Homework 2
CS 5787 Deep Learning
Spring 2023

Instructor: Prof. Alejandro (Alex) Jaimes

Student: Mykyta Turpitka

netID: mt689

Due: 3/6/23

Problem 1 - Linear Algebra Review #1

Let $\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 5 & -2 \end{pmatrix}$ and $\mathbf{B} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 2 \end{pmatrix}$
.

Part 1 (3 points)

Compute $Trace(\mathbf{A})$.

The Trace of a matrix is computed by summing the elements along the main diagonal. Thus, the $Trace(\mathbf{A}) = 1 + (-2) = -1$.

Answer:
$Trace(\mathbf{A}) = -1$.

Part 2 (4 points)

Compute $Trace(\mathbf{BB}^T)$.

First, we need to compute the resulting $\mathbf{BB}^T$ matrix

$\mathbf{BB^T} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 0 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 2 & 8 \end{pmatrix}$

Finally, compute the sum of the elements along the main diagonal of the new matrix to obtain the Trace. $Trace(\mathbf{BB^T}) = 1 + 8 = 9$.

Answer:
$Trace(\mathbf{BB}^T) = 9$.

Problem 2 - Linear Algebra Review #2 (4 points)

Let $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\mathbf{D} \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$, and $\mathbf{x}^T(\mathbf{A} + \mathbf{D}) = \mathbf{b}^T$. Use the matrix inverse to solve for $\mathbf{x}$ and simplify. Assume that $\det(\mathbf{A} + \mathbf{D}) \neq 0$.

According to matrix properties, multiplying a matrix by its own inverse produces an identity matrix. This, we can start by multiplying both sides of the equation by $(\mathbf{A}+\mathbf{D})^{-1}$

Thus,

$\mathbf{x}^T = \mathbf{b}^T \cdot (\mathbf{A} + \mathbf{D})^{\mathbf{-1}}$

Next, we can take a transpose of both sides to single out x:

$\mathbf{x} = \mathbf{b} \cdot (\mathbf{A} + \mathbf{D})^{\mathbf{-T}}$

Answer:
$\mathbf{x} = \mathbf{b} \cdot (\mathbf{A} + \mathbf{D})^{\mathbf{-T}}$

Problem 3 - Linear Algebra Review #3

Let matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and matrix $\mathbf{B} \in \mathbb{R}^{n\times m}$, where $n \neq m$.

Part 1 (1 point)

If it is possible to compute the matrix product $\mathbf{AB}$, give the size of the output.

Answer: It is possible to compute the matrix product the size of the output matrix will be $\mathbf{n \times m}$

Part 2 (1 point)

If it is possible to compute the matrix product $\mathbf{BA}$, give the size of the matrix produced.

Answer: It is not possible to compute the matrix product BA because $\mathbf{n \neq m}$

Problem 4 - Regression

Part 1 - Load and Explore the Data (5 points)

To load the dataset, I defined the following function. Note, all of the functions are available in the notebook. The function adds a bias term to each of the rows in the dataset (feature vectors) and returns numpy arrays that represent partitions of the dataset into training and test sets.

```python
def load_MSD(fname, addBias=True):
    data = np.loadtxt(fname, delimiter=',')
    year, features = data[:, 0], data[:, 1:]
    if addBias:
        features = np.hstack((features, np.ones((features.shape[0], 1))))
    trainYears = year[:463714]
    trainFeat = features[:463714]
    testYears = year[463714:]
    testFeat = features[463714:]
    return trainYears, trainFeat, testYears, testFeat
```

Next, we proceed to the musicMSE(grad, gt) function. The function computes mean squared error between predictions and the true values of year. It is important to note, that before the comparison is made, we need to round the prediction to the nearest year. Following is Python implementation of the function:

```python
def musicMSE(pred, gt):
    pred = np.round(pred)
    mse = np.sum(np.square(gt - pred)) / len(gt)
    return mse
```

The following are some basic stats for the data:

Labels (years):

- Range: [1922, 2011]

- Standard deviation: 10.93

- Median: 2002

Features:

- Range: [-14861.69535, 65735.77953]

- Standard deviation: 508.48

- Mean: 116.28

It is worth pointing out that these descriptive statistics for features don't provide any useful information. A better way to approach this would be to find the descriptive statistics for each of the 90 features. This could be done with Pandas describe() method.

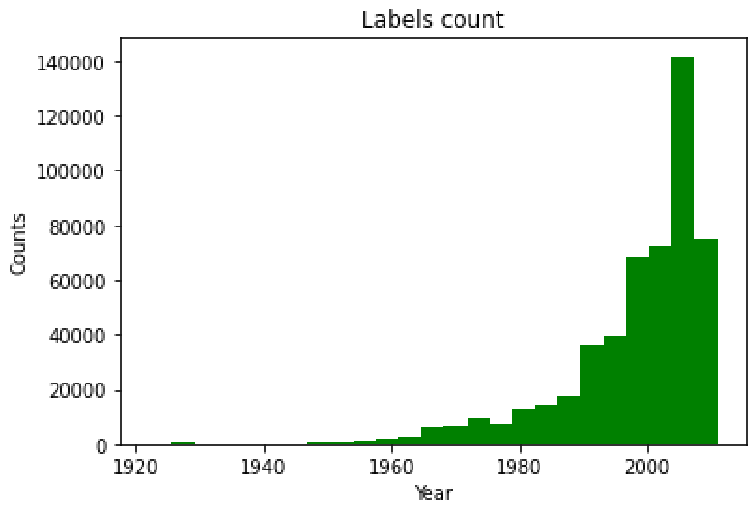
We can normalize the features by 'z-score' normalization. To do this, I defined an normalize_MSD function which could be found in the notebook. In the function, I first strip the feature vectors of the added bias term, then calculate z-scores for each of the features by subtracting its mean and dividing by its standard deviation for each of the feature columns. I then add the bias terms back before returning normalized feature vectors. The implementation:

```python
def normalize_MSD(trainFeat, testFeat):
    trainX = trainFeat[:, :-1] #remove bias
    testX = testFeat[:, :-1] #remove bias
    mean_trainX = np.mean(trainX, axis=0)
    std_trainX = np.sqrt(np.sum(np.square(trainX - mean_trainX), axis=0) / (
        trainX.shape[0] - 1))
    normalized_trainFeat = (trainX - mean_trainX) / std_trainX
    normalized_testFeat = (testX - mean_trainX) / std_trainX

    normalized_trainFeat = np.hstack((normalized_trainFeat, np.ones((trainX.shape
        [0], 1)))) #add bias back
    normalized_testFeat = np.hstack((normalized_testFeat, np.ones((testX.shape
        [0], 1)))) #add bias back

        return normalized_trainFeat, normalized_testFeat
```

Finally, to determine what would the test MSE be if our classifier always outputs the most common year in the dataset we first need to find the mode label. Using scipy.stats.mode() we obtain the year 2007 as the mode of the labels array with 39404 total occurrences. Next, we use 2007 as gt (ground truth) in the musicMSE function that I have implemented earlier:

Part 2 - Classification? (5 points)

The problem with this approach would be that this will turn this problem into a multi-class classification problem. Additionally, as can be seen from the chart below, the distribution of the year labels in the dataset is heavily skewed left. In this scenario, any common multi-class classification approach would not output the best results.



Part 3 - Implementing Ridge (Tikhonov) Regression (6 points)

I followed the advice and implemented a minibatch_SGD() function according to specifications. It takes in all the parameters as mentioned, runs SGD with minibatches, plots the test and train loss as a function of epochs and outputs the vector of weights as it appears at the end of training.

For this and the following parts 4, 5, and 6 I am not going to include the implementation of the minibatch_SGD() function in the written report. Instead, I will include one-liners which contain the function calls to minibatch_SGD and focus on the output.

For this part, first, let's look at the implementation of the pseudo-inverse. I broke the formula up into several parts for ease of reading. I also had do add an additional transpose after the last X in order to match dimensions with y.

```
def pseudoInverse(X, y, alpha):
    XXT = X @ X.T
    Ia = np.identity(X.shape[0]) * alpha
    w = (np.linalg.inv(XXT + Ia) @ X).T @ np.expand_dims(y, axis=1)
    return w
```

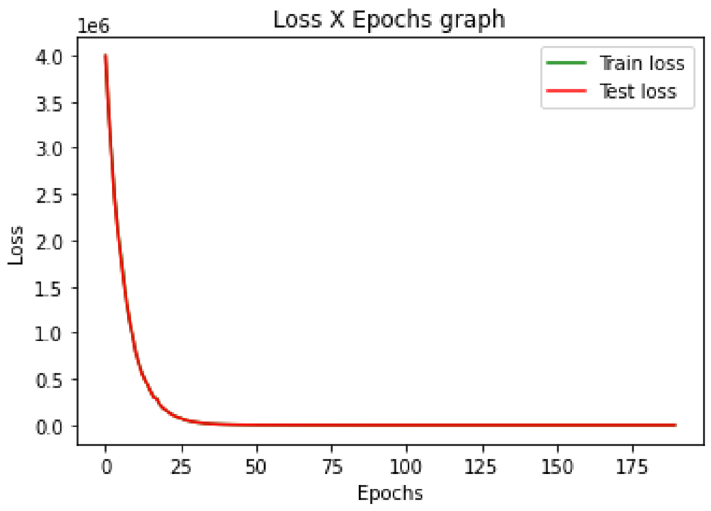
Next, we are going to obtain the weight vectors from both pseudo-inverse and Ridge Regression and compare them. The only limitation to my implementation is that I am not able to run the pseudo-inverse on the complete data-set as apparently I don't have enough RAM. This is exactly why in this case the mini-batched approach to SGD is great, it allows to reduce the amount of RAM required to complete the task.

```python
w_pseudoinverse = pseudoInverse(ntrainFeat[:10000], trainYears[:10000], alpha
    =0.01)

weights_Ridge = minibatch_SGD(ntrainFeat, trainYears, ntestFeat, testYears,
    weight_decay_factor=0.0001, wd_type = 'L2', loss_f="L2", learning_rate=0.04,
    batch_size=128, n_epochs=190)
```

The learning rate, weight decay factor, number of epochs, and the batch size that is passed into the minibatch_SGD() above appear to be ones that minimize the training loss.

Comparing the two weight vectors, I get the result that suggests that my implementation of the Ridge Regression is most likely correct. I subtracted the Ridge weights from the pseudo-inverse weights and the minimum difference is -3.53, max difference is 3.02, and the mean difference is -0.03, which is very close to 0. This suggests that if I am able to run the pseudo-inverse on the complete data-set as well as spend more time tuning the parameters for Ridge, the difference between the two vectors will be close to 0. This validates my solution for this question.

Finally, below is the plot showing the test and train losses as the function of epochs:

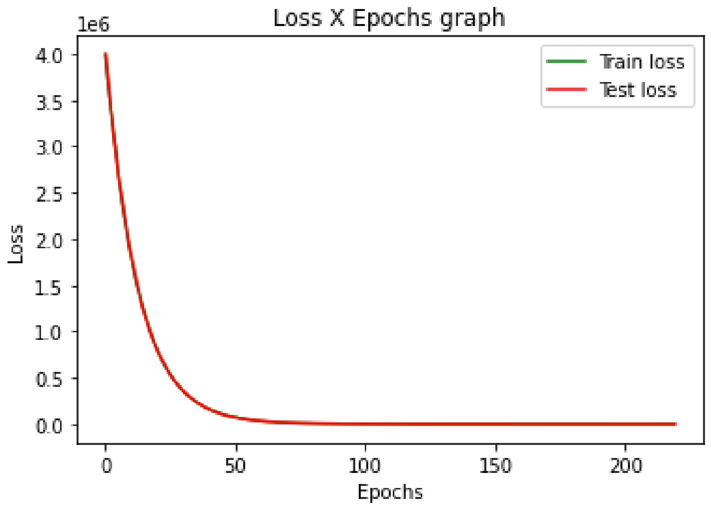


Part 4 - Implementing L_1 Weight Decay (6 points)

Similar to the Ridge regression, I am only including the function call and the Loss X Epochs graph. I noticed that increasing the batch size has a great effect on the accuracy of predictions, but increasing it too much contradicts to the whole idea of mini-batch optimization. Therefore, I set it to be 150.

```
weights_Lasso = minibatch_SGD(ntrainFeat, trainYears, ntestFeat, testYears,
    weight_decay_factor=0.0009, wd_type = 'L1', loss_f="L2", learning_rate
    =0.02, batch_size=150, n_epochs=220)
```

Below is the graph that shows training and test set losses as function of the number of epochs:

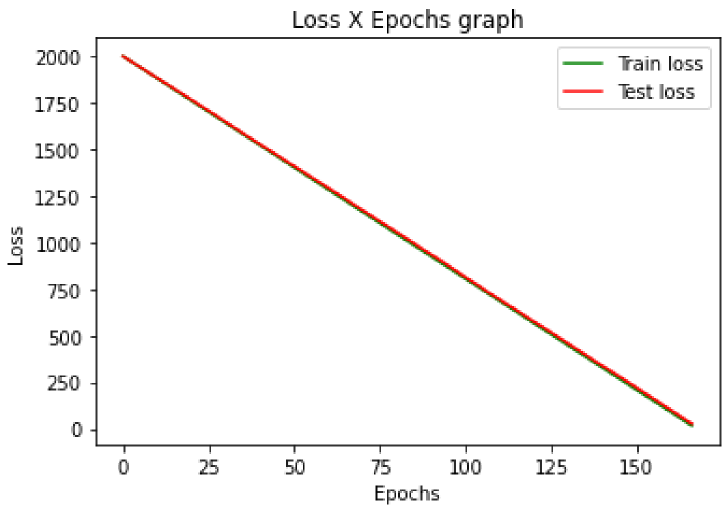


Part 5 - Implementing Count Regression (6 points)

Refer to the notebook for the count regression implementation. I am not sure whether I got it right because I am only able to get the losses go down if the learning rate is extremely low. Here is the function call that produces a meaningful loss reduction. However, it is important to point out that the loss keeps going down into the negative numbers if we use more than 167 epochs with these input parameters.

```
weights_count = minibatch_SGD(ntrainFeat, trainYears, ntestFeat, testYears,
    weight_decay_factor=0.01, wd_type = 'none', loss_f="count", learning_rate
    =0.000003, batch_size=64, n_epochs=167)
```

The following is the train and test loss graphs as a function of epochs. It is evident that the losses are going towards negative infinity.

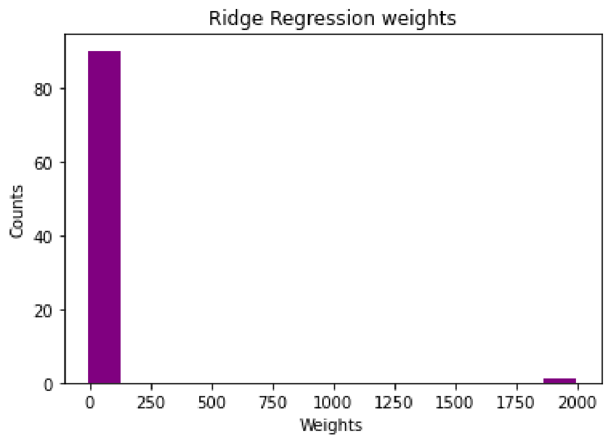


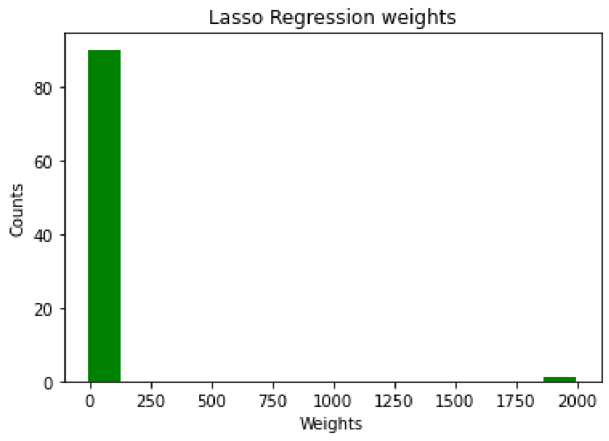
Solution:

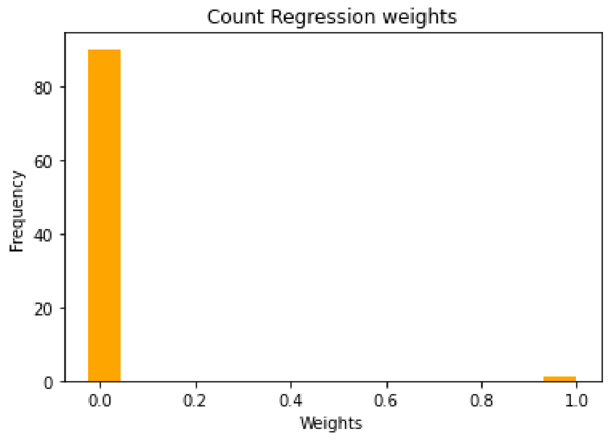
Part 6 - Model Comparison (10 points)

The three plots below are the histograms for the weights from Ridge, Lasso, and Count regressions respectively. We can notice that for the Ridge and Lasso regressions the weight distribution is very similar. Most of the weights are between 0 and about 170 with an insignificant number of outliers closer to 2000. In Count Regression, though, the weights are much smaller and are situated around just below 0 and 0.1. There is also an insignificant amount of outliers around 1, following a similar pattern to that of the Ridge and Lasso.

While the Ridge and Lasso did not exhibit any unexpected behavior patterns, the count regression was definitely way more noteworthy. First, the function of loss over the epochs looks like a linear function that does not converge to 0 as the number of epochs increases. This makes it very sensitive to the hyper-parameters in the SGD. Even marginal changes to the learning rate, batch size, number of epochs have profound impact on the accuracy of the model. all of the models perform better in the parts of the dataset that have more data. For example, when comparing the training performance between the first 1000 elements of the dataset and the lest 1000 elements of the dataset, it was evident that the model performs better with the later years as there is more data for each of them as can be seen from part 2. Conversely, the models suffer the largest errors for those years that have less data in the dataset. I also believe that the regularization did help all of the models as we can see that in all three graphs both train and test loss functions go 'hand-in-hand'


Ridge Regression weights
Counts
Weights
0
20
40
60
80
0
250
500
750
1000
1250
1500
1750
2000


Lasso Regression weights
Counts
Weights
0
20
40
60
80
0
250
500
750
1000
1250
1500
1750
2000


Count Regression weights
Frequency
Weights
0
20
40
60
80
0.0
0.2
0.4
0.6
0.8
1.0

Problem 5 - Softmax Properties

Part 1 (7 points)

We want to show that the Softmax function is invariant to constant offsets to its input. To do that, we can expand the left-hand side (LHS) of the equation

$$\operatorname{softmax}(\mathbf{a} + c\mathbf{1}) = \operatorname{softmax}(\mathbf{a}),$$

and then simplify it to the form of the right-hand side (RHS).

Thus, first, we expand the LHS by adding the $c\mathbf{1}$ term:

$$\operatorname{softmax}(\mathbf{a} + c\mathbf{1}) = \frac{\exp(\mathbf{a} + c\mathbf{1})}{\sum_{j=1}^{K} \exp(a_j + c_j 1)},$$

Next, we can further expand the fraction by expressing the $c\mathbf{1}$ parts as separate factors according to the properties of the exponent and summation:

$$\frac{\exp(\mathbf{a} + c\mathbf{1})}{\sum_{j=1}^{K} \exp(a_j + c_j 1)} = \frac{\exp(\mathbf{a}) \cdot \exp(c\mathbf{1})}{\sum_{j=1}^{K} \exp(a_j) \cdot \exp(c_j \mathbf{1})}$$

Now, we can bring the $\exp(c_j \mathbf{1})$ term outside of the summation to cancel out with the same term in the numerator:

$$\frac{\exp(\mathbf{a}) \cdot \exp(c\mathbf{1})}{\sum_{j=1}^{K} \exp(a_j) \cdot \exp(c_j \mathbf{1})} = \frac{\exp(\mathbf{a}) \cdot \exp(c\mathbf{1})}{\exp(c\mathbf{1}) \cdot \sum_{j=1}^{K} \exp(a_j)}$$

Finally, as mentioned earlier, the $\exp(c\mathbf{1})$ terms cancel out and we are left with the following fraction with is exactly the $\operatorname{softmax}(a)$:

$$\frac{\exp(\mathbf{a})}{\sum_{j=1}^{K} \exp(a_j)} = \operatorname{softmax}(a)$$

This validates the equation $\operatorname{softmax}(\mathbf{a} + c\mathbf{1}) = \operatorname{softmax}(\mathbf{a})$ the property that the softmax function is invariant to constant offsets to its input.

Part 2 (3 points)

In practice, this property comes in handy when there is a concern that the input is skewed by constant offsets. The function's ability to assign the probabilities to each label so that they all add up to 1 is not impacted by that, which makes it very useful as an activation function for a neural network.

Problem 6 - Implementing a Softmax Classifier

Similar to the Music Dataset I implemented the loadIris() and normalize_Iris() functions. The only difference this time was that I decided not to add the bias term in the loadIris() function, but instead add it in the normalize_Iris() after I performed the normalization. The following is the implementation of the loadIris():

```
def loadIris():
    trIris = np.loadtxt('iris-train.txt')
    tsIris = np.loadtxt('iris-test.txt')

    trIrisX = trIris[: , 1:]
    trIrisy = trIris[: , 0 ]
    tsIrisX = tsIris[: , 1:]
    tsIrisy = tsIris[: , 0 ]

    return trIrisX, trIrisy, tsIrisX, tsIrisy
```

The load_Iris() function is different from the normalize_MSD() because the objective is to transform the features into values between -1 and 1, not 0 and 1. This is why the normalization method used here is the min-max normalization:

```
def normalize_Iris(trainX, testX):

    minX = np.min(trainX, axis=0)
    maxX = np.max(trainX, axis=0)

    trainX_normalized = (((trainX - minX) / (maxX - minX)) * 2) - 1
    testX_normalized = (((testX - minX) / (maxX - minX)) * 2) - 1

    trainX_normalized = np.hstack((trainX_normalized, np.ones((trainX_normalized.
        shape[0], 1)))) #add bias
    testX_normalized = np.hstack((testX_normalized, np.ones((testX_normalized.
        shape[0], 1)))) # add bias

    return trainX_normalized, testX_normalized
```

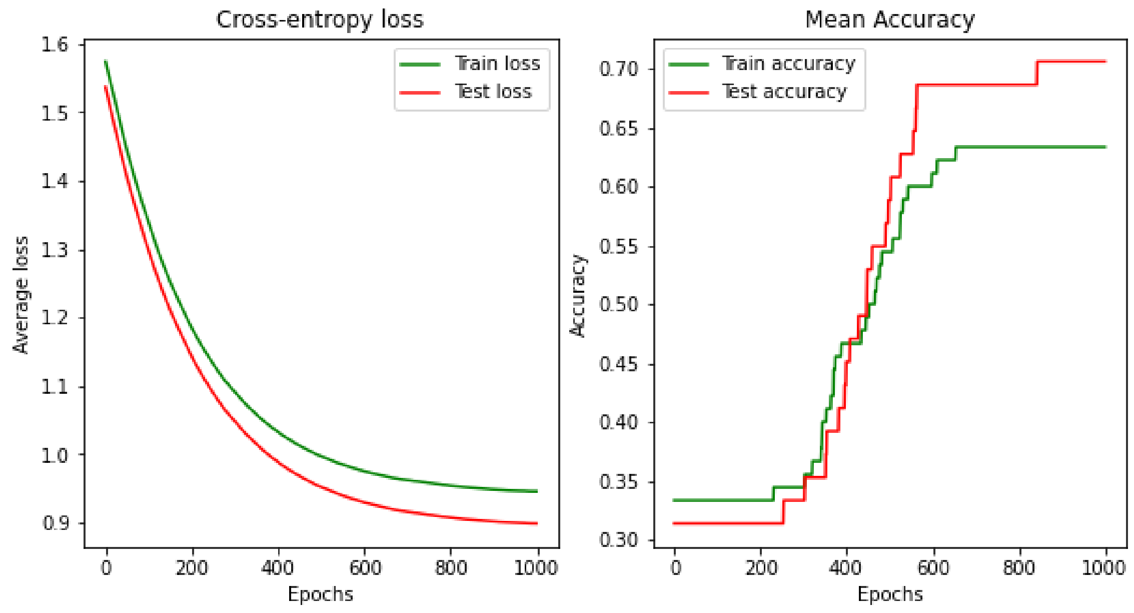
Part 1 - Implementation & Evaluation (20 points)

I implemented the Softmax Classifier in a separate function from the minibatchSGD() but reused a great deal of code, which made the process more efficient. Among the changes that I made were removing the loss type and weight decay type parameters from the function definition as well as adding the momentum rate. Please, refer to the notebook for the implementation.

After tuning the hyper-parameters, it appears that the model performs the best with a weight decay factor of 0.01, a momentum rate of 0.2, learning rate of 0.02, and a batch size of 64. In the function call to minibatch_SGD_softmax() I perform an elementwise subtraction of 1 to avoid index error when calculating loss:

```
softmax_weights = minibatch_SGD_softmax(n_trainIris, trainIrisy-1, n_testIris
    , testIrisy-1, weight_decay_factor=0.01, momentum_rate=0.2, learning_rate
    =0.02, batch_size=64, n_epochs = 1000)
```

The following image shows the two plots requested by the assignment prompt. The plot of the left shows the cross-entropy loss as a function of epochs, while the plot on the right shows the mean accuracy for both sets as a function of epochs:



With the mentioned hyperparameters and the performance metrics shown in the graphs above, the model was able to achieve a 70.59% test accuracy.

Running the model multiple times produced different results. Sometimes, the accuracy would decrease below the best value seen throughout all 1000 epochs, which suggests that early stopping could help improve accuracy. However, to account for the early stopping problem, I recorded the weights vector every time that the model fins a new best accuracy. Later, instead of returning the latest version of the weights vector, I return the best one.

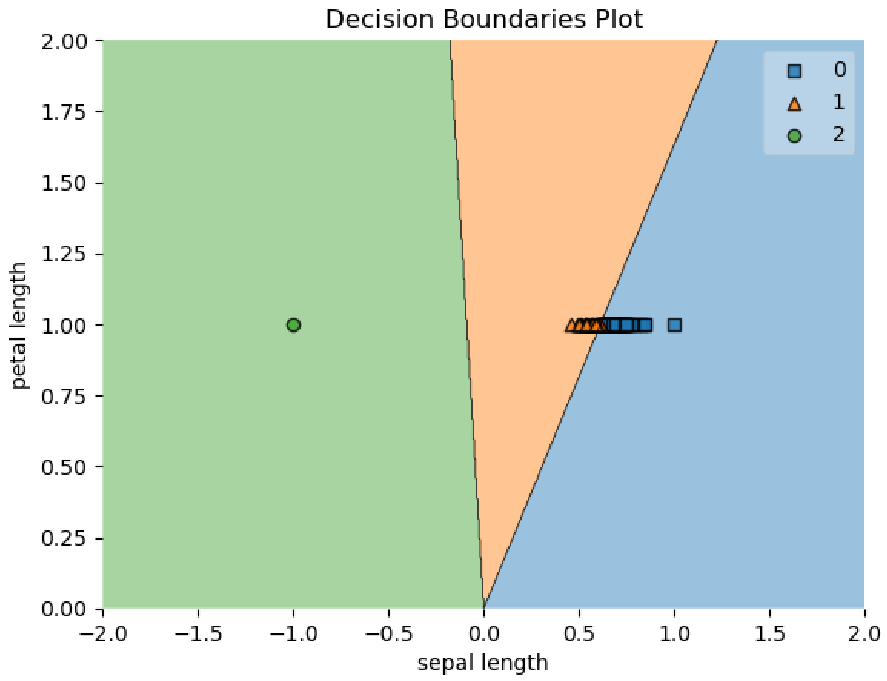
Part 2 - Displaying Decision Boundaries (10 points)

I implemented the region boundaries plotting using the mlxtend library. I had to splice the label and feature sets before passing into the plotting method in order to remove the bias terms. The mlxtend plot_decision_regions() function requires a model object with a defined predict() method inside. I Implemented the class to require the weights produced by the softmax SGD at initialization. Following is the class implementation:

```python
class SoftmaxClassifier():
    def __init__(self, weights):
        self.weights = weights

    def predict(self, X):
        y_pred = X @ self.weights[:, [0,2]].T
        return np.argmax(y_pred, axis=1)
```

Next, I preprocess the data and feed it to the plot_decision_regions() along with the classifier. The following are the code implementation and the resulting plot:

```python
from mlxtend.plotting import plot_decision_regions
classifier = SoftmaxClassifier(softmax_weights)
X = n_trainIris[:, [0, 2]]
preds = classifier.predict(X)

plot_decision_regions(X, preds, clf=classifier,)
plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.title('Decision Boundaries Plot')
```



Problem 7 - Classifying Images (10 points)

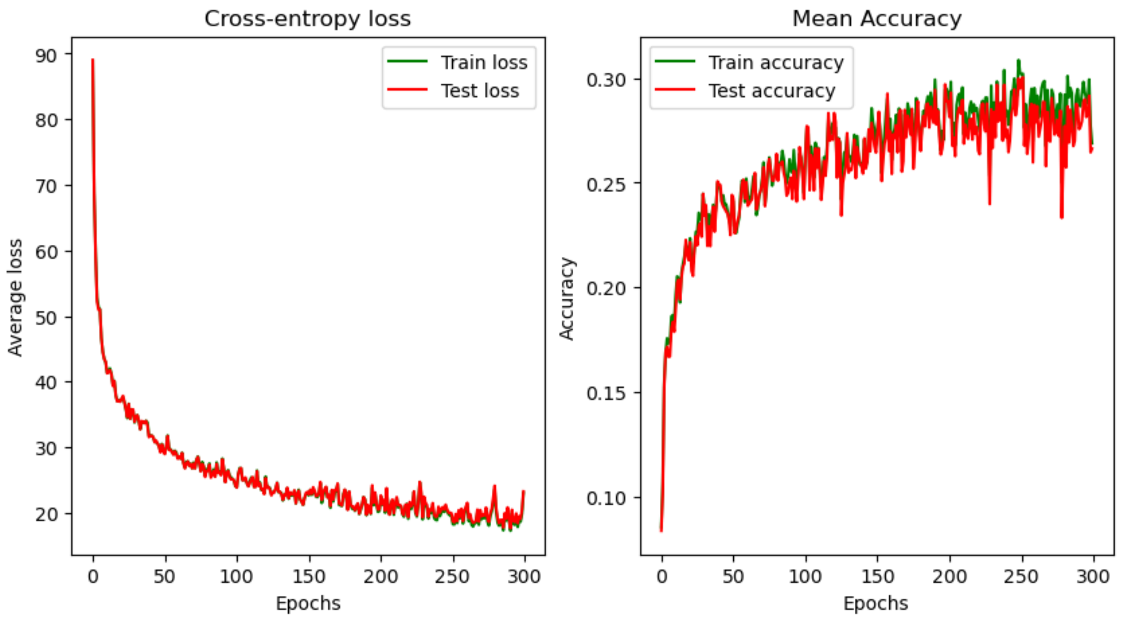
I load the data set using Keras. Next, before I can use my Softmax classifier, the data set has to be normalized. I perform standard normalization by computing the mean and standard deviation of the training data and subtracting and dividing the train and test sets. Here is the implementation:

```
def normalizeCIFAR(Ctrain, Ctest):
    feature_n = Ctrain.shape[1] * Ctrain.shape[2] * Ctrain.shape[3] #height *
        width * n_channels
    Ctrain_flat = Ctrain.reshape(Ctrain.shape[0], feature_n)
    Ctest_flat = Ctest.reshape(Ctest.shape[0], feature_n)

    mean = np.mean(Ctrain_flat, axis=0)
    std = np.sqrt(np.sum(np.square(Ctrain_flat - mean), axis=0) / (Ctrain.shape
        [0] - 1))

    nCtrain = (Ctrain_flat - mean)/std
    nCtest = (Ctest_flat - mean)/std

    return nCtrain, nCtest
```

I then use the normalized feature sets to run Softmax classifier:

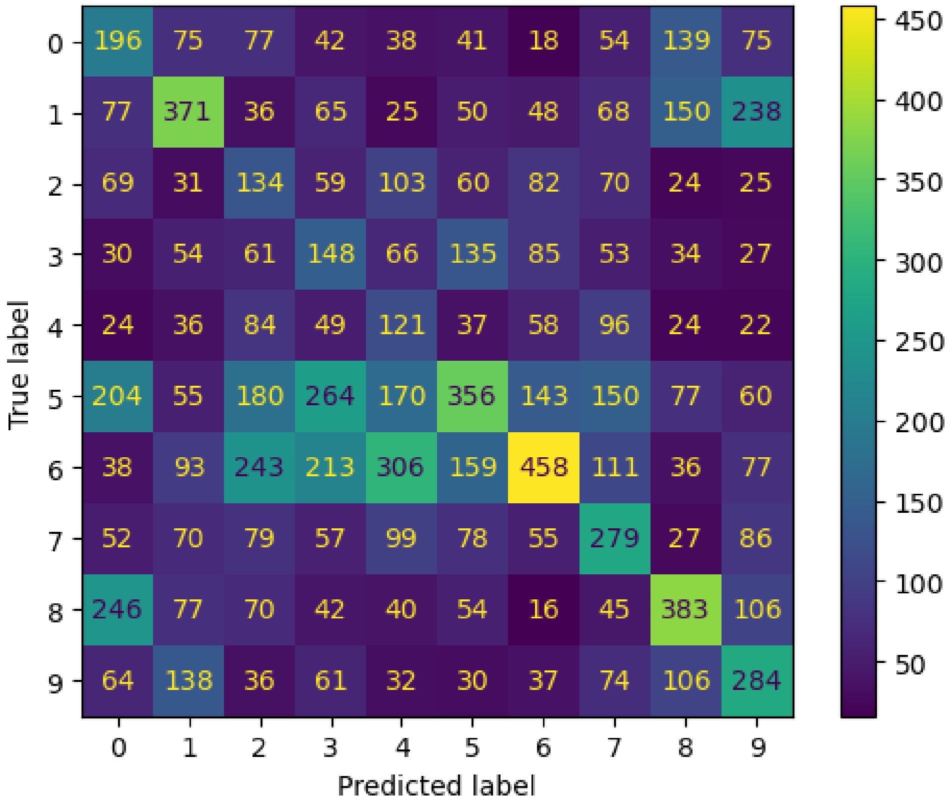
```
cifar_weights = minibatch_SGD_softmax(norm_cifar_trX, cifar10_trainy.flatten(),
    norm_cifar_tsX, cifar10_testy.flatten(), learning_rate=0.3, momentum_rate=0.2,
     weight_decay_factor=0.01, batch_size=150, n_epochs=300)
```

The hyper-parameters used in the classification above produced the best accuracy of about 30% and the following are the loss and average accuracy graphs:



After this, I used the output weights from the softmax classifier to make predictions on the test partition and built the confusion matrix:

```
cifar_preds = norm_cifar_tsX @ cifar_weights.T
cifar_label_preds = np.argmax(cifar_preds, axis = 1)
conf_m = confusion_matrix(cifar_label_preds, cifar10_testy, labels=[0, 1, 2, 3,
    4, 5, 6, 7, 8, 9])
conf_m_display = ConfusionMatrixDisplay(confusion_matrix=conf_m, display_labels
    =[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
conf_m_display.plot()
```


True label
Predicted label
0 1 2 3 4 5 6 7 8 9
196 75 77 42 38 41 18 54 139 75
77 371 36 65 25 50 48 68 150 238
69 31 134 59 103 60 82 70 24 25
30 54 61 148 66 135 85 53 34 27
24 36 84 49 121 37 58 96 24 22
204 55 180 264 170 356 143 150 77 60
38 93 243 213 306 159 458 111 36 77
52 70 79 57 99 78 55 279 27 86
246 77 70 42 40 54 16 45 383 106
64 138 36 61 32 30 37 74 106 284
450
400
350
300
250
200
150
100
50

Works Cited

http://rasbt.github.io/mlxtend/api$_subpackages/mlxtend.plotting/plot_decision_regions$

http://rasbt.github.io/mlxtend/user$_guide/plotting/plot_decision_regions/$

https://medium.com/lcc-unison/how-to-poisson-regression-model-python-implementation-1c672582eb96

https://stackoverflow.com/questions/29831489/convert-array-of-indices-to-one-hot-encoded-array-in-numpy

https://p61402.github.io/2018/12/05/CS224n-assignment-1/

Debugging and looking for loss gradients

Softmax Classifier Code Appendix

Please, see the notebook attached in the same .zip