CS 412 Homework 3 - Logistic Regression from Scratch

1.Preprocessing

Firstly, the Titanic dataset is loaded from the CSV file named "titanicdata.csv" using the pandas library. The data is then split into three subsets: 60% for training, 20% for validation, and 20% for testing. This splitting is achieved by first splitting the data into 60% training and 40% temporary (validation + test) sets, and then further splitting the temporary set evenly into validation and test sets. Next, the features (X) and the target (y) are separated. The features include "Pclass", "Sex", and "Age" columns, while the target is the "Survived" column, indicating whether a passenger survived or not. After that, the features are scaled using MinMaxScaler, which scales the features to a specified range, typically between 0 and 1. The scaler is fitted to the training data to learn the scaling parameters and then applied to both the validation and test sets using the learned parameters. This ensures that the same scaling parameters are used consistently across all datasets. As a result, the training, validation, and test sets are prepared with the features scaled appropriately, making them ready for machine learning algorithms.

```
#load CSV
df = pd.read_csv("titanicdata.csv")

# Split the data
train_data, temp_data = train_test_split(df, test_size=0.4, random_state=42)
validation_data, test_data = train_test_split(temp_data, test_size=0.5, random_state=42)

#Split features and target
X_train = train_data[["Pclass", "Sex", "Age"]]
y_train = train_data["Survived"]

X_val = validation_data[["Pclass", "Sex", "Age"]]
y_val = validation_data["Survived"]

X_test = test_data[["Pclass", "Sex", "Age"]]
y_test = test_data["Survived"]

#Scaling
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
```

2. Implementing Logistic Regression Model from Scratch

This is the formula of sigmoid function that we use as a parameter for comput loss function.

$$h_\theta(x) = g(\theta^T x)$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Here is the implementation of sigmoid function Python.

```
def sigmoid(z):

    σ = 1/(1+np.exp(-z))

    return σ
```

Similarly, this is the formula of cost(loss) function in logistic regresion.

Cost function in logistic regression is:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^i log(h_\theta(x^i)) + (1-y^i) log(1-h_\theta(x^i))]$$

Here is my implementation of loss function in Python.

```
#Takes X y and w as numpy arrays.
def compute_cost_logistic(X, y, w, b):

    m = X.shape[0]
    cost = 0.0
    for i in range(m):
        z_i = np.dot(X[i],w) + b
        f_wb_i = sigmoid(z_i)
        cost +=  -y[i]*np.log(f_wb_i) - (1-y[i])*np.log(1-f_wb_i)

    cost = cost / m
    return cost
```

Now, we must calculate gradient and use gradient descent in order to find best w and b for minimizing our cost(loss) function. These are formulas for gradient.

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)}) \cdot x_j^{(i)} \text{ for j=0}$$

$$\frac{\partial J(\theta)}{\partial \theta_0} = (\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)}) \cdot x_j^{(i)}) + \frac{\lambda}{m}\theta_j \text{ for j}\geq 1$$

Here is my implementation for gradient descen in Python. I also store loss values for training data and validation data in order to plot them.

```python
def compute_gradient_logistic(X, y, w, b):

    m,n = X.shape
    dj_dw = np.zeros((n,))
    dj_db = 0.

    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i],w) + b)
        err_i  = f_wb_i  - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err_i * X[i,j]
        dj_db = dj_db + err_i
    dj_dw = dj_dw/m
    dj_db = dj_db/m

    return dj_db, dj_dw
```

```python
def gradient_descent(X, y, X_val, y_val, w_in, b_in, alpha, num_iters):

    J_history = []
    validation_loss_history = []
    w = copy.deepcopy(w_in)
    b = b_in

    for i in range(num_iters):
        # Gradient
        dj_db, dj_dw = compute_gradient_logistic(X, y, w, b)

        # Update Parameters
        w = w - alpha * dj_dw
        b = b - alpha * dj_db

        # Training loss
        training_loss = compute_cost_logistic(X, y, w, b)
        J_history.append(training_loss)

        # Validation loss
        validation_loss = compute_cost_logistic(X_val, y_val, w, b)
        validation_loss_history.append(validation_loss)

    return w, b, J_history, validation_loss_history
```

3.Calculate training and validation losses across 100 iteration

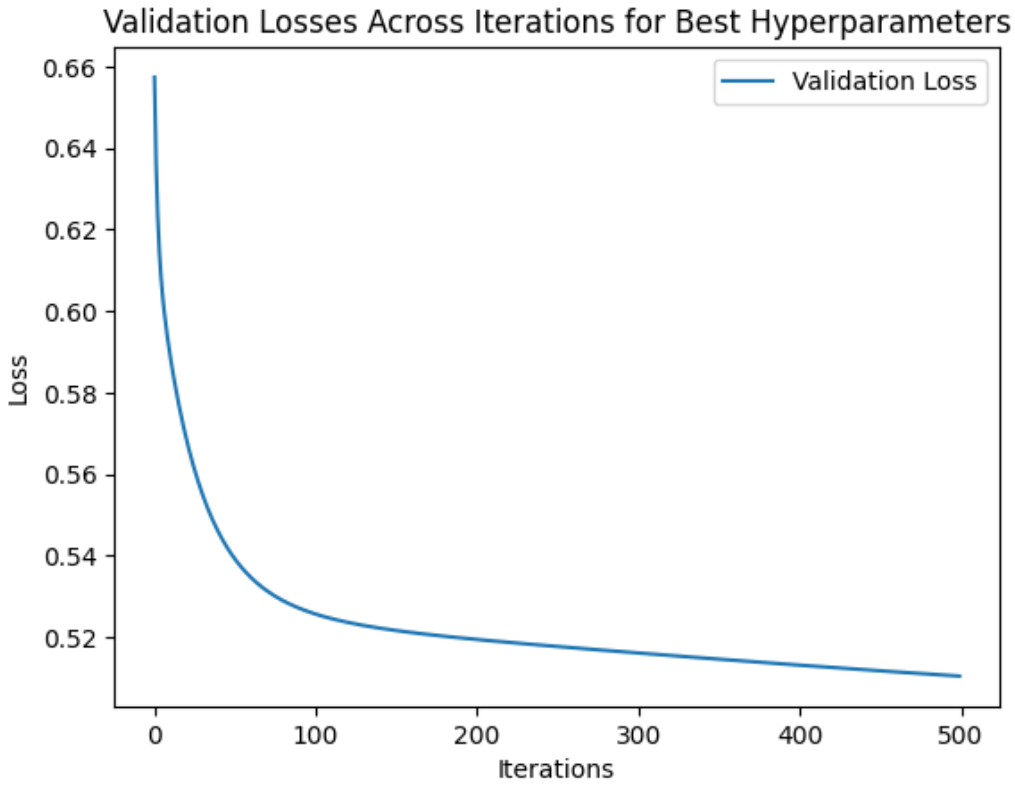
We noticed that the loss calculated during training gradually goes down as the model learns from the training data. This loss starts at around 0.685 and decreases to about 0.570 over 100 rounds of training. However, when we check how well the model performs on new, unseen data (validation data), we see a similar trend in loss reduction, but it stays a bit higher, starting at 0.685 and ending at 0.570. This difference between the training and validation losses suggests that while the model gets better at dealing with the training data, it struggles a bit with new data.

Here is my implementation:

```
1
2 alpha = 0.1
3
4
5 w_initial = np.zeros(X_train_scaled.shape[1])
6 b_initial = 0
7
8
9 w_trained, b_trained, training_loss_history, validation_loss_history = gradient_descent(X_train_scaled, y_train.value
10
11
12 plt.plot(range(len(training_loss_history)), training_loss_history, label='Training Loss')
13 plt.plot(range(len(validation_loss_history)), validation_loss_history, label='Validation Loss')
14 plt.xlabel('Iterations')
15 plt.ylabel('Loss')
16 plt.title('Training and Validation Losses Across 100 Iterations')
17 plt.legend()
18 plt.show()
19
```

4.Varying number of steps and iterations

Later on, we experimented with various step sizes and iteration counts. We changed the number of iterations to 50, 100, 200, and 500, and the step sizes to 0.01, 0.05, 0.1, and 0.5. You can see result of loss function across diffent iterations counts and step sizes in Figure 1. In the end, we discovered that a validation loss of 0.5105025803845864 was obtained by setting the step size to 0.5 and the number of iterations to 500. This suggests that the model successfully assimilated the unseen validation data as well as the training set. The number of iterations determines how long the learning process takes, and the step size controls how quickly the model learns. Our selection of values achieved a good balance between allowing the model to grow and not becoming overly entangled in the details of the training set. This demonstrates how these settings can significantly impact the model's performance.



Şekil 1

Here is my implementation about varying step size and iterations.

```
step_sizes = [0.01, 0.05, 0.1, 0.5]
num_iterations = [50, 100, 200, 500]

best_loss = float('inf')
best_params = {}

for step_size in step_sizes:
    for num_iter in num_iterations:

        w_trained, b_trained, training_loss_history, validation_loss_history = gradient_descent(X_train_scaled, y

        validation_loss = validation_loss_history[-1]

        if validation_loss < best_loss:
            best_loss = validation_loss
            best_params['step_size'] = step_size
            best_params['num_iterations'] = num_iter
            best_params['w_trained'] = w_trained
            best_params['b_trained'] = b_trained
            best_params['training_loss_history'] = training_loss_history
            best_params['validation_loss_history'] = validation_loss_history



plt.plot(range(len(best_params['validation_loss_history'])), best_params['validation_loss_history'], label='Valid
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Validation Losses Across Iterations for Best Hyperparameters')
plt.legend()
plt.show()

print("Best hyperparameters:")
print("Step size:", best_params['step_size'])
print("Number of iterations:", best_params['num_iterations'])
print("Validation loss:", best_loss)

```

5.Combining validation and training data

To create a larger dataset for training the final model with the selected hyperparameters, we combined the training and validation data. Next, we used the selected hyperparameters—a step size of 0.5 and 500 iterations—to retrain the finished model. We assessed the final model's performance using the test data after it had been trained. Using the logistic regression cost function, the test loss was determined to be 0.4152676420264039. This test loss shows how well the model generalizes to new instances and reflects its performance on unseen data. This result implies that the final model performs fairly well on test data that has not yet been seen. A lower test loss indicates better performance.

Implementation:

```
# Combine
X_train_combined = np.vstack((X_train_scaled, X_val_scaled))
y_train_combined = np.concatenate((y_train.values, y_val.values))


# Retrain the final model with the chosen hyperparameters
final_w, final_b, _, _ = gradient_descent(X_train_combined, y_train_combined, X_test_scaled, y_test.values
# Evaluate the final model on test data
test_loss = compute_cost_logistic(X_test_scaled, y_test.values, final_w, final_b)
print("Test Loss:", test_loss)

```

6.Evaluate the accuracy

It's amazing how accurate our trained model was with the test data. Based on the test data, the estimated accuracy is roughly 0.8101. This shows that the model predicts the test data with an accuracy of about 81%. These outcomes show that the model can successfully generalize the learned patterns and performs well on the test data. All things considered, this indicates that our model performs well and is competitive when measured against earlier findings.

Implementation:

```

y_pred = []
for i in range(len(X_test_scaled)):
    z_i = np.dot(X_test_scaled[i], final_w) + final_b
    y_pred_i = sigmoid(z_i)
    y_pred.append(1 if y_pred_i >= 0.5 else 0)


accuracy = np.mean(y_pred == y_test.values)
print("Accuracy on Test Data:", accuracy)
```

By contrasting our model's performance with the one trained with Scikit-learn, we can gain important understanding of the advantages and disadvantages of each.

Using test data, the Scikit-learn logistic regression model produced the following performance metrics:

Precision: 0.8156
Accuracy: 0.8308
Remember: 0.7105 F1 Points: 0.7659
Conversely, the accuracy of our implemented logistic regression model on the test data was about 0.8101.

We find that, on average, the Scikit-learn model performs better than our implemented model on all metrics. More specifically, the F1 score, recall, accuracy, and precision are all higher in the Scikit-learn model. This suggests that the Scikit-learn model performs better in accurately classifying instances that are positive or negative and in striking a balance between precision and recall.

It's crucial to remember that, despite being implemented, our model still functions fairly well and has a similar accuracy score. This shows that our model reasonably well generalizes to previously undiscovered instances and captures important patterns in the data. Furthermore, the two models' reasonably close performance suggests that our implementation is effective, even though it performs marginally worse than the optimized Scikit-learn implementation.