DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION ENGINEERING

UNIVERSITY OF MORATUWA

EN3150 Assignment 02

Learning from data and related

challenges and classification

PUSHPAKUMARA H.M.R.M.

200488E

October 02, 2023

1. Logistic regression weight update process

This question asks us to perform a gradient descent-based weight update for the given data using binary cross entropy as a loss function. We are given the following:

- Data matrix X of dimension N×(D+1), where N is the total number of data samples and D is the number of features.
- Learning rate $\alpha = 0.1$
- Number of iterations $t = 10$

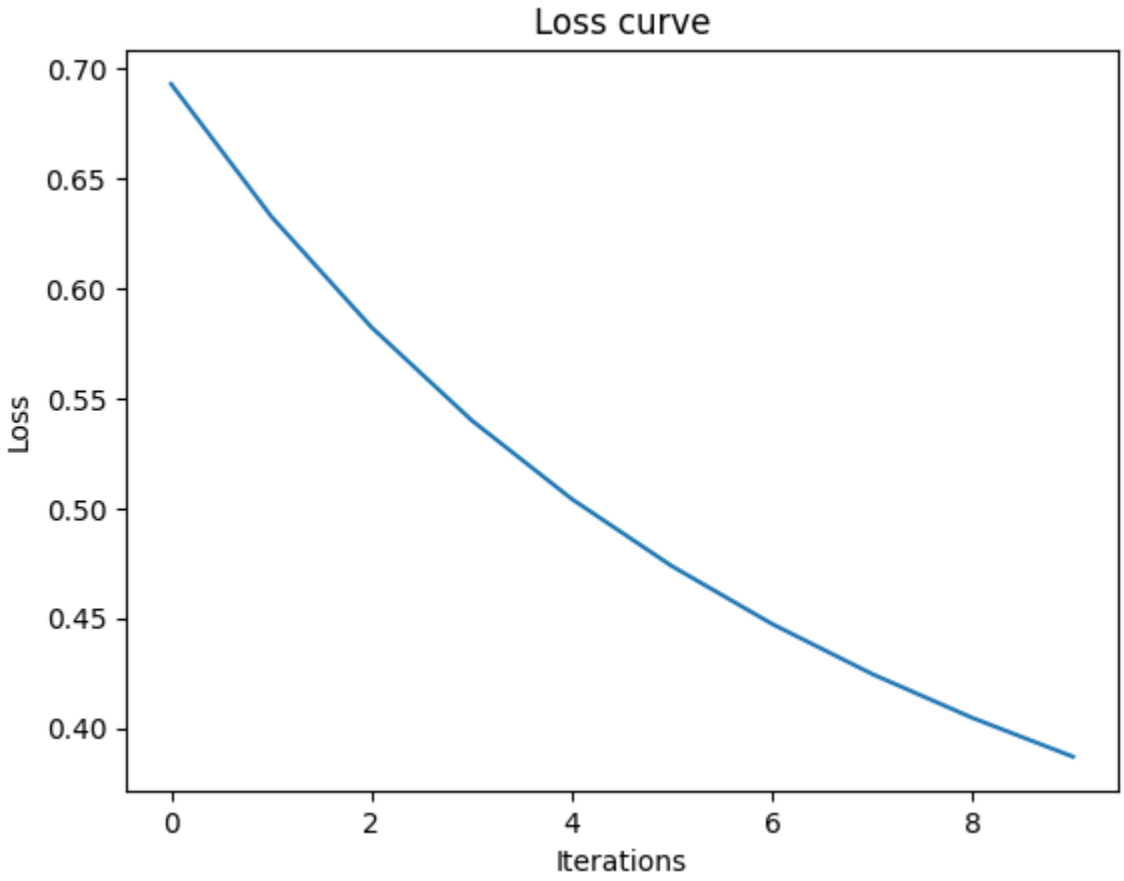
Batch gradient descent weight update is given as follows:

$$\boldsymbol{w}_{(t+1)} \leftarrow \boldsymbol{w}_{(t)} - \alpha \frac{1}{N} \left(\mathbf{1}_N^T \, \mathrm{diag}(\mathrm{sigm}(\boldsymbol{w}_{(t)}^T \boldsymbol{x}_i) - \boldsymbol{y}_i) \boldsymbol{X} \right)^T$$

```
# Perform gradient descent
for i in range(iterations):
    # Calculate predicted probabilities using the sigmoid function
    y_pred = sigmoid(np.dot(X, W))

    # Calculate the logistic loss and store it in the loss_history list
    loss = log_loss(y, y_pred)
    loss_history_BGD.append(np.mean(loss))

    # Calculate the residuals (difference between predicted and true labels)
    residuals = y_pred - y

    # Reshape residuals for matrix multiplication with one_matrix
    residuals = residuals.reshape(1, -1)

    # Create a diagonal matrix with residuals
    diagonal_residuals = np.diag(residuals[0])

    # Compute the gradient and update the weights
    gradient = one_matrix.T @ diagonal_residuals @ X
    gradient = gradient.T / y.shape[0]

    # Update the weights using gradient descent
    W -= learning_rate * gradient
```

Losses after each iteration are as follows:

```
Loss for iteration 0: 0.6931471805599454
Loss for iteration 1: 0.6328211522065751
Loss for iteration 2: 0.5823972949578101
Loss for iteration 3: 0.5400400231011129
Loss for iteration 4: 0.5042051102371915
Loss for iteration 5: 0.4736389534183553
Loss for iteration 6: 0.44734329598374695
Loss for iteration 7: 0.4245298511469433
Loss for iteration 8: 0.4045770169455826
Loss for iteration 9: 0.38699321002389653
```



This question asks us to perform weight updates using Newton's method for binary cross-entropy as the loss function. We initialize the weights as zeros and conduct 10 iterations to update them using this advanced optimization technique.

Batch Newton's method weight update is given below.

$$\boldsymbol{w}_{(t+1)} \leftarrow \boldsymbol{w}_{(t)} - \left(\frac{1}{N}\boldsymbol{X}^T\boldsymbol{S}\boldsymbol{X}\right)^{-1}\left(\frac{1}{N}\left(\mathbf{1}_N^T \operatorname{diag}(\operatorname{sigm}(\boldsymbol{w}_{(t)}^T\boldsymbol{x}_i) - \boldsymbol{y}_i)\boldsymbol{X}\right)^T\right).$$

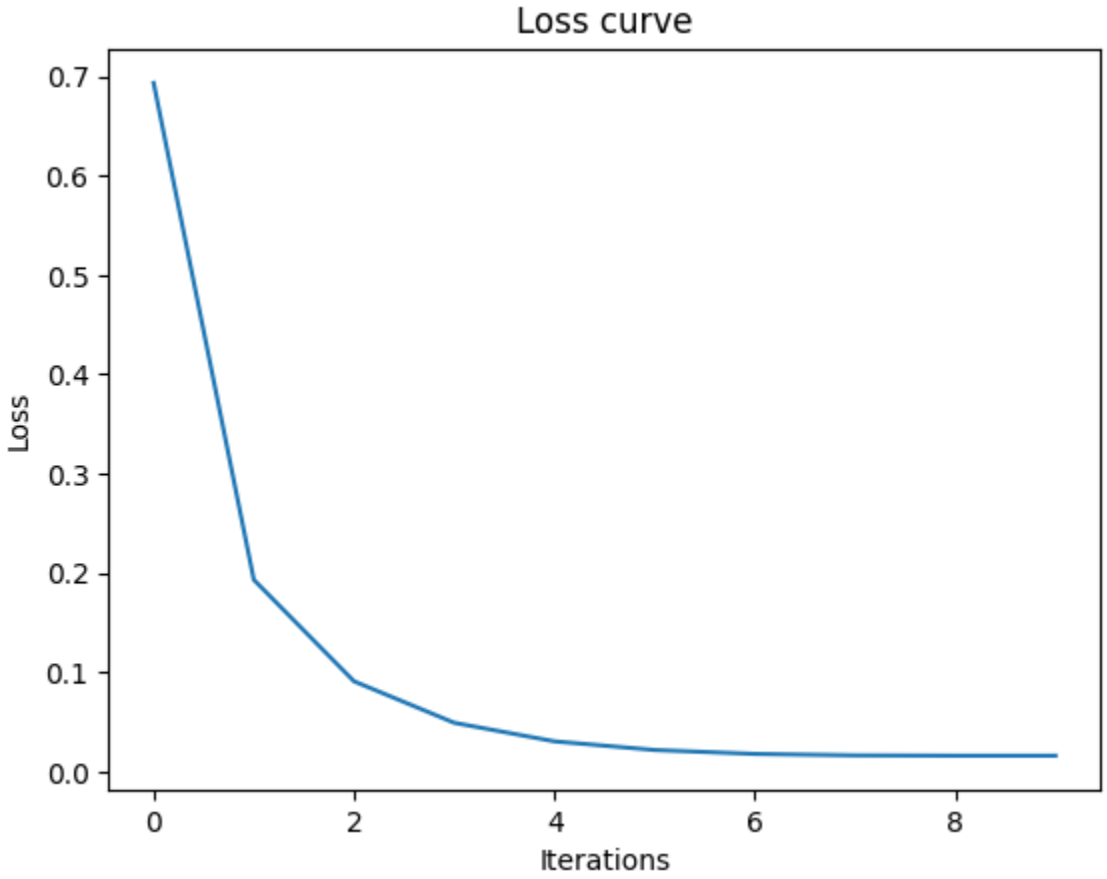
and is $\boldsymbol{S}$ given by

$$\boldsymbol{S} = \operatorname{diag}(s_1, s_2, \ldots, s_N),$$
$$\boldsymbol{s_i} = \left(\operatorname{sigm}(\boldsymbol{w}_{(t)}^T\boldsymbol{x}_i) - \boldsymbol{y}_i\right)\left(1 - \operatorname{sigm}(\boldsymbol{w}_{(t)}^T\boldsymbol{x}_i) - \boldsymbol{y}_i\right).$$

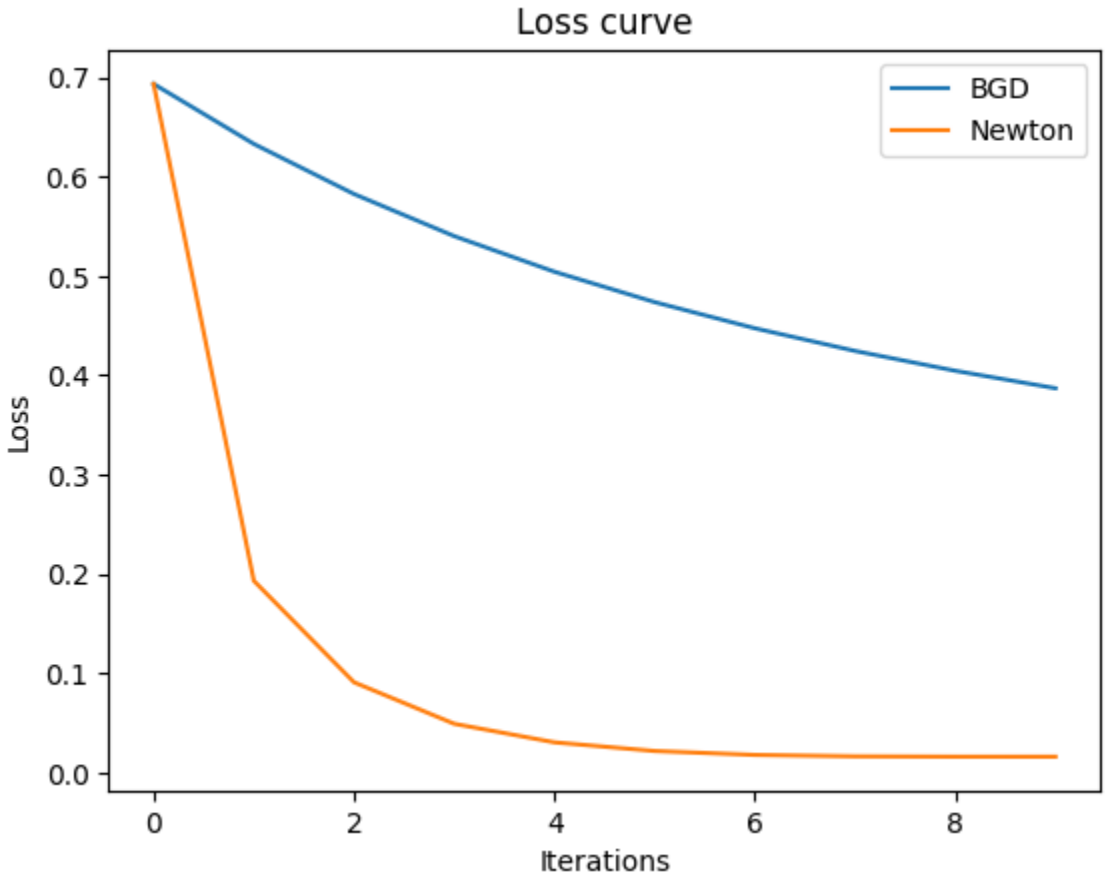
```
for j in range(iterations_newton):
    # Calculate predicted probabilities using the sigmoid function
    y_pred_newton = sigmoid(np.dot(X, W_newton))

    # Calculate the logistic loss and store it in the loss_history list
    loss_newton = log_loss(y, y_pred_newton)
    loss_history_Newton.append(np.mean(loss_newton))

    # Calculate the residuals (difference between predicted and true labels)
    residuals = y_pred_newton - y

    # Define the learning rate
    s = (y_pred_newton - y) * (1 - y_pred_newton - y)

    # Create a diagonal matrix with s
    S = np.diag(s.reshape(-1))

    learning_rate = np.linalg.inv((1 / X.shape[0]) * (X.T @ S @ X))

    # Reshape residuals for matrix multiplication with one_matrix
    residuals = residuals.reshape(1, -1)

    # Create a diagonal matrix with residuals
    diagonal_residuals = np.diag(residuals[0])

    # Compute the gradient and update the weights
    gradient = one_matrix.T @ diagonal_residuals @ X
    gradient = gradient.T / y.shape[0]


    # Update the weights using gradient descent
    W_newton -= learning_rate @ gradient
```

Losses after each iteration are follows:

```
Loss for iteration 0: 0.6931471805599454
Loss for iteration 1: 0.1932642413241019
Loss for iteration 2: 0.09113723316903417
Loss for iteration 3: 0.0493805881073356 64
Loss for iteration 4: 0.0307160745984797 44
Loss for iteration 5: 0.0220225499319704 6
Loss for iteration 6: 0.0180735604539971 9
Loss for iteration 7: 0.0166180327226281 5
Loss for iteration 8: 0.0163234577557707 03
Loss for iteration 9: 0.0163071962292348 1
```



Loss with respect to number of iterations for both Gradient descent and Newton method's



In both approaches, the loss starts at a value of 0.6931 since the initial W values are zero.

In the Batch Gradient Descent approach, the loss decreases significantly from iteration 0 to iteration 5, indicating rapid early learning, and after the 5^{th} iteration, the rate of decrease slows down. After the 10^{th} iteration, the loss is approximately 0.3869.

In Newton's method approach, the loss drops significantly from iteration 0 to iteration 1. It continued to decrease rapidly over the next few iterations, reaching a low loss value of approximately 0.013 after the 10^{th} iteration.

When we compare these two approaches, we can identify that the Batch Gradient Descent shows a slower convergence rate when compared with Newton's method. Newton's method shows rapid convergence due to its ability to take curvature information into account.

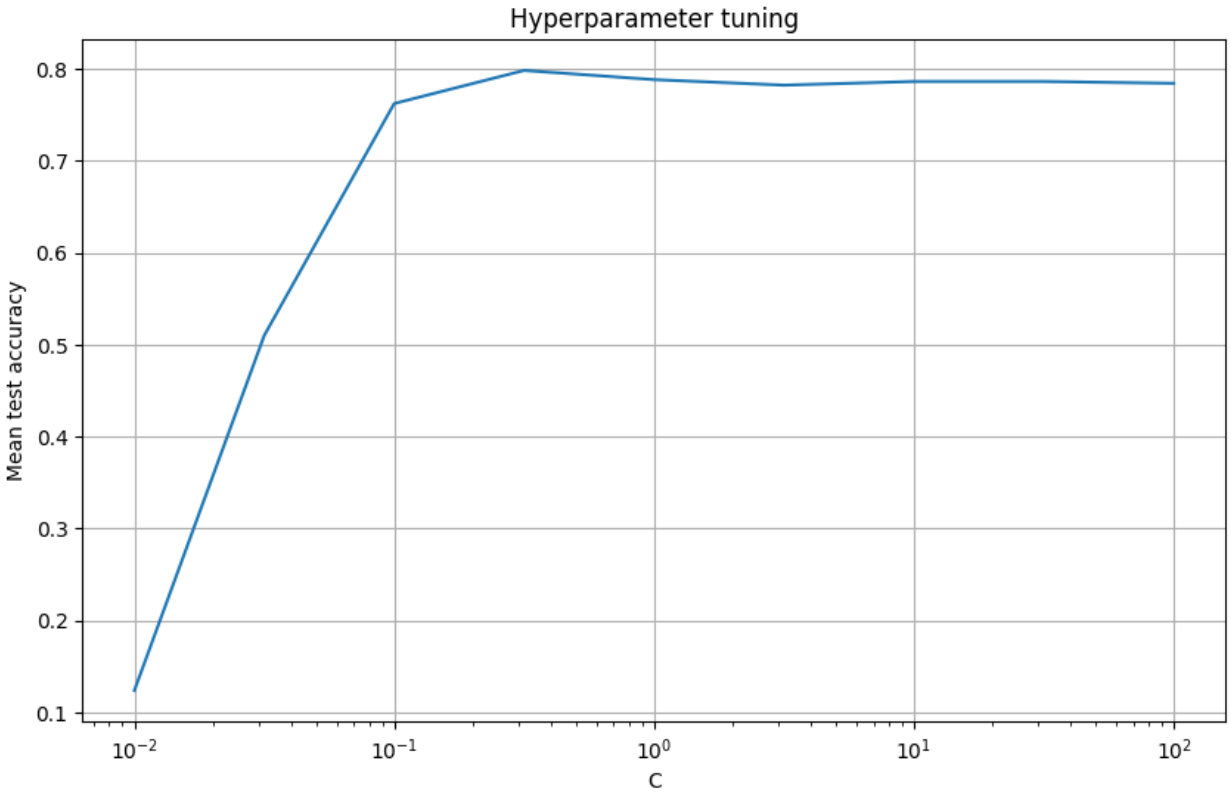
2. Perform grid search for hyper-parameter tuning

This task involves using Lasso logistic regression for image classification with specific settings: 'penalty' set to 'l1', 'solver' set to 'liblinear', and 'multi_class' set to 'auto'. To optimize the model's hyperparameter 'C', we created a machine learning pipeline that includes data scaling, the Lasso logistic regression estimator, and a parameter grid for hyperparameter tuning. We employ GridSearchCV to systematically explore a range of 'C' values spanning from 0.01 to 100 in order to find the optimal value for this hyperparameter. The goal is to identify the 'C' value that maximizes the model's performance for image classification.

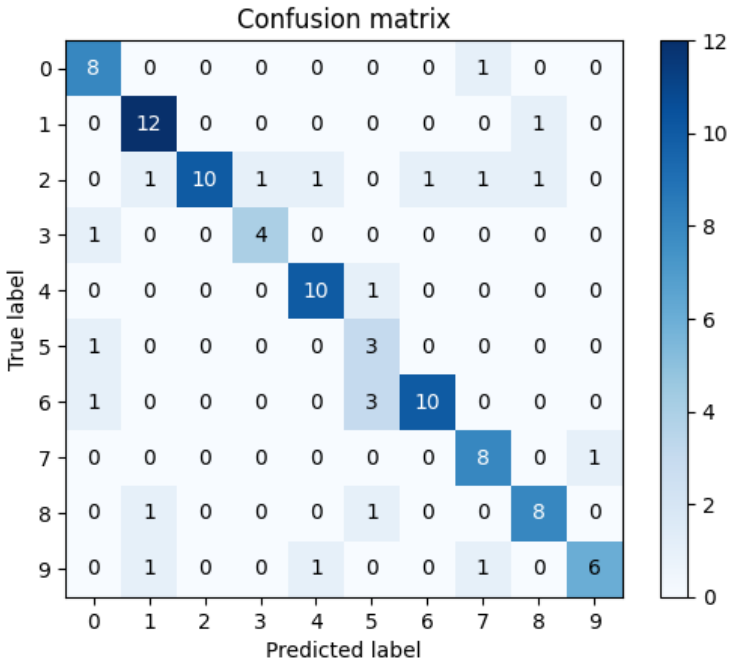
```
# Create a pipeline for Lasso Logistic Regression
lasso_logistic_pipeline = Pipeline([
    ("scaler", StandardScaler()),  # Standardize features
    ("lasso_logistic", LogisticRegression(penalty='l1', solver='liblinear', multi_class='auto'))  # Lasso Logistic Regression
])

# Hyperparameter grid for tuning C (inverse of regularization strength)
param_grid = {
    'lasso_logistic__C': np.logspace(-2, 2, 9)  # Range for C values
}

# Create a GridSearchCV instance for hyperparameter tuning
grid_search = GridSearchCV(lasso_logistic_pipeline, param_grid, cv=5, n_jobs=-1)

# Fit the model with training data
grid_search.fit(X_train, y_train)

# Print the best hyperparameters and accuracy
print("Best hyperparameters:", grid_search.best_params_)
print("Best accuracy:", grid_search.best_score_)

# Predict using the best estimator
y_pred = grid_search.best_estimator_.predict(X_test)

# Evaluate accuracy on the test set
test_accuracy = accuracy_score(y_test, y_pred)
print("Test set accuracy:", test_accuracy)
```

```
Best hyperparameters: {'lasso_logistic__C': 0.31622776601683794}
Best accuracy: 0.798
Test set accuracy: 0.79
```

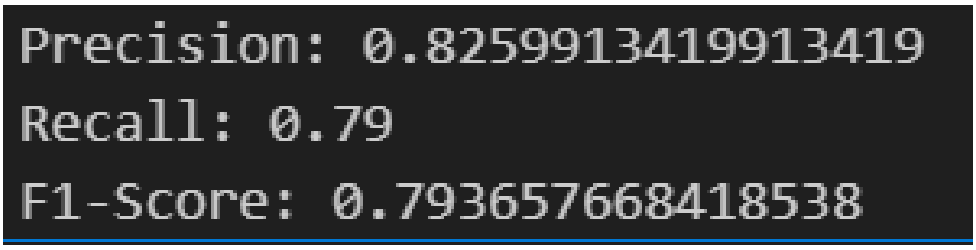
Plot of classification accuracy with respect to hyperparameter C



Confusion matrix



Precision, Recall, and F1-score



- The confusion matrix provides a detailed breakdown of the classification results for each class. Each row corresponds to the actual class, while each column represents the predicted class.
- Precision (0.826) measures the accuracy of positive predictions. It indicates that out of all the samples predicted as positive, approximately 82.6% were correct.
- Recall (0.79) measures the ability of the model to correctly identify all relevant instances. It suggests that the model correctly identified about 79% of all positive instances.
- The F1-Score (0.794) is the harmonic mean of precision and recall and provides a balanced measure of a model's performance. An F1-Score of 0.794 indicates a good overall performance of the model.
- Overall, the model appears to perform well with relatively high precision, recall, and F1-Score values. However, further domain-specific considerations might be needed to assess whether these metrics meet the desired level of performance for the specific application.

3. Logistic regression

a) To estimate the probability that a student, who has studied for 40 hours (x_1) and has an undergraduate GPA of 3.5 (x_2), will receive an A+ in the class (y = 1),

we can use the logistic regression equation:

$$P(y = 1) = \frac{e^{\mu}}{1 + e^{\mu}}$$

$$\mu = w_0 + w_1 x_1 + w_2 x_2$$

Where:

- $w_0 = -6$
- $w_1 = 0.05$
- $w_2 = 1$

Now here:

- $x_1 = 40$
- $x_2 = 3.5$

We can calculate:

$$\mu = -6 + 0.05 \times 40 + 1 \times 3.5$$

$$\mu = -0.5$$

Then,

$$\boldsymbol{P(y = 1) = \frac{e^{-0.5}}{1 + e^{-0.5}} = 0.3775}$$

b) To achieve a 50% chance of receiving an A+ in the class,

c)

$$P(y = 1) = \frac{e^{\mu}}{1 + e^{\mu}}$$

The student has a 3.5 GPA. Therefore $x_2 = 3.5$. We have to find x_1;

$$\mu = -6 + 0.05 \times x_1 + 1 \times 3.5$$

$$P(y = 1) = \frac{e^{\mu}}{1 + e^{\mu}}$$

$$0.5 = \frac{e^{\mu}}{1 + e^{\mu}}$$

$$0.5 + 0.5e^{\mu} = e^{\mu}$$

$$e^{\mu} = 1$$

$$\mu = 0$$

$$-6 + 0.05 \times x_1 + 1 \times 3.5 = 0$$

$$0.05 \times x_1 = 2.5$$

$$\boldsymbol{x_1 = 50\ hrs}$$