Analysis of Linear and Logistic Regression on the Iris Dataset

February 2022

Dataset Introduction, Processing, and Visualization

The Setosa samples are marked as class 0. Versicolor and Virginica together are marked as class 1. The dataset is split into training and testing parts randomly. 10% of the samples are used for testing. The visualizations are provided in Figures 1 and 2.

In Figure 1, the samples are visualized as a scatter plot in 2 dimensions - one feature being along one dimension. Since there are 4 features, there will be 12 such plots. The remaining 4 plots which appear along the diagonal of the figure are redundant because the first and the second dimensions are the same. In Figure 1, the training set is covered while in Figure 2 the test set is covered.

There are a total of 150 samples in the dataset and each sample has 4 features - petal length and width, sepal length and width. It is an introductory dataset for machine learning learners and can be used for both classification and regression tasks.

Linear Regression - Implementation and Analysis

12 Linear Regression models are trained on various permutations of the input features. These permutations correspond exactly to the ones shown in Figure 1. The feature along the x axis is taken as the independent variable and the feature along the y axis is taken as the dependent variable. As mentioned in the question, no scikit-learn libraries are used for regression. The models are trained with a learning rate of 0.1 for 100 epochs at batch size 32. The resulting learning curves are shown in Figure 3.

The code for the implementation of linear regression is shown below.

```
def sgd_regression(X):
    """
    This function implements the stochastic gradient descent for
    linear regression
    Input:
```

```
        X - the numpy array of the inpendent and dependent features
    Output:
        losses - average loss across batches per epoch
        w - the weight vector for linear regression line
    """
    ## create the data
    losses = []
    n_batches = X.shape[0]//32 + 1
    Xd = np.zeros((X.shape[0], 2))
    Xd[:, 0] = X[:, 0]
    Xd[:, 1] = 1
    ## create the initialization for weights
    w = np.array([[1.], [1.]])
    for i in range(100):
        loss = 0
        for j in range(n_batches):
            # extract the batch from the data
            x = Xd[j*n_batches:(j+1)*n_batches, :]
            y = X[j*n_batches:(j+1)*n_batches, 1].reshape((-1, 1))
            # compute the error
            err = y - np.dot(x, w)
            # compute the loss
            loss = loss + 0.5*np.linalg.norm(err)**2/32
            # gradient descent steps
            w[0, 0] = w[0, 0] + 0.1*np.dot(x[:, 0].reshape((1, -1)), err)/32
            w[1, 0] = w[1, 0] + 0.1*err.sum()/32
        loss = loss/n_batches
        losses.append(loss)
    return losses, w
```

Effect of Regularization

The effect of regularization is studied on one model. The input is petal length and the output is petal width. Without regularization, the weights are [2.1829, 1.0891]. With regularization, the weights are [0.3801, -0.1717]. Regularization has the effect of reducing the weights. The code for the L2 regularized linear regression is shown below.

```
def sgd_regression_regularized(X):
    """
    This function implements the stochastic gradient descent for linear regression with L2 regularization
    Input:
        X - the numpy array of the inpendent and dependent features
    Output:
        losses - average loss across batches per epoch
        w - the weight vector for linear regression line
    """
    ## create the data
    losses = []
    n_batches = X.shape[0]//32 + 1
    Xd = np.zeros((X.shape[0], 2))
    Xd[:, 0] = X[:, 0]
    Xd[:, 1] = 1
```

```
    ## create the initialization for weights
    w = np.array([[1.], [1.]])
    for i in range(100):
        loss = 0
        for j in range(n_batches):
            # extract the batch from the data
            x = Xd[j*n_batches:(j+1)*n_batches, :]
            y = X[j*n_batches:(j+1)*n_batches, 1].reshape((-1, 1))
            # compute the error
            err = y - np.dot(x, w)
            # compute the loss
            loss = loss + 0.5*np.linalg.norm(err)**2/32
            # gradient descent steps with L2 regularization
            w[0, 0] = w[0, 0]*(1.0 - 0.1*1.0/32) + 0.1*np.dot(x[:,
    0].reshape((1, -1)), err)/32
            w[1, 0] = w[1, 0]*(1.0 - 0.1*1.0/32) + 0.1*err.sum()/32
        loss = loss/n_batches
        losses.append(loss)
    return losses, w
```

The performance of the resulting models on the test set are shown in the Figure 4. From Figure 4, it is clear that the model which predicts sepal width from petal width is the most predictive.

Logistic Regression - Implementation and Analysis

6 Logistic Regression models are trained on various permutations of the input features. There are 4 input features in total and the number of distinct feature combinations that we can have is ${}^4C_2 = 6$. The two features are the independent features for input into the algorithm and the output is the output label, or the class of the sample. The models are trained with a learning rate of 0.1 for 100 epochs at batch size 32. The resulting learning curves are shown in Figure 5. The code the L2 regularized logistic regression is shown below.

```
def sgd_classification_regularized(X, y, lr):
    """
    This function implements the stochastic gradient descent for
    logistic regression with L2 regularization
    Input:
        X - the numpy array of the inpendent features
        y - the output labels of the corresponding samples
    Output:
        losses - average loss across batches per epoch
        w - the weight vector for linear regression line
    """
    ## create the data
    losses = []
    n_batches = X.shape[0]//32 + 1
    Xd = np.zeros((X.shape[0], 3))
    Xd[:, 0] = X[:, 0]
    Xd[:, 1] = X[:, 1]
    Xd[:, 2] = 1
```

```
    ## create the initialization for weights
    w = np.array([[1.], [1.], [1.]])
    for i in range(100):
        loss = 0
        for j in range(n_batches):
            # extract the batch from the data
            x = Xd[j*n_batches:(j+1)*n_batches, :]
            y = X[j*n_batches:(j+1)*n_batches, 1].reshape((-1, 1))
            y_pred = 1./(1. + np.exp(-np.dot(x, w)))
            # compute the error
            err = y - y_pred
            # compute the loss
            loss = loss + 0.5*np.linalg.norm(err)**2/32
            # gradient descent steps with L2 regularization
            w[0, 0] = w[0, 0]*(1.0 - lr*1.0/32) + lr*np.dot(x[:,
    0].reshape((1, -1)), err)/32
            w[1, 0] = w[1, 0]*(1.0 - lr*1.0/32) + lr*np.dot(x[:,
    1].reshape((1, -1)), err)/32
            w[2, 0] = w[2, 0]*(1.0 - lr*1.0/32) + lr*err.sum()/32
        loss = loss/n_batches
        losses.append(loss)
    return losses, w
```

Effect of Learning Rate

The effect of learning rate is studied on one of the models. This model uses petal width and sepal width as features for classification. The learning curves are shown in Figure 6. From the figure, it is clear that the model with learning rate 0.1 converges the fastest.

The accuracy of these models on the test set are shown in Figure 7. It can be seen that each feature combination is equally discriminative.

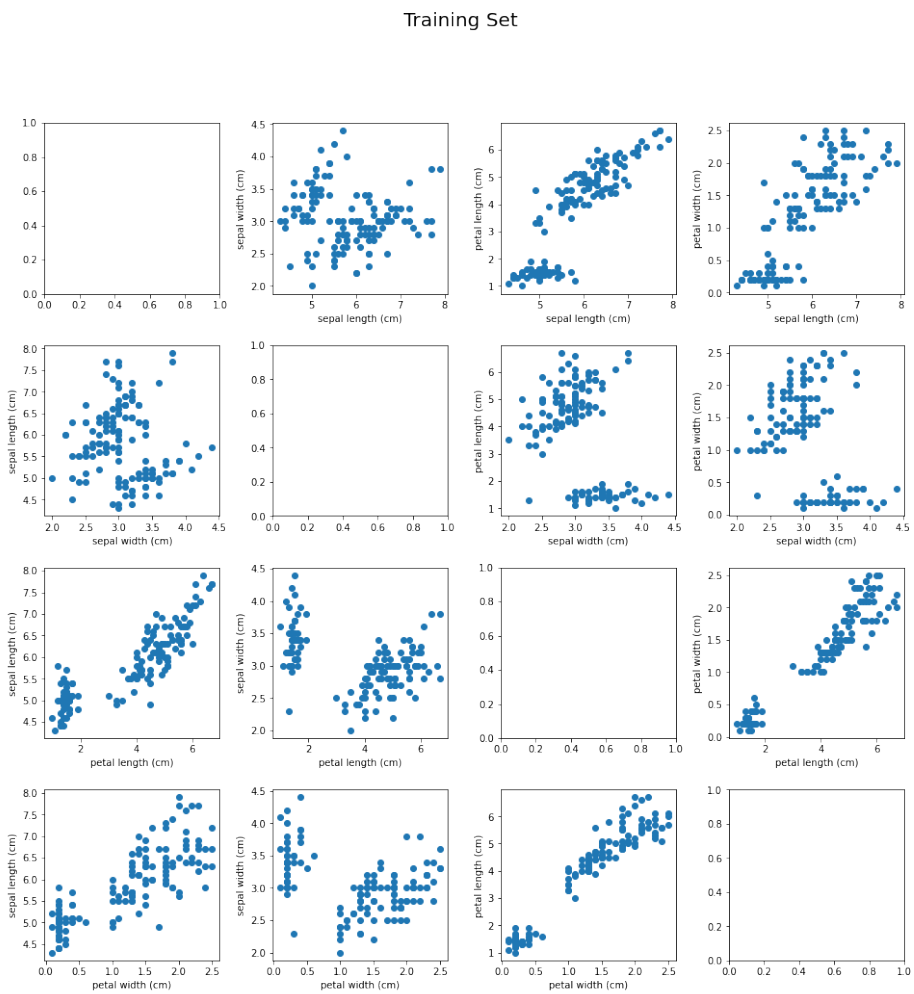

Figure 1: Training Set

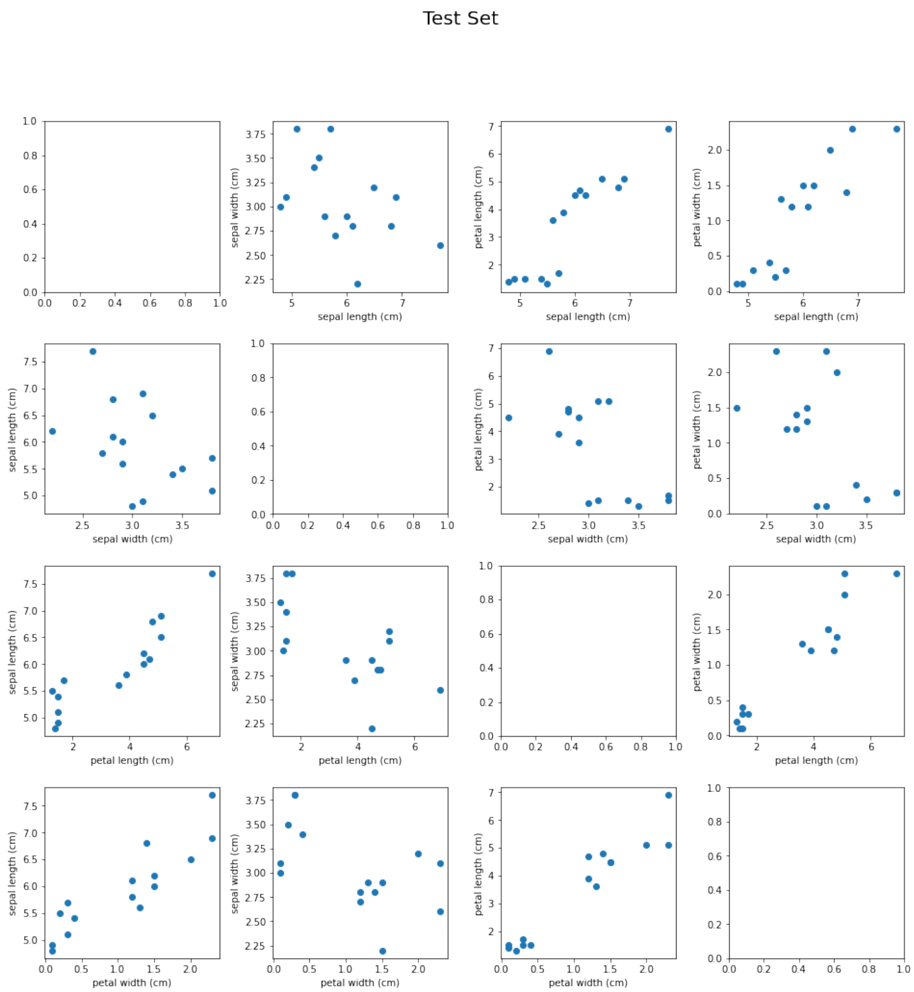

Figure 2: Test Set

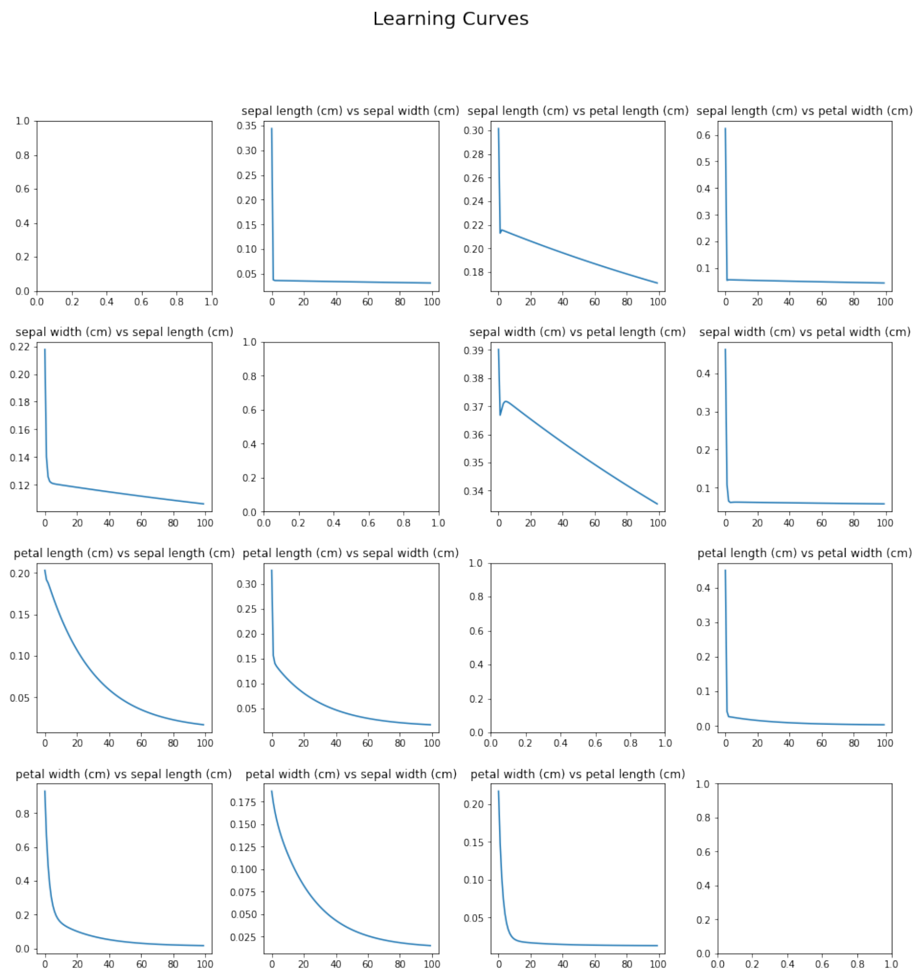

Figure 3: Learning Curves from Linear Regression

X	Y	LOSS
sepal length (cm)	sepal width (cm)	0.22974048174270997
sepal length (cm)	petal length (cm)	0.9130884128785335
sepal length (cm)	petal width (cm)	0.22491095120575755
sepal width (cm)	sepal length (cm)	0.7299659402787538
sepal width (cm)	petal length (cm)	1.780190807689164
sepal width (cm)	petal width (cm)	0.28416397548278766
petal length (cm)	sepal length (cm)	0.1276017664308557
petal length (cm)	sepal width (cm)	0.08858520324931805
petal length (cm)	petal width (cm)	33.609569514829076
petal width (cm)	sepal length (cm)	0.12167772302800745
petal width (cm)	sepal width (cm)	0.08092217611818574
petal width (cm)	petal length (cm)	0.1244781390636703

Figure 4: Losses from Linear Regression

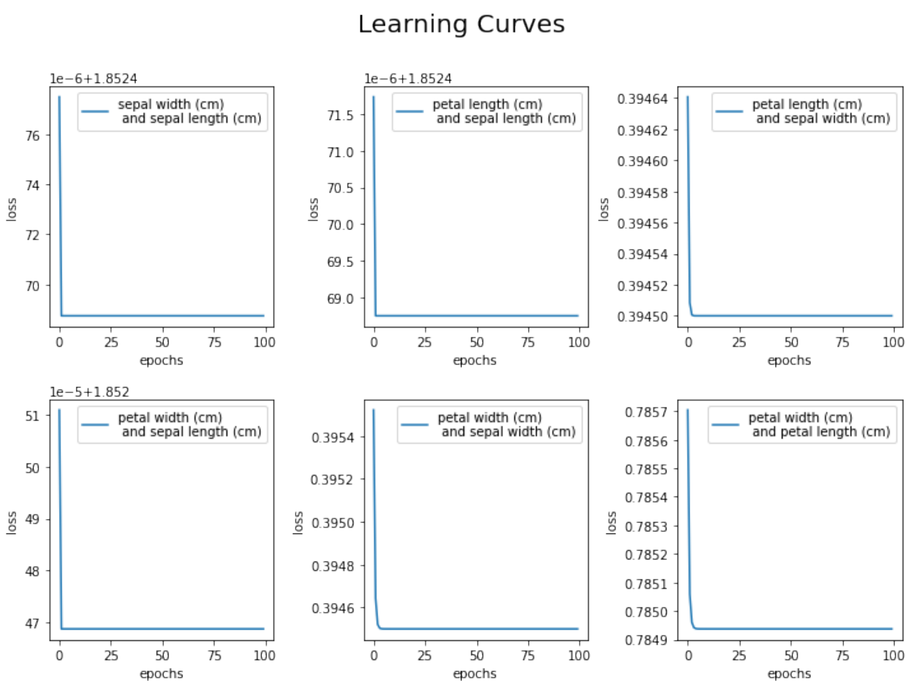

Figure 5: Learning Curves for Logistic Regression

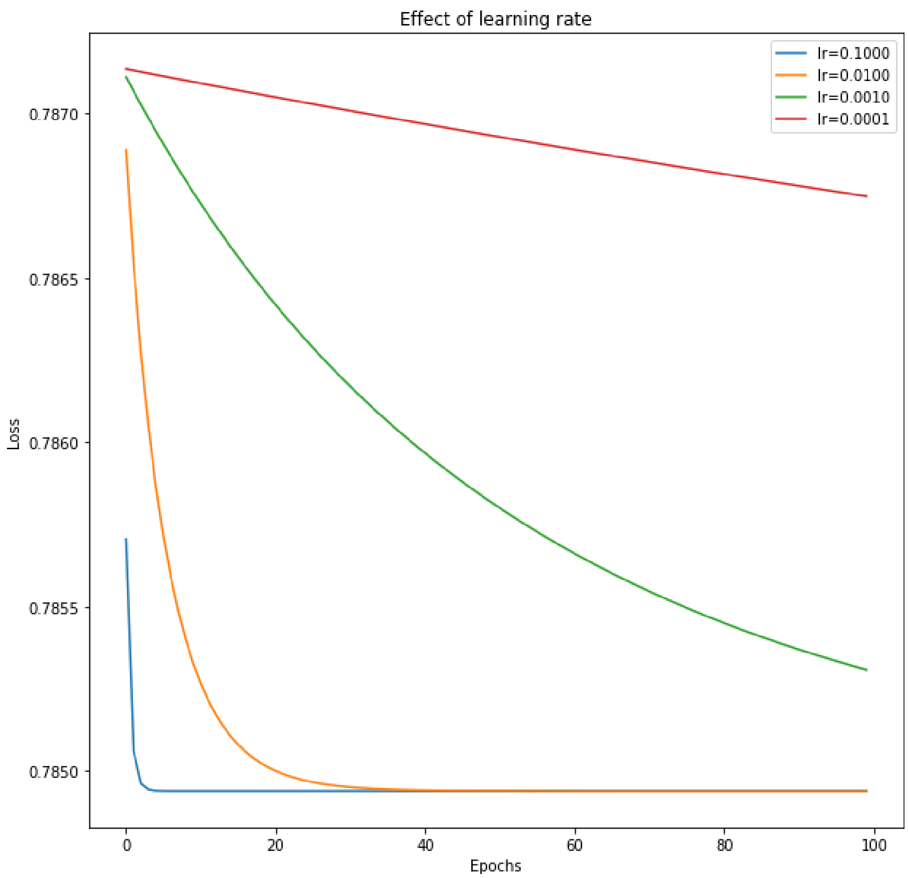

Figure 6: Effect of learning rate for Logistic Regression

X	Y	ACCURACY
sepal width (cm)	sepal length (cm)	0.6
petal length (cm)	sepal length (cm)	0.6
petal length (cm)	sepal width (cm)	0.6
petal width (cm)	sepal length (cm)	0.6
petal width (cm)	sepal width (cm)	0.6
petal width (cm)	petal length (cm)	0.6

Figure 7: Accuracies for Logistic Regression