Table of Contents

Artificial Intelligence
Assignment 2
Report

- ## Dataset

1. Introduction

The Titanic dataset is widely used in the field of data science and machine learning. It contains information about passengers aboard the RMS Titanic, including details such as their demographics, ticket class, cabin, and most notably, whether they survived the disaster. This dataset serves as a valuable resource for exploratory data analysis, predictive modeling, and drawing insights related to the tragic sinking of the Titanic.

2. Dataset Overview:

The dataset comprises various features such as:

- PassengerID: Unique identifier for each passenger
- Survived: Indicates whether the passenger survived (0 = No, 1 = Yes)
- Pclass: Ticket class (1st, 2nd, or 3rd)
- Name: Passenger's name
- Sex: Gender of the passenger
- Age: Age of the passenger
- SibSp: Number of siblings/spouses aboard
- Parch: Number of parents/children aboard
- Ticket: Ticket number
- Fare: Passenger fare
- Cabin: Cabin number
- Embarked: Port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

3. Objective of Analysis for KNN:

The primary objective of applying the K-Nearest Neighbours algorithm to the Titanic dataset is to build a predictive model that can classify passengers' survival outcomes based on their attributes. By using the KNN algorithm, we aim to explore patterns in the dataset, identify similar instances among passengers, and predict survival outcomes based on the similarity of a given passenger's features to those of their nearest neighbours.

4. Objective of Analysis for Neural Network:

The primary objective of applying Neural Network models to the Titanic dataset is to create a predictive model that can effectively learn complex relationships and patterns within the data to predict passenger survival.

5. Inputs Feature and Ground Truth

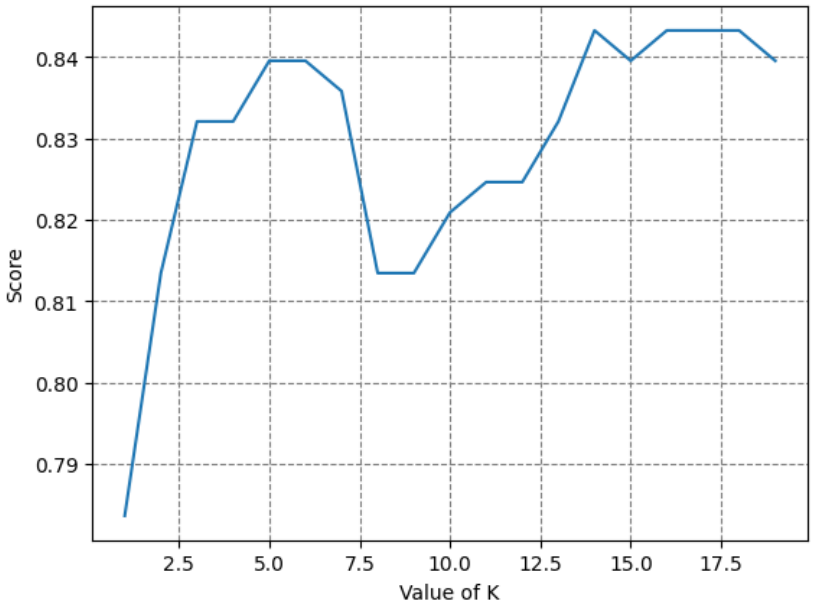
Code:

```
inputs = np.array(titn1.filter(['Pclass','Sex','Age','Fare','Embarked','Family'], axis=1))

target = np.array(titn1['Survived'])
```

• MODEL 1: K-Nearest Neighbors

Code:

```
k_range = range(1,20)

score = []

for k in k_range:

    knn = KNeighborsClassifier(n_neighbors=k, metric='minkowski')

    knn.fit(X_train, y_train)

    y_pred = knn.predict(X_test)

    score.append(metrics.accuracy_score(y_pred,y_test))

score
```



1. Conclusion:

Optimal Range of K: The graph displays a peak in accuracy within a specific range of K values. In this case, the highest accuracy is observed in the range from approximately 0.82 to 0.85 for K values between 11 and 18. This indicates that the model achieves its optimal performance within this range.

Upon analysis, the highest accuracy achieved by the KNN model was 84%. This optimal accuracy was obtained when the number of neighbors (K) was set to 13. This finding suggests that considering 13 nearest neighbors provides the most accurate predictions for survival outcomes among passengers in this dataset.

• MODEL 2: Neural Network

1. Weight Initialization

```
def zero_initialize_weights(self):
    self.weights['W1'] = np.zeros((X.shape[1], 4))
    self.weights['W2'] = np.zeros((4, 1))


 def random_initialize_weights(self):
    self.weights['W1'] = np.random.randn(X.shape[1], 4) * 0.1
    self.weights['W2'] = np.random.randn(4, 1) * 0.1
```

2. Forward Propagation

```
def forward_propagation(self, X):
    Z1 = np.dot(X, self.weights['W1']) + self.weights['b1']
    A1 = self.sigmoid(Z1)
    Z2 = np.dot(A1, self.weights['W2']) + self.weights['b2']
    A2 = self.sigmoid(Z2)
    return {'Z1': Z1, 'A1': A1, 'Z2': Z2, 'A2': A2}
```

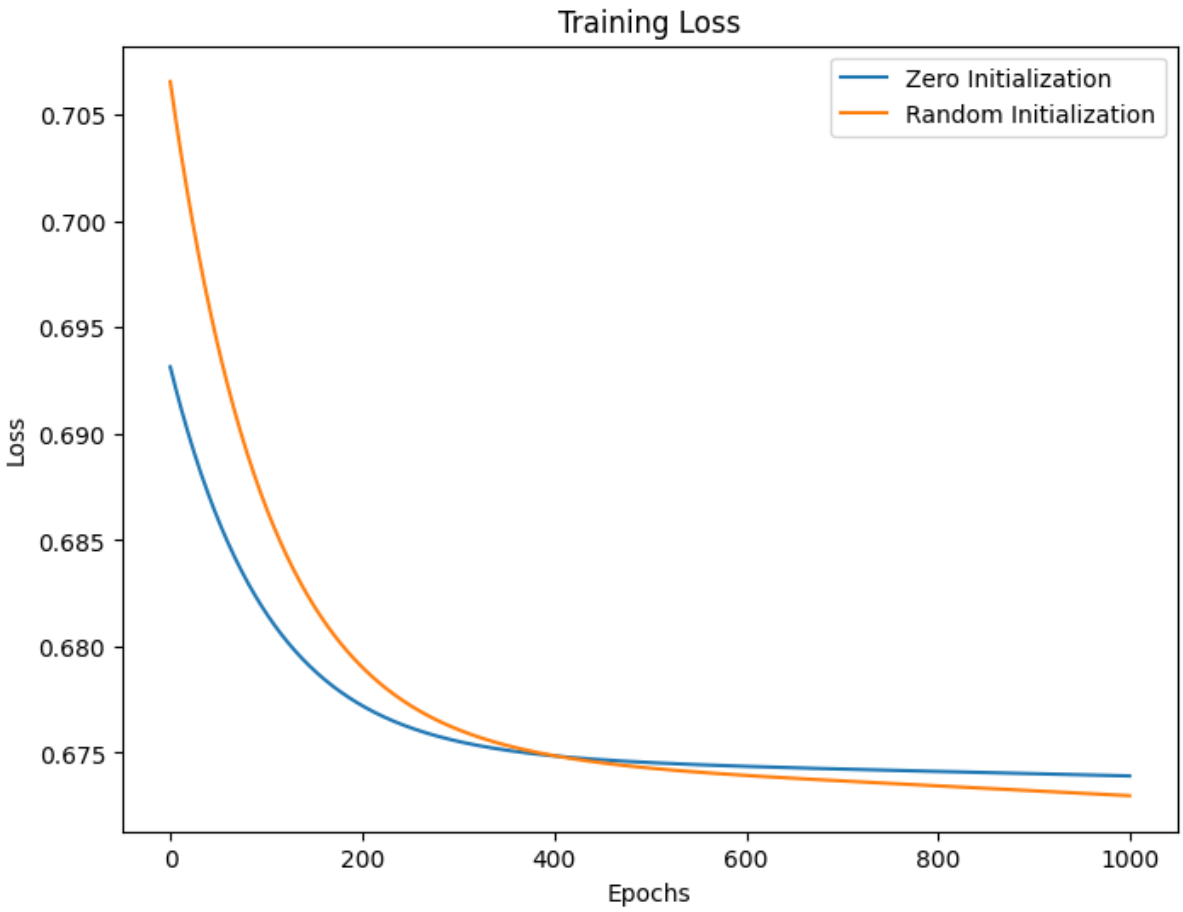
3. Backward Propagation

```
m = X.shape[0]
    dZ2 = cache['A2'] - y
    dW2 = 1 / m * np.dot(cache['A1'].T, dZ2)
    db2 = 1 / m * np.sum(dZ2, axis=0, keepdims=True)
```

```
dZ1 = np.dot(dZ2, self.weights['W2'].T) * (cache['A1'] * (1 - cache['A1']))

dW1 = 1 / m * np.dot(X.T, dZ1)

db1 = 1 / m * np.sum(dZ1, axis=0, keepdims=True)


grads = {'dW1': dW1, 'db1': db1, 'dW2': dW2, 'db2': db2}

return grads
```

4. Training with Zero & Random Weight Initialization with learning_rate = 0.01

```
# Training with zero initialization

model_zero = NeuralNetwork(num_features, 4, 1)

losses_zero = model_zero.train(X, y.reshape(-1, 1), initialization_method='zero', learning_rate=0.01, epochs=1000)

# Training with random initialization

model_random = NeuralNetwork(num_features, 4, 1)

losses_random = model_random.train(X, y.reshape(-1, 1), initialization_method='random', learning_rate=0.01, epochs=1000)
```



I. Experiment Overview:

Two neural network models were trained using different weight initialization methods—zero initialization and random initialization. The objective was to evaluate their performance during training by observing the loss function across epochs.

II. Observations:

- The plot indicates that the model initialized with random weights maintained lower loss values compared to the model initialized with zeros at the end of 1000 epochs.
- Despite both models showing a decreasing trend in loss, the random initialization model achieved a marginally lower final loss.

III. Conclusion:

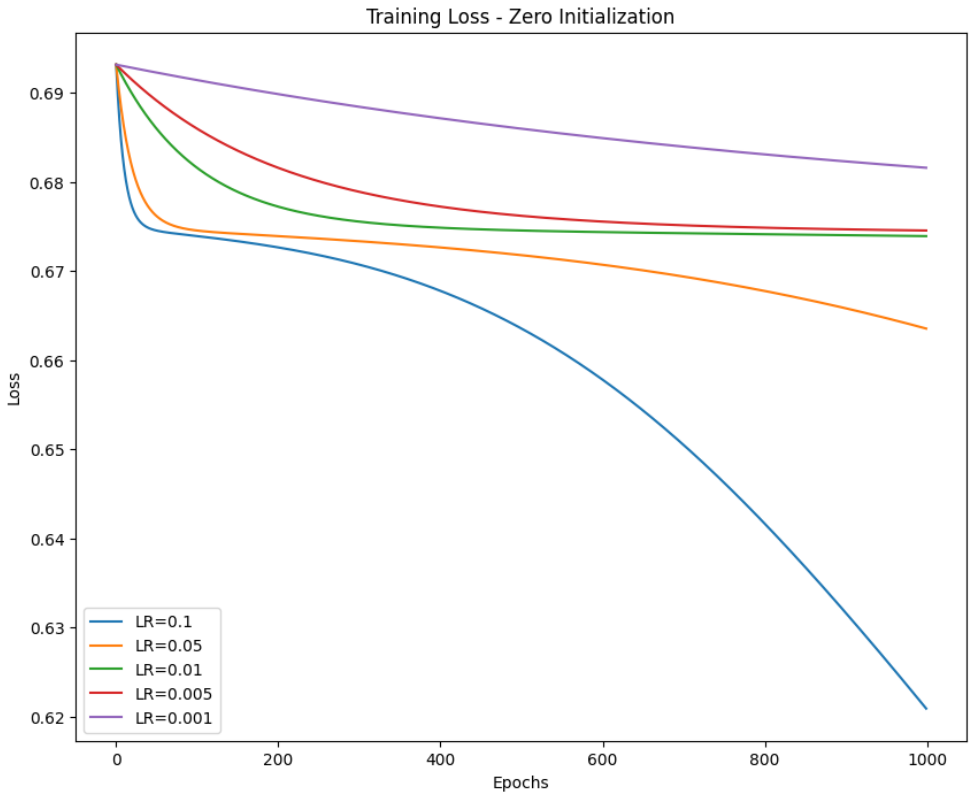
The experiment showcased that the model initialized with random weights maintained lower loss values compared to the model initialized with zeros at the end of the training process. This indicates that, in this specific scenario and observation period, random initialization had a slight advantage in terms of loss reduction.

5. Training with Zero & Random Weight Initialization with learning_rate [0.1,0.05,0.01,0.005,0.001]

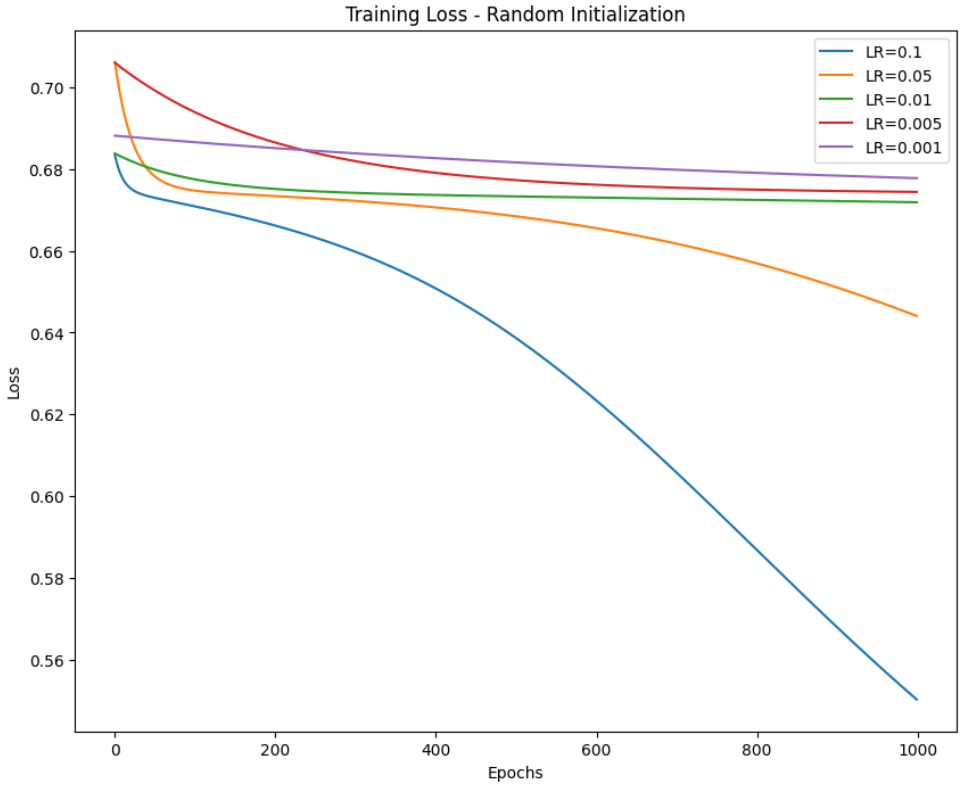
Code

```
def train_with_different_learning_rates(X, y, initialization_method, learning_rates, num_hidden_units=4, epochs=1000):
    all_losses = []

    for lr in learning_rates:
        model = NeuralNetwork(X.shape[1], num_hidden_units, 1)
        losses = model.train(X, y.reshape(-1, 1), initialization_method=initialization_method, learning_rate=lr, epochs=epochs)
        all_losses.append(losses)

    return all_losses
```

I. Training Loss- Zero Initialization [0.1,0.05,0.01,0.005,0.001]



II. Training Loss- Random Initialization [0.1,0.05,0.01,0.005,0.001]



6. Avoiding overfitting using early stopping.

Code

```
def train_with_early_stopping(self, X_train, y_train, X_val, y_val, learning_rate, epochs):
    self.random_initialize_weights()

    train_losses = []
    val_losses = []
    best_loss = float('inf')
    best_epoch = 0
    early_stopping_patience = 20
    no_improvement_count = 0

    for epoch in range(epochs):
        train_cache = self.forward_propagation(X_train)
        train_loss = self.compute_loss(y_train, train_cache['A2'])
        train_losses.append(train_loss)
        train_grads = self.backward_propagation(X_train, y_train, train_cache)
        self.update_parameters(train_grads, learning_rate)

        val_cache = self.forward_propagation(X_val)
        val_loss = self.compute_loss(y_val, val_cache['A2'])
        val_losses.append(val_loss)

        if val_loss < best_loss:
            best_loss = val_loss
            best_epoch = epoch
            no_improvement_count = 0
        else:
            no_improvement_count += 1

        if no_improvement_count >= early_stopping_patience:
```

```
            print(f"Best Weight {train_grads}. Best Learning Rate {learning_rate}.")
            print(f"Early stopping at epoch {epoch}. Best epoch: {best_epoch}.")
            break

        if epoch % 100 == 0:
            print(f"Epoch {epoch}: Training Loss {train_loss}, Validation Loss {val_loss}")

    return train_losses + val_losses
```

I. Random Initialization – Best Model with Learning Rate = 0.1

Code

```
combined_losses = nn.train_with_early_stopping(X_train, y_train.reshape(-1, 1), X_val, y_val.reshape(-1, 1), learning_rate=0.1, epochs=10000)
```

Output

```
Epoch 4600: Training Loss 0.4511309169519024, Validation Loss 0.4302911
0764673
Early stopping at epoch 4695. Best epoch: 4685.
```

II. Experiment Overview:

During the training of a Neural Network model for the Titanic dataset, early stopping was employed to prevent overfitting. The objective was to monitor the model's performance on a validation set and halt the training process when further iterations led to deteriorating performance on the validation set.

III. Training Process:

- The training process involved optimizing the Neural Network model's weights with a learning rate of 0.1.
- Initially, both the training and validation losses decreased as the model learned patterns from the data.
- However, while the training loss continued to decrease, the validation loss eventually started increasing, indicating the model was overfitting the training data.

IV. Early Stopping Implementation:

- Early stopping was employed to prevent overfitting by monitoring the validation loss during training.
- At each epoch, the validation loss was calculated using the validation dataset, separate from the training data.

- Training was stopped when the validation loss didn't improve after a predefined number of epochs (patience) despite the training loss continuing to decrease.

V. Observations:

- The training loss consistently decreased over epochs, indicating the model's ability to learn from the training data.
- Meanwhile, the validation loss initially decreased but eventually started to increase after a certain point **(epoch ~4695).**
- Early stopping was triggered when the validation loss did not improve for a defined number of epochs (patience) after its last decrease.

7. Models Validation

Code

```
test_cache = nn.forward_propagation(X_test)

test_predictions = (test_cache['A2'] > 0.5).astype(int)

test_accuracy = np.mean(test_predictions == y_test.reshape(-1, 1))

print(f"Testing Accuracy: {test_accuracy * 100:.2f}%")


train_cache = nn.forward_propagation(X_train)

train_predictions = (train_cache['A2'] > 0.5).astype(int)

train_accuracy = np.mean(train_predictions == y_train.reshape(-1, 1))

print(f"Training Accuracy: {train_accuracy * 100:.2f}%")
```

Output

```
Testing Accuracy: 80.45%
Training Accuracy: 80.52%
```

I. Experiment Overview:

A Neural Network model was trained using early stopping on the Titanic dataset to prevent overfitting. The objective was to select optimal hyperparameters based on training performance, and subsequently evaluate the model's accuracy on both the training and test datasets.

II. Training Process:

- *Early Stopping Implementation*: The training process included early stopping, which monitored the model's performance on a validation set to prevent overfitting. The training halted when the validation loss failed to improve after a specific number of epochs.
- *Hyperparameter Selection*: Initially, the model was trained with different learning rates and epochs. Among these variations, a learning rate of 0.1 was selected based on its lower validation loss compared to other learning rates (0.01, 0.1, 0.5, 1.0) and epochs (10000) chosen for early stopping.

III. Model Evaluation:

- *Training Accuracy*: The model achieved a training accuracy of 80.52%.
- *Testing Accuracy*: The accuracy on the test dataset was 80.45%.

IV. Selection Criteria:

- *Weight Initialization*: Initially, both zero and random weight initializations were considered. However, the model trained with random initialization demonstrated lower loss during training, leading to its selection over the model trained with zero initialization.
- *Learning Rate Selection*: Among the various learning rates tested (0.01, 0.1, 0.5, 1.0), the learning rate of 0.1 exhibited lower loss during training and was chosen for the final model due to its superior performance in terms of convergence and validation loss reduction.

V. Conclusion:

The Neural Network model trained with early stopping showcased a balanced performance, achieving an accuracy of approximately 80% on both the training and test datasets. The utilization of early stopping helped prevent overfitting, and careful selection of hyperparameters—particularly the learning rate—contributed to achieving optimal convergence and lower validation loss.