse_mean_weight(model.W, param1, param2)
    else:
    W_original = model.W

    # output the result
    model.plot_fit(X, y)
    print(f'n of iteration: {n_iter}, loss rate: {lr}')
    print(f'Learned weights: {W_original}, Training loss: {train_loss}, Testing loss: {test_loss}')
    print(f'Time: {end_time - start_time}s')
```

2 Analyzation of Different Methods

2.1 Different Update Methods

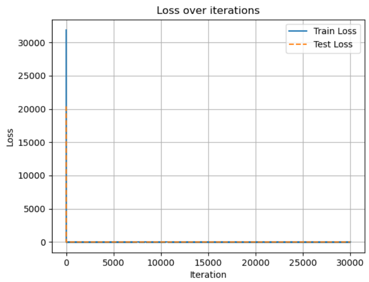
To control varaible, we don't use any normalization method, to see the difference of update methods only.

MBGD:
n of iteration: 10000, loss rate: 0.0002
Learned weights: [10.15597587 1.0220182], Training loss: 14.307060377519765, Testing loss: 9.575175286124201
Time: 4.011836767196655s

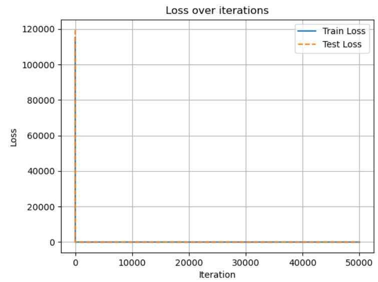
BGD:
n of iteration: 50000, loss rate: 0.0001
Learned weights: [9.80610593 1.00979871], Training loss: 11.736946331559594, Testing loss: 13.800069762104055
Time: 3.438744068145752s

SGD:
n of iteration: 2000, loss rate: 0.0001
Learned weights: [10.24699952 1.03413564], Training loss: 12.558711291049434, Testing loss: 8.769263129234005
Time: 6.4901885986328125s

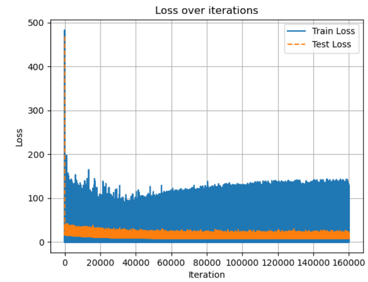
BGD can compute parallelly, but it use too much samples at a time, so it works slowly and needs large memory consume. BGD also may falls to local minimum as the loss function may be to smooth. But BGD can cope with input noises, so it can converge well.



(a) MBGD Loss Function

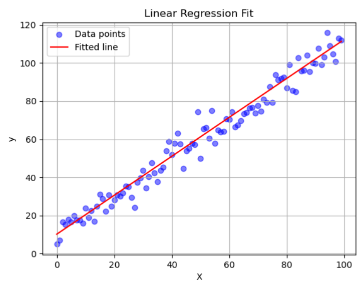


(b) BGD Loss Function

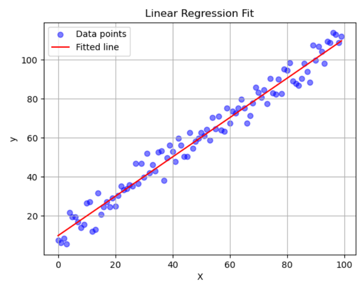


(c) SGD Loss Function

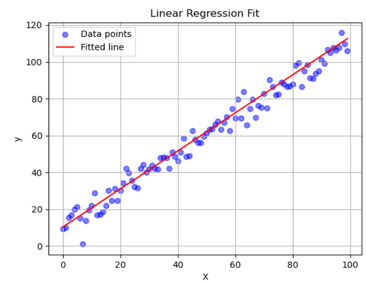
Fig. 1: The Trend of Loss Function of Different Gradient Method



(a) MBGD Fitting Results



(b) BGD Fitting Results



(c) SGD Loss Function

Fig. 2: The Fitting Results of Different Gradient Method

SGD only use one sample at a time, so it uses less memory and works quickly. Thanks to the noises in input data, SGD can easily jump out of local minimum. However, when it actually converges, those noises may make SGD harder to get the global minimum, so it doesn't converge very well.

MBGD take the advantage of both methods, but it need suitable choice of batch size to balance the influence.

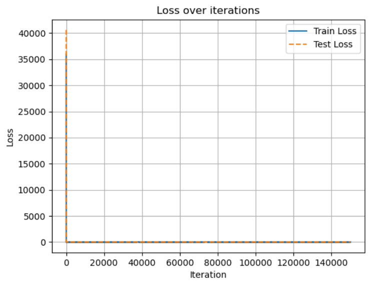
2.2 Different Normalization Method

To control variable, we all use MBGD method in this section to only compare the difference of normalization method.

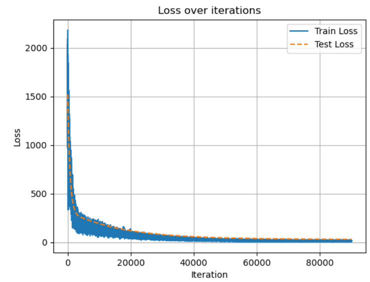
No Normalization Method:
n of iteration: 50000, loss rate: 0.0002
Learned weights: [9.0494355 1.02331261], Training loss: 10.862699339670602, Testing loss: 17.802074655015762
Time: 6.295790195465088s

Min-Max:
n of iteration: 30000, loss rate: 0.0005
Learned weights: [10.72168941 0.97626939], Training loss: 12.24969162788654, Testing loss: 31.357194692974275
Time: 4.983108997344971s

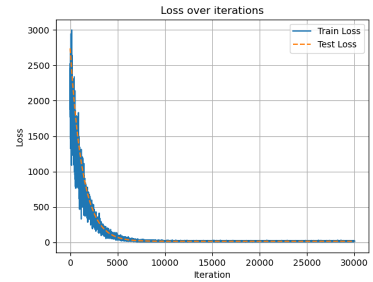
Mean:
n of iteration: 10000, loss rate: 0.0004
Learned weights: [10.84252905 1.00589178], Training loss: 13.687261666680957, Testing loss: 14.04630888716398
Time: 6.000371217727661s



(a) No Normalization

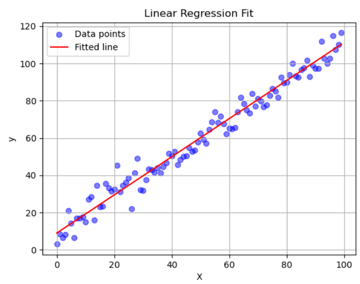


(b) Min-Max Normalization

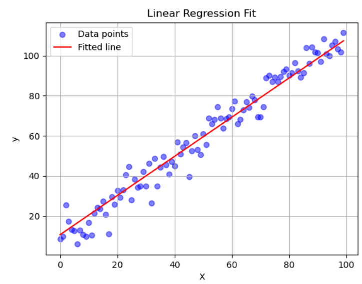


(c) Mean Normalization

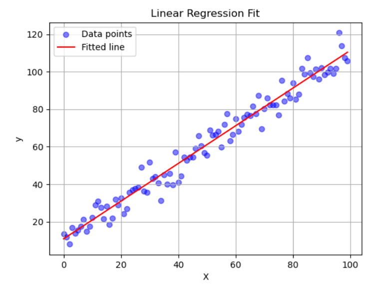
Fig. 3: The Trend of Loss Function of Different Gradient Method



(a) No Normalization



(b) Min-Max Normalization



(c) Mean Normalization

Fig. 4: The Fitting Results of Different Gradient Method

Normalization actually changes the step of each element of W. When normalization can balance the influence of each feature by correct choice of normalization, it can work well on inputs of different order of magnitude, and thus, it makes fitting better and quicker. However, it also lose efficacy when there is extreme outlier. Normalization also needs inverse normalization after fitting, which makes the algorithm more complex.

Min-Max Normalization can eliminate the effects of magnitude and order of magnitude. However, it may make the method overly dependent on two extreme values when changing the weights of the variables.

Mean Normalization can remove the quantisation. Dimensionlessness also eliminates the differences in the degree of variation of each variable, so that the converted variables are treated equally in the cluster analysis. However, in actual analyses, the importance of each variable in the analysis shouldn't be exactly the same. At the same time, mean normalization only work well with guassian input, otherwise it may makes the matter rather worse.