AI and Machine Learning, Homework4

Author: Ying Yiwen
Number: 12210159

Contents

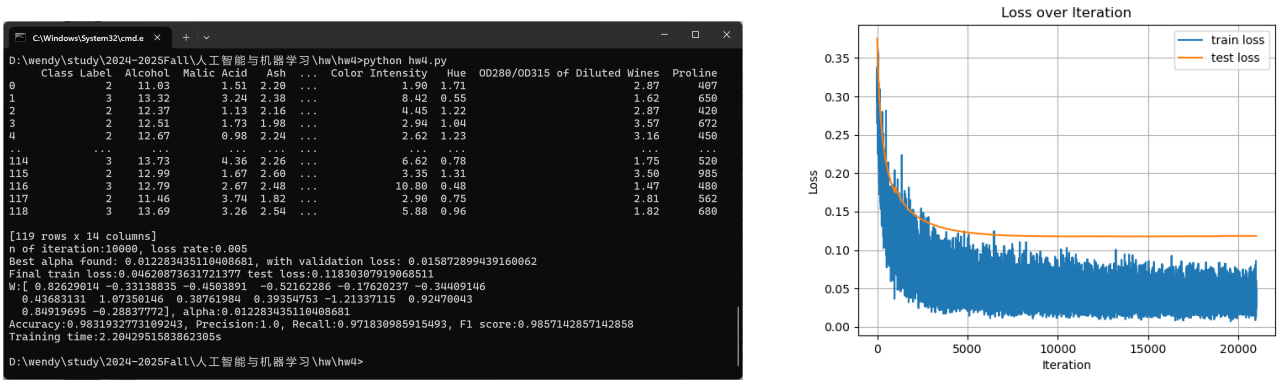
1 Analyze

Before training, we can use random function to prevent the fitting leans to a small set of samples. I also prepreocess the data to be label 0 and 1, which is easier for follow-up computing.
According to what we found in homework 2, we should use mean normalization for the features, so we can fit the model better.
Then, after adding the bias term, choosing the suitable parameters, with the correct loss function and update method, we can train the model.
The performance can be evaluated by accuracy, precision, recall, f1 score. And also, we should record the time of it.

1.1 Mini-Batch Update Method

In mini-batch update method, we can fit the model well.

The super parameter we use is: n of iteration:10000, loss rate:0.005, batch size:64
Using the k-fold cross-validation, we have the adjusted parameter. We use 20 random numbers from 0.001 to 1 to test the performance of it over 100 iteration. For the best performance, we take its α to train in the follow-up continuous training.
Best alpha found: 0.012283435110408681, with validation loss: 0.015872899439160062



(a) Running Result of MBGD

(b) MBGD Loss Function

Fig. 1: The Result of MBGD Fitting Method

The result is: Final train loss:0.04620873631721377 test loss:0.11830307919068511
ω: [0.82629014 -0.33138835 -0.4503891 -0.52162286 -0.17620237 -0.34409146 0.43683131 1.07350146 0.38761984 0.39354753 -1.21337115 0.92470043 0.84919695 -0.28837772], α: 0.012283435110408681

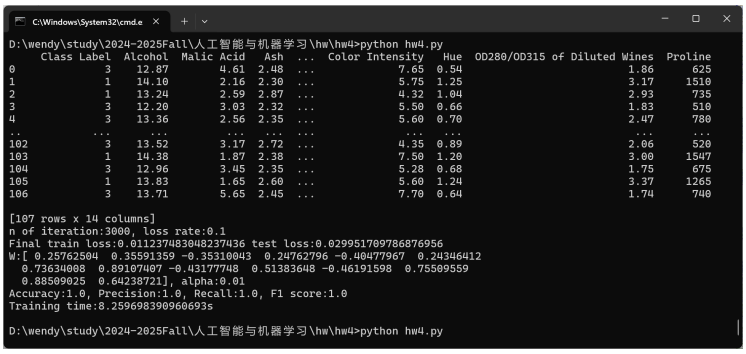
And we can evaluate the classification performance by:
Accuracy:0.9831932773109243, Precision:1.0, Recall:0.971830985915493, F1 score:0.9857142857142858
For the training process, time is also an important standard, training time:2.2042951583862305s

Between the first phase (attempting α) and the second phase (updating ω), there may be a position where the loss value suddenly increases, this is because, the training set is divided into a 4:1 training and validation set in the first phase, while the second phase is entirely devoted to training.

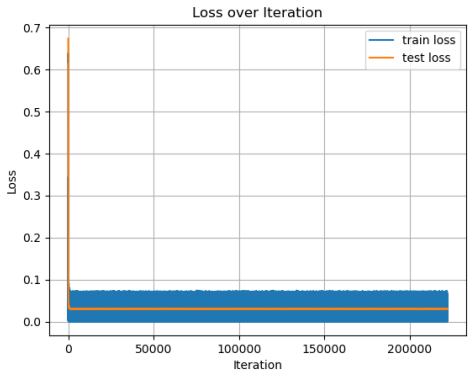
1.2 Stochastic Update Method

In mini-batch update method, we can fit the model well.

The super parameter we use is: n of iteration:3000, loss rate:0.1



(a) Running Result of SGD



(b) SGD Loss Function

Fig. 2: The Result of SGD Fitting Method

The result is: Final train loss:0.011237483048237436 test loss:0.029951709786876956
ω: [0.25762504 0.35591359 -0.35310043 0.24762796 -0.40477967 0.24346412 0.73634008 0.89107407 -0.43177748 0.51383648 -0.46191598 0.75509559 0.88509025 0.64238721], α: 0.01

And we can evaluate the classification performance by:
Accuracy:1.0, Precision:1.0, Recall:1.0, F1 score:1.0
For the training process, time is also an important standard, training time:8.259698390960693s

2 Difference between Perceptron and Logistic Regression

Perceptron is a basic linear classification model, mainly used for binary classification problems. The core idea is to classify by a linear combination of input features and apply a step function to determine the output category. The training process uses a perceptual machine learning algorithm that relies on stepwise adjustment of the weights with the aim of reducing the number of misclassified samples in the training set. Perceptual machines use misclassification loss, which only considers whether the classification is correct or not, and are therefore effective when dealing with linearly divisible data, but for non-linearly divisible data, the model may not converge. In addition, perceptual machines cannot provide a measure of classification confidence, limiting their application to complex tasks.

The logistic regression model, on the other hand, is a more flexible linear classifier whose output is a linear combination of inputs converted to a probability value between 0 and 1 via a logistic function (sigmoid). This allows logistic regression to not only make binary classification decisions, but also to provide a level of confidence that each sample belongs to a particular category. Logistic regression is trained using a log-loss function that takes into account the probability of categorization, which allows for better handling of unbalanced data and probabilistic predictions. The training process usually uses maximum likelihood estimation to optimize the parameters, and common optimization methods include gradient descent or simulated Newton's method. Such models can effectively handle both linearly separable and non-separable cases, and their output probabilities can be used for subsequent decision making.

3 Important Code Implementation

3.1 Read in Data and Split Data

We need to read in the dataset, randomly remove one class of wine samples from the dataset, retaining the other two classes to generate a new dataset.

```python
def read_dataset():
  # read dataset
  column_names = [
  'Class Label', 'Alcohol', 'Malic Acid', 'Ash', 'Alcalinity of Ash',
  'Magnesium', 'Total Phenols', 'Flavanoids', 'Nonflavanoid Phenols',
  'Proanthocyanins', 'Color Intensity', 'Hue', 'OD280/OD315 of Diluted Wines',
  'Proline'
  ]
  data = pd.read_csv('wine.data', header=None, names=column_names)
  # randomly remove a class
  unique_classes = data['Class Label'].unique()
  class_to_remove = np.random.choice(unique_classes)
  data_filtered = data[data['Class Label'] != class_to_remove]
  # shuffle the data
  data_filtered = data_filtered.sample(frac=1, random_state=42).reset_index(drop=
      True)
  print(data_filtered)
  return data_filtered
```

Then we need to split the features and labels.

```python
def convert_labels(y):
  # convert labels to 0 and 1
  converted_labels = []
  y = np.array(y)
  for label in y:
  if label == y[0]:
  converted_labels.append(0)
  else:
  converted_labels.append(1)
  return np.array(converted_labels)

def split_data(data_filtered):
  # split data into features and labels
  data_filtered = data_filtered.sample(frac=1)
  # separate features and labels
  x = data_filtered.iloc[:, 1:]
  y = data_filtered.iloc[:, 0]
  y = convert_labels(y)
  return x, y
```

Also, it's important to split the data to train set and test set by 7:3.

```python
def split_test_and_train(x, y):
  # split data into train and test
  x_train = x.iloc[:int(len(x)*0.7), :]
  x_test = x.iloc[int(len(x)*0.7):, :]
  y_train = y[:int(len(y)*0.7)]
  y_test = y[int(len(y)*0.7):]
  return x_train, x_test, y_train, y_test
```

3.2 Comput Loss and Gradient

We need to predict the label of the samples.

```
def _linear_tf(self, X):
  return X @ self.W  # Linear transformation of inputs

def _sigmoid(self, z):
  return 1 / (1 + np.exp(-z))  # Sigmoid function for probability estimation

def predict_probability(self, X):
  # Linear transformation and sigmoid activation function
  z = self._linear_tf(X)
  return self._sigmoid(z)
```

Compute the loss. The equation is $\ell_{\log}(w) = -\sum_{i=1}^{N} t^{(i)} \log y(x^{(i)}, w) - \sum_{i=1}^{N} (1 - t^{(i)}) \left(1 - \log y(x^{(i)}, w)\right)$.

```
def _loss(self, y, y_pred):
  epsilon = 1e-5  # Small constant to avoid log(0)
  loss = -np.mean(y * np.log(y_pred + epsilon) + (1 - y) * np.log(1 - y_pred +
      epsilon))
  return loss
```

Compute the gradient. The equation is $\nabla l(w) = -X^T(t - y)$. At the same time, we can add bias term in this formula. So the formula becomes, $\nabla \tilde{l}(w) = \nabla l(w) + \alpha w$. When $\alpha = 0$, there is no regularize term.

```
def _gradient(self, X, y, y_pred):
  # add regularization term
  regularization_term = self.alpha * self.W
  return -(y - y_pred) @ X / y.size + regularization_term  # Gradient calculation
```

3.3 Update Method

We have mini-batch gradient descent, stochasitc gradient descent, and batch gradient descent. In this section, we only show the code of mini-batch gradient descent to show the principle of update method.
We need to compute and record train loss and test loss for visualization after the training; compute gradient to update the parameters.
The first stage is to choose the best α, and the second stage is to fit ω. So in the first stage, the train set is rather split into train set and validation set. And in the second stage, the validation set is no longer necessary, so we take all samples into train set.
For mini-batch update method, we need another recursive to use each mini batch to train.

```
def mini_batch_update(self, X, y, X_test, y_test, batch_size=64, k=5, n_samples
    =20, n_iter_alpha=100):
  # find the best alpha
  best_alpha = None
  best_val_loss = float('inf')
  # Random search for alpha
  for _ in range(n_samples):
    current_alpha = np.random.uniform(0.001, 1.0)  # Randomly select alpha
    self.alpha = current_alpha  # Set current alpha
    loss1 = []
    test_loss1 = []
    # Train the model for a specified number of iterations
    for iter in range(n_iter_alpha):
      # shuffle the data
      indices = np.random.permutation(X.shape[0])
      for i in range(0, X.shape[0], batch_size):
```

```
        # get the batch
        batch_indices = indices[i:min(i + batch_size, X.shape[0])]
        X_batch, y_batch = X[batch_indices], y[batch_indices]
        # split the data into k folds
        for train_indices, val_indices in self.k_fold_split(X_batch.shape[0], k):
          X_train, X_val = X_batch[train_indices], X_batch[val_indices]
          y_train, y_val = y_batch[train_indices], y_batch[val_indices]
          # Calculate training loss
          y_pred = self.predict_probability(X_train)
          loss = self._loss(y_train, y_pred)
          loss1.append(loss)  # Store the loss value
          # Calculate test loss
          y_pred_test = self.predict_probability(X_test)
          loss_test = self._loss(y_test, y_pred_test)
          test_loss1.append(loss_test)  # Store the test loss value
          # Update weights
          grad = self._gradient(X_train, y_train, y_pred)
          self.W = self.W - self.lr * grad  # Update the weights
      # Calculate validation loss
      y_pred_val = self.predict_probability(X_val)
      val_loss = self._loss(y_val, y_pred_val)
      # Check if the current alpha has the best validation loss
      if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_alpha = current_alpha
        self.loss = loss1  # Store the loss value
        self.test_loss = test_loss1  # Store the test loss value
        final_W = self.W # Store the weights
  # Set the best alpha for further training
  self.alpha = best_alpha
  self.W = final_W
  print(f"Best alpha found: {best_alpha}, with validation loss: {best_val_loss}")

  # Continue training with the best alpha
  for iter in range(self.n_iter):
    # shuffle the data
    indices = np.random.permutation(X.shape[0])
    for i in range(0, X.shape[0], batch_size):
      # get the batch
      batch_indices = indices[i:min(i + batch_size, X.shape[0])]
      X_train, y_train = X[batch_indices], y[batch_indices]
      # train loss
      y_pred = self.predict_probability(X_train)
      loss = self._loss(y_train, y_pred)
      self.loss.append(loss)  # Store the loss value
      # test loss
      y_pred_test = self.predict_probability(X_test)
      loss_test = self._loss(y_test, y_pred_test)
      self.test_loss.append(loss_test)  # Store the test loss value
      # update weights
      grad = self._gradient(X_train, y_train, y_pred)
      self.W = self.W - self.lr * grad  # Update the weights
```

To better fit the model, we need another bias term in the gradient function, like $\nabla\tilde{l}(w) = \nabla l(w) + \alpha w$. It's ok to

appoint a specific number for α, but it would be better if we have validation set to do k-fold cross validation and update α each time.

```
def k_fold_split(self, length, k):
  indices = np.arange(length)
  np.random.shuffle(indices)  # shuffle the indices
  folds = np.array_split(indices, k)  # split the data into k folds
  # yield the train and validation indices
  for i in range(k):
    train_indices = np.concatenate(folds[:i] + folds[i+1:])  # train set
    val_indices = folds[i]  # validation set
    yield train_indices, val_indices
```

3.4 Evaluating

After training, we need to evaluate the accuracy, precision, recall, F1 score.

```
def evaluate(self, X, y):
  # predict the label of X
  y_pred = self.predict(X)
  y_pred = np.where(y_pred >= 0.5, 1, 0)
  # evaluate model accuracy, precision, recall, f1 score
  TP = sum((y_pred[i] == 1) and (y[i] == 1) for i in range(len(y)))
  FP = sum((y_pred[i] == 1) and (y[i] == 0) for i in range(len(y)))
  TN = sum((y_pred[i] == 0) and (y[i] == 0) for i in range(len(y)))
  FN = sum((y_pred[i] == 0) and (y[i] == 1) for i in range(len(y)))
  accuracy = (TP + TN) / (TP + TN + FP + FN)
  precision = TP / (TP + FP)
  recall = TP / (TP + FN)
  f1 = 2 * precision * recall / (precision + recall)
  print(f"Accuracy:{accuracy}, Precision:{precision}, Recall:{recall}, F1:{f1}")
```

Also, we can visualize the trend of loss function.

```
def plot_loss(self):
  # plot loss
  plt.plot(self.loss, alpha=1, label='train loss')
  plt.plot(self.test_loss, alpha=1, label='test loss')
  # add title and labels
  plt.legend(['train loss', 'test loss'])
  plt.title('Loss over Iteration')
  plt.xlabel('Iteration')
  plt.ylabel('Loss')
  plt.grid()
  plt.savefig('loss.png')
  plt.show()
  print(f"Final train loss:{self.loss[-1]} test loss:{self.test_loss[-1]}")
  print(f"W:{self.W}, alpha:{self.alpha}")
```

3.5 Main Function

So the whole process, using the above function is:

```
# read dataset
```

```
data = read_dataset()
x, y = split_data(data)
X_train, X_test, y_train, y_test = split_test_and_train(x, y)

# initialize the logistic regression model
start_time = time.time()
_, n_features = X_train.shape
n_iter, lr = 10000, 0.005
print(f"n of iteration:{n_iter}, loss rate:{lr}")
model = LogisticRegression(n_features=n_features, n_iter=n_iter, lr=lr)

# Train the model
model.train(X_train, y_train, X_test, y_test, method='mini-batch')
end_time = time.time()

# plot loss and evaluate the model
model.plot_loss()
model.evaluate(x, y)
print(f"Training time:{end_time - start_time}s")
```