CS 5787 Deep Learning, Spring 2020
Homework 1
Ananya Shivaditya (aks298)

Due: See Canvas

Instructions

Your homework submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Canvas.

Your homework solution must be typed. We urge you to prepare it in LaTeX. It must be output to PDF format. To use LaTeX, we suggest using `http://overleaf.com`, which is free and can be accessed online.

Your programs must be written in Python. The relevant code to the problem should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution to that problem. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}. For this assignment, you may not use a neural network toolbox. **The algorithm should be implemented using only NumPy.**

If you have forgotten your linear algebra, you may find *The Matrix Cookbook* useful, which can be readily found online. You may wish to use the program *MathType*, which can easily export equations to AMS LaTeX so that you don't have to write the equations in LaTeX directly: `http://www.dessci.com/en/products/mathtype/`

If told to implement an algorithm, don't use a toolbox, or you will receive no credit.

Problem 1 - Softmax Properties

Part 1 (7 points)

Recall the softmax function, which is the most common activation function used for the output of a neural network trained to do classification. In a vectorized form, it is given by

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

where $\mathbf{a} \in \mathbb{R}^K$. The exp function in the numerator is applied element-wise and a_j denotes the j'th element of $\mathbf{a}$.

Show that the softmax function is invariant to constant offsets to its input, i.e.,

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

where $c \in \mathbb{R}$ is some constant and $\mathbf{1}$ denotes a column vector of 1's.

Solution:
For all dimensions $1 \leq i \leq dim(a)$:

$$\begin{aligned}
&\{\text{softmax}(\mathbf{a} + c\mathbf{1})\}_i \\
&= \frac{\exp(a_i + c)}{\sum_{j=1}^{K} \exp(a_j + c)} \\
&= \frac{\exp(a_i)\exp(c)}{\sum_{j=1}^{K} \exp(a_j)\exp(c)} \\
&= \frac{\exp(a)\cancel{\exp(c)}}{(\cancel{\exp(c)})\sum_{j=1}^{K} \exp(a_j)} \\
&= \frac{\exp(a)}{\sum_{j=1}^{K} \exp(a_j)} \\
&= \{\text{softmax}(a)\}_i
\end{aligned}$$

So each element at the i^{th} position in softmax $(\mathbf{a})$, is equal to the corresponding i^{th} element in softmax $(\mathbf{a} + c\mathbf{1})$. Therefore, softmax $(\mathbf{a} + c\mathbf{1}) =$ softmax $(\mathbf{a})$

Part 2 (3 points)

In practice, why is the observation that the softmax function is invariant to constant offsets to its input important when implementing it in a neural network?

Solution:
The softmax function reacts to low stimulation (i.e small differences in its input) with rather uniform distribution and to high stimulation (i.e. larger differences in its input) with probabilities close to 0 and 1. In case cross entropy loss is used (usually in classification problems) - the exp in the softmax function roughly cancels out the log in the cross-entropy loss causing the loss to be roughly linear in k_i. This leads to a roughly constant gradient, when the model is wrong, allowing it to correct itself quickly. Thus, a wrong saturated softmax does not cause a vanishing gradient, unlike some other activation functions. This serves well for practical implementation purposes.

Problem 2 - Implementing a Softmax Classifier

For this problem, you will use the 2-dimensional Iris dataset. Download `iris-train.txt` and `iris-test.txt` from Canvas. Each row is one data instance. The first column is the label (1, 2 or 3) and the next two columns are features.

Write a function to load the data and the labels, which are returned as NumPy arrays.

Part 1 - Implementation & Evaluation (20 points)

Recall that a softmax classifier is a shallow one-layer neural network of the form:

$$P\left(C=k|\mathbf{x}\right)=\frac{\exp\left(\mathbf{w}_k^T\mathbf{x}\right)}{\sum_{j=1}^{K}\exp\left(\mathbf{w}_j^T\mathbf{x}\right)}$$

where $\mathbf{x}$ is the vector of inputs, K is the total number of categories, and $\mathbf{w}_k$ is the weight vector for category k.

In this problem you will implement a softmax classifier from scratch. **Do not use a toolbox.** Use the softmax (cross-entropy) loss with L_2 weight decay regularization. Your implementation should use stochastic gradient descent with mini-batches and momentum to minimize softmax (cross-entropy) loss of this single layer neural network. To make your implementation fast, do as much as possible using matrix and vector operations. This will allow your code to use your environment's BLAS. Your code should loop over epochs and mini-batches, but do not iterate over individual elements of vectors and matrices. Try to make your code as fast as possible. I suggest using profiling and timing tools to do this.

Train your classifier on the Iris dataset for 1000 epochs. You should either subtract the mean of the training features from the train and test data or normalize the features to be between -1 and 1 (instead of 0 and 1). Hand tune the hyperparameters (i.e., learning rate, mini-batch size, momentum rate, and L_2 weight decay factor) to achieve the best possible training accuracy. During a training epoch, your code should compute the mean per-class accuracy for the training data and the loss. After each epoch, compute the mean per-class accuracy for the testing data and the loss as well. **The test data should not be used for updating the weights.**

After you have tuned the hyperparameters, generate two plots next to each other. The one on the left should show the cross-entropy loss during training for both the train and test sets as a function of the number of training epochs. The plot on the right should show the mean per-class accuracy as a function of the number of training epochs on both the train set and the test set.

What is the best test accuracy your model achieved? What hyperparameters did you use? Would early stopping have helped improve accuracy on the test data?

Solution:

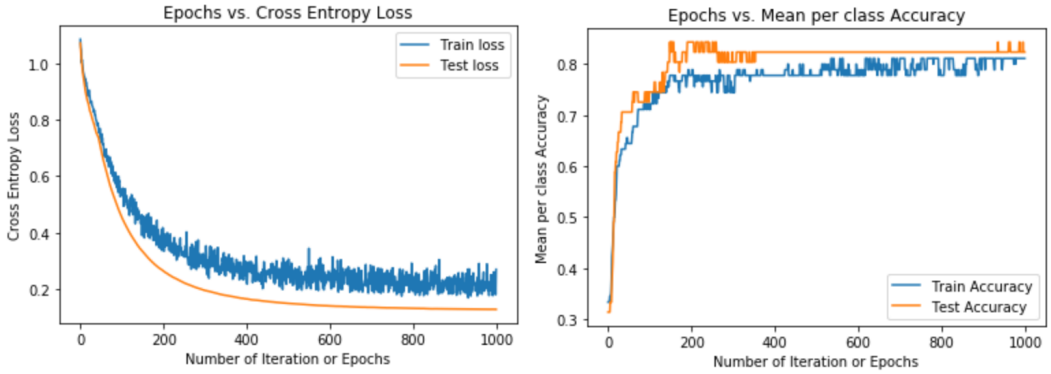
Function to load the data and the labels, which are returned as NumPy arrays:

```
[ ]  def readData(filename):
         dataMatrix = np.loadtxt(filename)
         np.random.shuffle(dataMatrix)
         X = dataMatrix[:, 1:]
         y = dataMatrix[:, 0].astype(int)
         y = y.reshape((-1, 1))
         y -= 1
         return X, y
```

Implemented softmax classifier from scratch: Appendix: [A]

The classifier was trained on the Iris dataset, after normalization.
Hand tuned hyperparameters for best possible training accuracy:
epochs = 1000
learning rate = 0.07
mini-batch size = 10
L_2 weight decay factor = 0.001
momentum rate= 0.05
During a training epoch, the mean per-class accuracy for the training data was calculated, as well as the loss. After each epoch, the mean per-class accuracy was calculated for the testing data, and the loss as well. They are shown on the next page.

Plots of loss and accuracy:



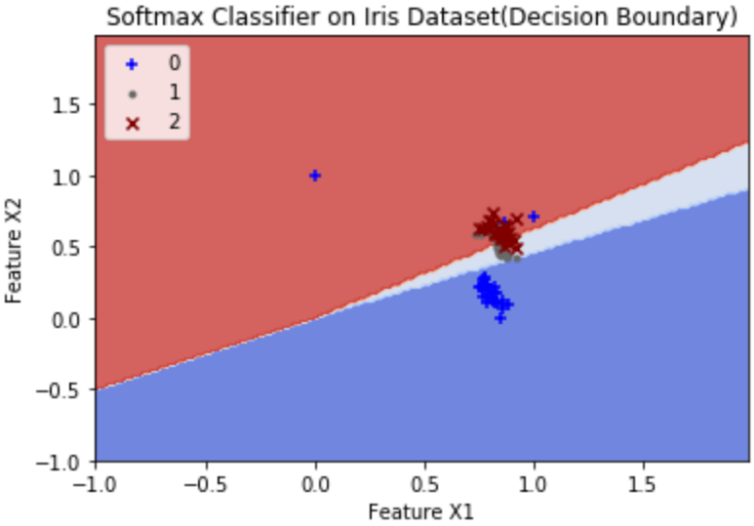
The best test accuracy achieved was 84.31373 %
The hyperparameters used are given on the page above. Early stopping would not have helped improve accuracy on the test data in this case.

Part 2 - Displaying Decision Boundaries (10 points)

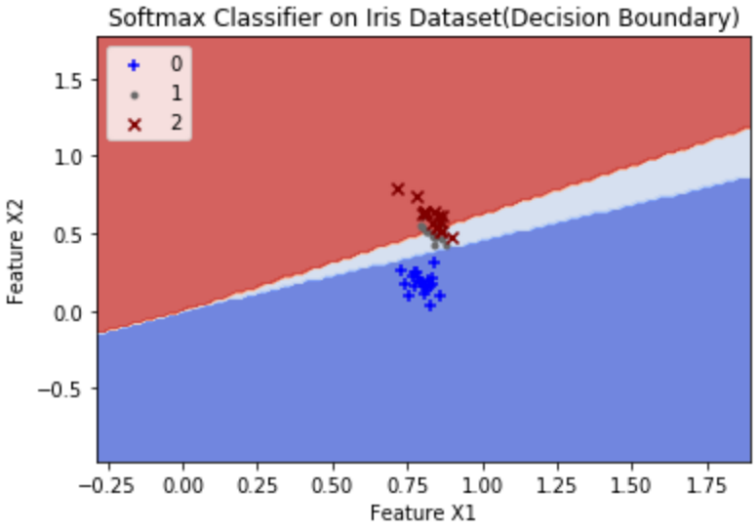
Plot the decision boundaries learned by softmax classifier on the Iris dataset, just like we saw in class. On top of the decision boundaries, generate a scatter plot of the training data. Make sure to label the categories.

Solution:

For training data:



For testing data:

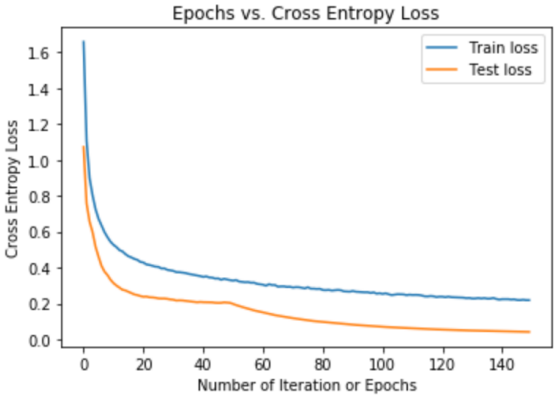


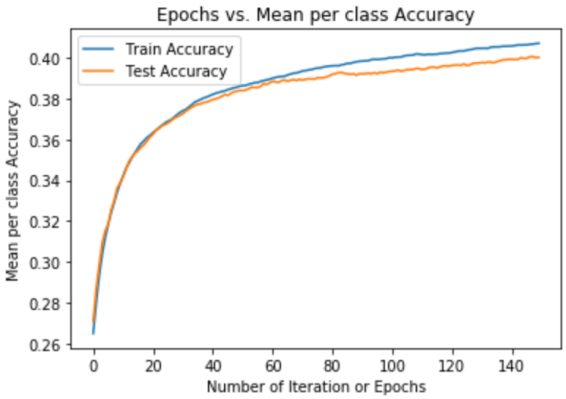
Problem 3 - Classifying Images (10 points)

Recall the CIFAR-10 dataset from Homework 0. Using the softmax classifier you implemented, train the model on CIFAR-10's training partitions. To do this, you will need to treat each image as a vector. You will need to tweak the hyperparmaters you used earlier.

Plot the training loss as a function of training epochs. Try to minimize the error as much as possible. What were the best hyperparmeters? Output the final test accuracy and a normalized 10×10 confusion matrix computed on the test partition. Make sure to label the columns and rows of the confusion matrix.

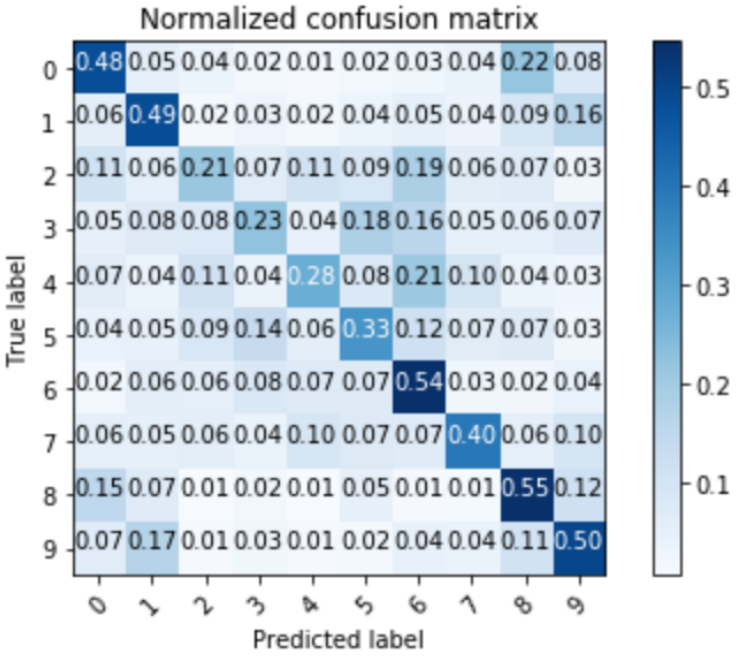
Solution: CIFAR-10 Plot of losses and accuracy as a function of epochs.





Error was minimised with the following as the best hyperparameters:
epochs = 150
learningRate = 0.0007
batchSize = 200
regStrength = 0.001
momentum = 0.005

Final test accuracy is tabulated below:



Problem 4 - Regression with Shallow Nets

Tastes in music have gradually changed over the years, and our goal is to predict the year of a song based on its timbre summary features. This dataset is from the 2011 Million Song Challenge dataset: `https://labrosa.ee.columbia.edu/millionsong/`

We wish to build a linear model that predicts the year. Given an input $\mathbf{x} \in \mathbb{R}^{90}$, we want to find parameters for a model $\hat{y} = \text{round}\left(f\left(\mathbf{x}\right)\right)$ that predicts the year, where $\hat{y} \in \mathbb{Z}$.

We are going to explore three shallow (linear) neural network models with different activation functions for this task.

To evaluate the model, you must round the output of your linear neural network. You then compute the mean squared error.

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

Download the music year classification dataset from Canvas, which is located in `music-dataset.txt`. Each row is an instance. The first value is the target to be predicted (a year), and the remaining 90 values in a row are all input features. Split the dataset into train and test partitions by treating the first 463,714 examples as the train set and the last 51,630 examples as the test set. The first 12 dimensions are the average timbre and the remaining 78 are the timbre covariance in the song.

Write a function to load the dataset, e.g.,
`trainYears, trainFeat, testYears, testFeat = loadMusicData(fname, addBias)`
where `trainYears` has the years for the training data, `trainFeat` has the features, etc. `addBias` appends a '1' to your feature vectors. Each of the returned variables should be NumPy arrays.

Write a function `mse = musicMSE(pred, gt)` where the inputs are the predicted year and the 'ground truth' year from the dataset. The function computes the mean squared error (MSE) by rounding `pred` before computing the MSE.

Load the dataset and discuss its properties. What is the range of the variables? How might you normalize them? What years are represented in the dataset?

Generate a histogram of the labels in the train and test set and discuss any years or year ranges that are under/over-represented.

What will the test mean squared error (MSE) be if your classifier always outputs the most common year in the dataset?

What will the test MSE be if your classifier always outputs 1998, the rounded mean of the years?

Solution:

Function to calculate MSE after rounding pred:

```
def musicMSE(pred,gt):
    return ((np.rint(pred) - gt)**2).mean(axis=None)
```

Most common year in the dataset: 2007.
Test MSE if classifier always outputs year 2007
= 190.08239236117836

Test MSE if classifier always outputs 1998 which is the rounded mean of the years
= 118.0090062171951

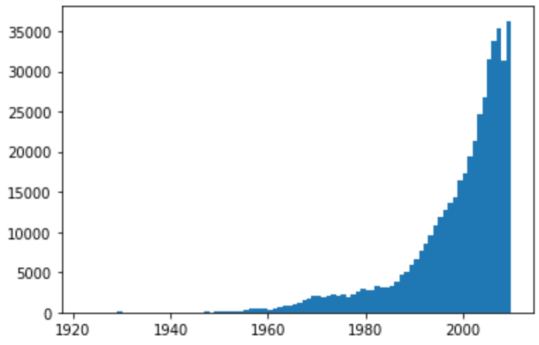
Function to load the dataset given on next page:

```
def loadMusicData(fname, addBias):
    raw = np.loadtxt(fname, delimiter=',')
    X_train = raw[:463714,1:]
    Y_train = raw[:463714,0:1]
    X_test = raw[463714:,1:]
    Y_test = raw[463714:,0:1]
    up = X_train.max(axis=0,keepdims=True)
    down = X_train.min(axis=0,keepdims=True)
    lo = min(Y_train)
    hi = max(Y_train)
    print("Range of Years is from " + str(lo) + " to " +str(hi))
    # normalize
    m = X_train.mean(axis=0)
    v = X_train.std(axis=0)
    normXtrain = (X_train - m)/v
    normXtest = (X_test - m)/v
    normYtrain = Y_train - lo
    normYtest = Y_test - lo
    if addBias:
        normXtrain = np.append(normXtrain,np.ones((len(normXtrain),1)),axis=1)
        normXtest = np.append(normXtest,np.ones((len(normXtest),1)),axis=1)
    return Y_train, Y_test, normXtrain, normYtrain, normXtest, normYtest
```

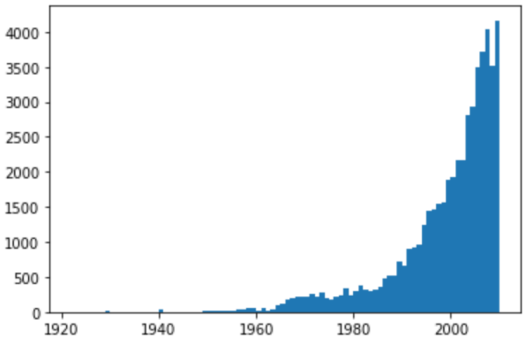
Range of the features from min to max is: variables is: -14861.69535 to 65735.77953
Range of years = 1922 to 2011

Normalization is z-score, as shown above. For X, the mean of each feature column is subtracted from that column, and is divided by the column's standard deviation. For Y, the years are subtracted with the lowest year. If this prediction is treated as classification, we further map each year in the range to a class number. Below is a histogram of the labels in the train and test.

Train



Test



Nearly each year from 1922 to 2011 is represented in the dataset, but the years from 1922 to 1965 are under represented, while years from 1992 to 2011 are over represented.

Part 2 - Ridge Regression (10 points)

Possibly the simplest approach to the problem is linear ridge regression, i.e., $\hat{y} = \mathbf{w}^T\mathbf{x}$, where $\mathbf{x} \in \mathbb{R}^d$ and we assume the bias is integrated by appending a constant to $\mathbf{x}$. The 'ridge' refers to L_2 regularization, which is closely related to L_2 weight decay.

Minimize the loss using gradient descent, just as we did with the softmax classifier to find $\mathbf{w}$. The loss is given by

$$L = \sum_{j=1}^{N} \left\| \mathbf{w}^T\mathbf{x}_j - y_j \right\|_2^2 + \alpha \left\| \mathbf{w} \right\|_2^2 ,$$

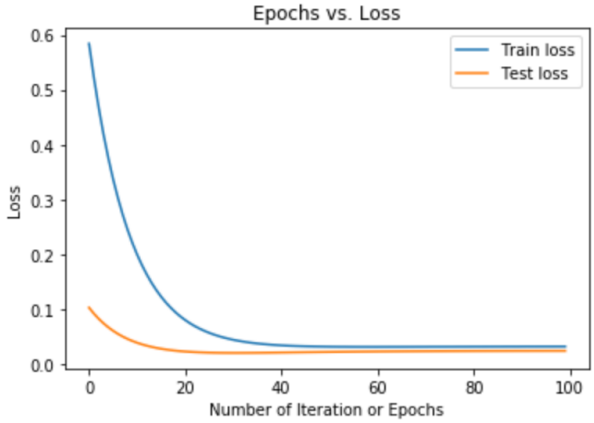
where $\alpha > 0$ is a hyperparameter, N is the total number of samples in the dataset, and y_j is the j-th ground truth year in the dataset. Differentiate the loss with respect to $\mathbf{w}$ to get the gradient descent learning rule and give it here. Use stochastic gradient descent with mini-batches to minimize the loss and evaluate the train and test MSE. Show the train loss as a function of epochs.

As you probably learned in earlier courses, this problem can be solved directly using the pseudoinverse. Compare both solutions.

Tip: Debug your models by using an initial training set that only has about 100 examples and make sure your train loss is going down.

Tip: If you don't use a constant (bias), things will go very bad. If you don't normalize your features by 'z-score' normalization of your data then things will go very badly. This means you should compute the training mean across feature dimensions and the training standard deviation, and then normalize by subtracting the training mean from both the train and test sets, and then divide both sets by the train standard deviation.

Solution:
Implemented Ridge Regression from scratch: Appendix: [B]
Loss as a function of epochs:



Hyperparameters:
epochs = 100
learningRate = 0.0006
batchSize = 10000
regStrength = 0.000008
Final MSEs: Train loss = 92.23977926049245 Test loss = 91.51823516879394

```
# As you probably learned in earlier courses, this problem can be solved directly
# using the pseudoinverse. Compare both solutions.
alpha = 1
n, m = normXtrain.shape
I = np.identity(m)
w = np.dot(np.dot(np.linalg.inv(np.dot(normXtrain.T, normXtrain) + alpha * I),
                  normXtrain.T), normYtrain)
diff = sm.wt - w
diff.mean()
```

```
0.005321821697593038
```

As we see above, the two methods are give very similar results.

Part 3 - L_1 Weight Decay (10 points)

Try modifying the model to incorporate L_1 regularization (L_1 weight decay). The new loss is given by

$$L = \sum_{j=1}^{N} \left\| \mathbf{w}^T \mathbf{x}_j - y_j \right\|_2^2 + \alpha \left\| \mathbf{w} \right\|_1^1 .$$

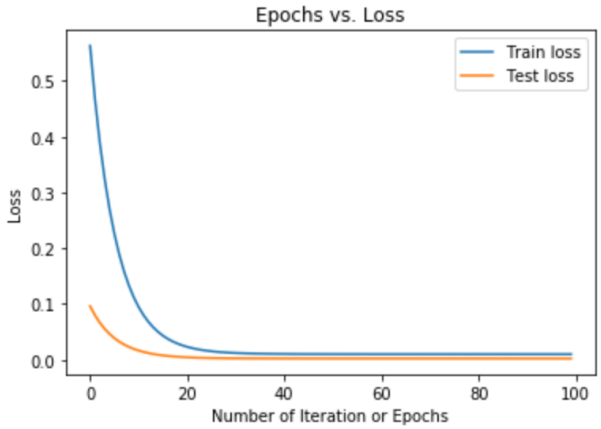
Tune the weight decay performance and discuss results.
Plot a histogram of the weights for the model with L_2 weight decay (ridge regression) compared to the model that uses L_1 weight decay and discuss.

Solution:

Implemented L1 Weight Decay from scratch:
Appendix: [C]

Hyperparameters:
epochs = 100
learningRate = 0.001
batchSize = 10000
regStrength = 0.000008

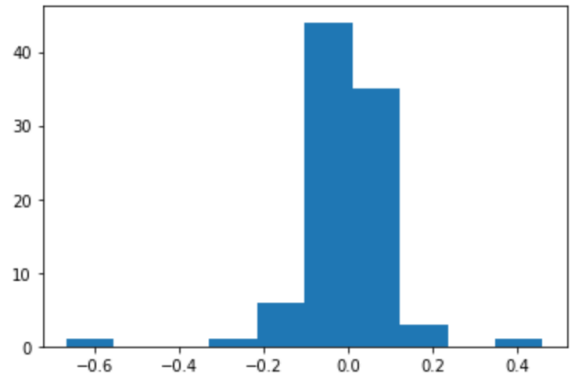
Loss as a function of epochs:



Below is a histogram of the weights for the model with L_2 weight decay (ridge regression) compared to the model that uses L_1 weight decay. We see that the weights are very similar, since most of the differences in weight are nearly 0.

```
plt.hist((sm.wt - sm2.wt))
```

```
(array([ 1.,  0.,  0.,  1.,  6., 44., 35.,  3.,  0.,  1.]),
 array([-0.66463529, -0.55221684, -0.43979839, -0.32737994, -0.21496148,
        -0.10254303,  0.00987542,  0.12229387,  0.23471232,  0.34713077,
         0.45954923]),
 <a list of 10 Patch objects>)
```



Also, final MSEs are nearly equal.

MSE for Ridge

Train loss = 92.23977926049245

Test loss = 91.51823516879394

MSE for L1 Weight Decay

Train loss = 91.5734569152538

Test loss = 90.81193469039918

Part 4 - Poisson (Count) Regression (10 points)

A potentially interesting way to do this problem is to treat it as a counting problem. In this case, the prediction is given by $\hat{y} = \exp\left(\mathbf{w}^T\mathbf{x}\right)$, where we again assume the bias is incorporated using the trick of appending a constant to $\mathbf{x}$.

The loss is given by

$$L = \sum_{j=1}^{N} \left(\exp\left(\mathbf{w}^T\mathbf{x}_j\right) - y_j\mathbf{w}^T\mathbf{x}_j\right),$$

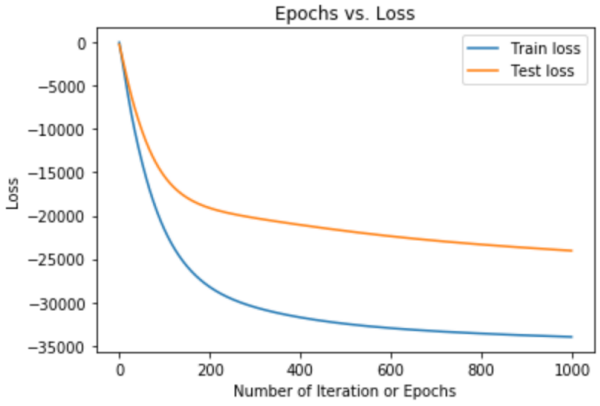
where we have omitted the L_2 regularization term. Minimize it with respect to parameters/weights $\mathbf{w}$ using SGD with mini-batches. Plot the loss. Compute the train and test MSE using the function we created earlier.

Solution:

Implemented Poisson Regression from scratch:
Appendix: [D]

Hyperparameters:
epochs = 1000
learningRate = 0.000001
batchSize = 100

Loss as a function of epochs:



Final MSEs:

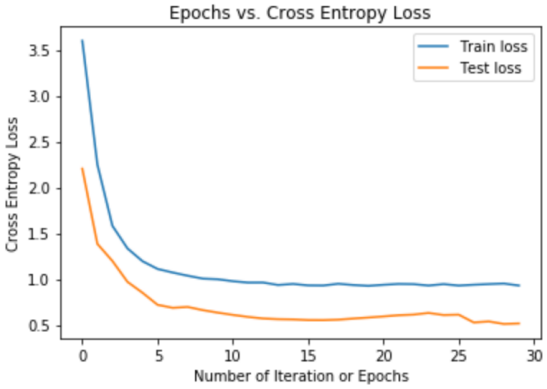
```
[136] mscMSE(normXtest,normYtest)

      6549.3982010213

[138] mscMSE(normXtrain,normYtrain).mean()

      7100.409993837297
```

Part 5 - Classification (5 points)

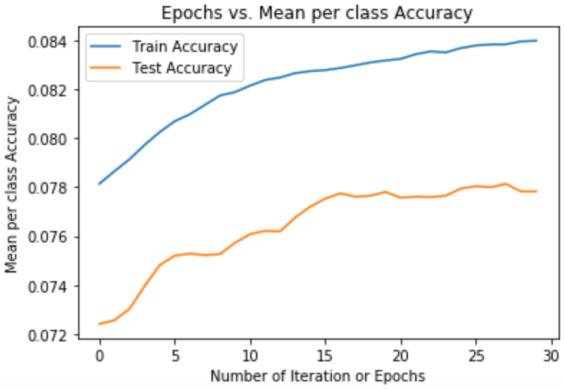
One way to do this problem is to treat it as a classification problem by treating each year as a category. Use your softmax classifier from earlier with this dataset and compute the MSE for the train and test dataset. Discuss the pros and cons of treating this as a classification problem.

Solution:

Loss as a function of epochs:



Accuracy as a function of epochs:



Using a classifier for this task may be an overkill given the continuous range of the output variable, as well as the fact that there would be too many classes. The accuracy is best, but at the cost of compute power. Tabulated below are the train and test losses for this model.

```
print("Final MSEs with Softmax Clasifier")
print("Train loss = "+ str(mscMSE(SMmsc.predict(normXtrain),normYtrain)))
print("Test loss = "+ str(mscMSE(SMmsc.predict(normXtest),normYtest)))
```

```
Final MSEs with Softmax Clasifier
Train loss = [4.16655643]
Test loss = [20.69869632]
```

Part 6 - Model Comparison (10 points)

Discuss and compare the behaviors of the models. Are there certain periods (ranges of years) in which models perform better than others? Where are the largest errors across models. Did L_2 regularization help for some models but not others?

Solution:
The year ranges for which accuracy is high is from 1965 onwards. The Ridge Regressor and L1 weight decay model worked much better than the Poisson model. Softmax classification model worked the best, with the lowest MSE loss.
The largest error across models is near the year ranges that were under represented in the dataset. L_2 Regularization helped the linear models, not the Poisson model.

Softmax Classifier Code Appendix

All code can be found at google colab notebook: https://tinyurl.com/r5rs2d8

Appendix: [**A**] Implemented Softmax Classifier from scratch:

```
class Softmax:
__slots__ = ("epochs", "learningRate", "batchSize", "regStrength", \
"wt", "momentum", "velocity")
def __init__(self, epochs, learningRate, batchSize, regStrength, \
momentum):
    self.epochs = epochs
    self.learningRate = learningRate
    self.batchSize = batchSize
    self.regStrength = regStrength
    self.momentum = momentum
    self.velocity = None
    self.wt = None

def train(self, xTrain, yTrain, xTest, yTest):
    D = xTrain.shape[1]  # dimensionality
    numOfClasses = int(np.max(yTrain))+1 # number of classes
    yTrainEnc = self.oneHotEncoding(yTrain, numOfClasses)
    yTestEnc = self.oneHotEncoding(yTest, numOfClasses)
    self.wt = 0.001 * np.random.rand(D, numOfClasses)
    self.velocity = np.zeros(self.wt.shape)
    trainLosses = []
    testLosses = []
    trainAcc = []
    testAcc = []
    for e in range(self.epochs): # loop over epochs
        trainLoss = self.SGDWithMomentum(xTrain, yTrainEnc)
        testLoss, dw = self.computeLoss(xTest, yTestEnc)
        trainAcc.append(self.meanAccuracy(xTrain, yTrain))
        testAcc.append(self.meanAccuracy(xTest, yTest))
        trainLosses.append(trainLoss)
        testLosses.append(testLoss)
        print("{:d}\t->\tTrainL : {:.7f}\t|\tTestL : {:.7f}\t|\tTrainAcc : \
        {:.7f}\t|\tTestAcc: {:.7f}".format(e, trainLoss, testLoss, \
        trainAcc[-1], testAcc[-1]))
    return trainLosses, testLosses, trainAcc, testAcc
```

```
def SGDWithMomentum(self, x, y):
    losses = []
    randomIndices = random.sample(range(x.shape[0]), x.shape[0])
    x = x[randomIndices]
    y = y[randomIndices]
    for i in range(0, x.shape[0], self.batchSize):
        Xbatch = x[i:i+self.batchSize]
        ybatch = y[i:i+self.batchSize]
        loss, dw = self.computeLoss(Xbatch, ybatch)
        self.velocity = (self.momentum * self.velocity) + (self.learningRate * dw)
        self.wt -= self.velocity
        losses.append(loss)
    return np.sum(losses) / len(losses)

def softmaxEquation(self, scores):
    scores -= np.max(scores)
    prob = (np.exp(scores).T / np.sum(np.exp(scores), axis=1)).T
    return prob

def computeLoss(self, x, yMatrix):
    numOfSamples = x.shape[0]
    scores = np.dot(x, self.wt)
    prob = self.softmaxEquation(scores)
    loss = -np.log(np.max(prob)) * yMatrix
    regLoss = (1/2)*self.regStrength*np.sum(self.wt*self.wt)
    totalLoss = (np.sum(loss) / numOfSamples) + regLoss
    grad = ((-1 / numOfSamples) * np.dot(x.T, (yMatrix - prob))) + \
    (self.regStrength * self.wt)
    return totalLoss, grad

def meanAccuracy(self, x, y):
    predY = self.predict(x)
    predY = predY.reshape((-1, 1))  # convert to column vector
    return np.mean(np.equal(y, predY))

def predict(self, x):
    return np.argmax(x.dot(self.wt), 1)

def oneHotEncoding(self, y, numOfClasses):
    y = np.asarray(y, dtype='int32')
```

```
        if len(y) > 1:
            y = y.reshape(-1)
        if not numOfClasses:
            numOfClasses = np.max(y) + 1
        yMatrix = np.zeros((len(y), numOfClasses))
        yMatrix[np.arange(len(y)), y] = 1
        return yMatrix
```

Appendix: [**D**] Implemented Poisson Regression from scratch:

```
    # Poisson ( /pwsn/), PWA-zAUŃ)
class Pois:
    __slots__ = ("epochs", "learningRate", "batchSize", "wt")
    def __init__(self, epochs, learningRate, batchSize):
        self.epochs = epochs
        self.learningRate = learningRate
        self.batchSize = batchSize
        self.wt = None

    def train(self, xTrain, yTrain, xTest, yTest):
        self.wt = 0.0000001 * np.random.rand(xTrain.shape[1], 1)
        trainLosses = []
        testLosses = []
        for e in range(self.epochs): # loop over epochs
            trainLoss = self.SGD(xTrain, yTrain)
            testLoss, dw = self.computeLoss(xTest, yTest)
            trainLosses.append(trainLoss)
            testLosses.append(testLoss)

            print(e)#
        return trainLosses, testLosses

    def SGD(self, x, y):
        losses = []
        randomIndices = random.sample(range(x.shape[0]), x.shape[0])
        x = x[randomIndices]
        y = y[randomIndices]
        for i in range(0, x.shape[0], self.batchSize):
            Xbatch = x[i:i+self.batchSize]
            ybatch = y[i:i+self.batchSize]
            loss, dw = self.computeLoss(Xbatch, ybatch)
            self.wt -= self.learningRate * dw
```

```
        losses.append(loss)
    return np.sum(losses) / len(losses)

def computeLoss(self, x, y):
    numOfSamples = x.shape[0]
    preds = np.exp(np.dot(x, self.wt))
    totalLoss = (preds - np.dot(y.T,np.log(preds))).sum()/numOfSamples
    grad = np.dot(x.T,preds) - np.dot(x.T,y)
    return totalLoss, grad

def predict(self, x):
    return np.exp(np.dot(x, self.wt))
```

Linear Models for Regression Code Appendix

Appendix: [B] Implemented Ridge Regression from scratch:

```
class Ridge:
__slots__ = ("epochs", "learningRate", "batchSize", "regStrength", "wt")
def __init__(self, epochs, learningRate, batchSize, regStrength):
    self.epochs = epochs
    self.learningRate = learningRate
    self.batchSize = batchSize
    self.regStrength = regStrength
    self.wt = None

def train(self, xTrain, yTrain, xTest, yTest):
    self.wt = 0.001 * np.random.rand(xTrain.shape[1], 1)
    trainLosses = []
    testLosses = []
    for e in range(self.epochs): # loop over epochs
        trainLoss = self.SGDWithMomentum(xTrain, yTrain)
        testLoss, dw = self.computeLoss(xTest, yTest)
        trainLosses.append(trainLoss)
        testLosses.append(testLoss)

        print("{:d}\t->\tTrainL : {:.7f}\t|\tTestL : {:.7f}".format(e, \
        trainLoss, testLoss))
    return trainLosses, testLosses

def SGDWithMomentum(self, x, y):
    losses = []
    randomIndices = random.sample(range(x.shape[0]), x.shape[0])
    x = x[randomIndices]
    y = y[randomIndices]
    for i in range(0, x.shape[0], self.batchSize):
        Xbatch = x[i:i+self.batchSize]
        ybatch = y[i:i+self.batchSize]
        loss, dw = self.computeLoss(Xbatch, ybatch)
        self.wt -= self.learningRate * dw
        losses.append(loss)
    return np.sum(losses) / len(losses)

def computeLoss(self, x, y):
```

```
        numOfSamples = x.shape[0]
        preds = np.dot(x, self.wt)
        loss = musicMSE(preds, y)
        regLoss = (1/2)*self.regStrength*np.sum(self.wt*self.wt)
        totalLoss = (np.sum(loss) / numOfSamples) + regLoss
        grad = ((2 / numOfSamples) * np.dot(x.T, (preds - y))) + \
        (self.regStrength * self.wt)
        return totalLoss, grad

    def predict(self, x):
        return x.dot(self.wt)
```

Appendix: [**C**] Implemented L1 Weight Decay from scratch:

```
    class L1_reg:
    __slots__ = ("epochs", "learningRate", "batchSize", "regStrength", "wt")
    def __init__(self, epochs, learningRate, batchSize, regStrength):
        self.epochs = epochs
        self.learningRate = learningRate
        self.batchSize = batchSize
        self.regStrength = regStrength
        self.wt = None

    def train(self, xTrain, yTrain, xTest, yTest):
        self.wt = 0.001 * np.random.rand(xTrain.shape[1], 1)
        trainLosses = []
        testLosses = []
        for e in range(self.epochs): # loop over epochs
            trainLoss = self.SGDWithMomentum(xTrain, yTrain)
            testLoss, dw = self.computeLoss(xTest, yTest)
            trainLosses.append(trainLoss)
            testLosses.append(testLoss)

            print("{:d}\t->\tTrainL : {:.7f}\t|\tTestL : {:.7f}".format(e, \
            trainLoss, testLoss))
        return trainLosses, testLosses

    def SGDWithMomentum(self, x, y):
        losses = []
        randomIndices = random.sample(range(x.shape[0]), x.shape[0])
        x = x[randomIndices]
```

```
        y = y[randomIndices]
        for i in range(0, x.shape[0], self.batchSize):
            Xbatch = x[i:i+self.batchSize]
            ybatch = y[i:i+self.batchSize]
            loss, dw = self.computeLoss(Xbatch, ybatch)
            self.wt -= self.learningRate * dw
            losses.append(loss)
        return np.sum(losses) / len(losses)

    def computeLoss(self, x, y):
        numOfSamples = x.shape[0]
        preds = np.dot(x, self.wt)
        loss = musicMSE(preds, y)
        regLoss = self.regStrength*np.sum(self.wt)
        totalLoss = (np.sum(loss) / numOfSamples) + regLoss
        grad = ((2 / numOfSamples) * np.dot(x.T, (preds - y))) + (self.regStrength)
        return totalLoss, grad

    def predict(self, x):
        return x.dot(self.wt)
```