Homework 1
CS 5787 Deep Learning
Spring 2020

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

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

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.,

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

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

Solution:

$$\text{softmax}(\mathbf{a})_i = \frac{e^{a_i}}{\sum_{j=1} e^{a_j}} \tag{1}$$

$$\text{softmax}(\mathbf{a} + c)_i = \frac{e^{(a_i + c)}}{\sum_{j=1} e^{(a_j + c)}} \tag{2}$$

$$= \frac{e^c}{e^c} \frac{e^{(a_i)}}{\sum_{j=1} e^{(a_j)}} \tag{3}$$

$$= \text{softmax}(\mathbf{a})_i \tag{4}$$

Since this is true for an arbitrary i, we can see that the linear shift has no affect on the softmax probabilities.

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:
In practice, this observation is important because it means that if the vector has high values, then the exponential will also be very large, leading to potential overflow. Since the softmax function is invariant to constant offsets to its input, however, we can subtract

the maximum value from all the attributes, while still preserving the overall integrity of the data.

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?

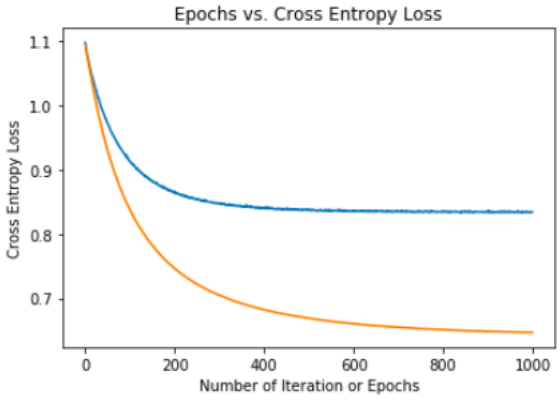
Solution:
HYPERPARAMETERS:

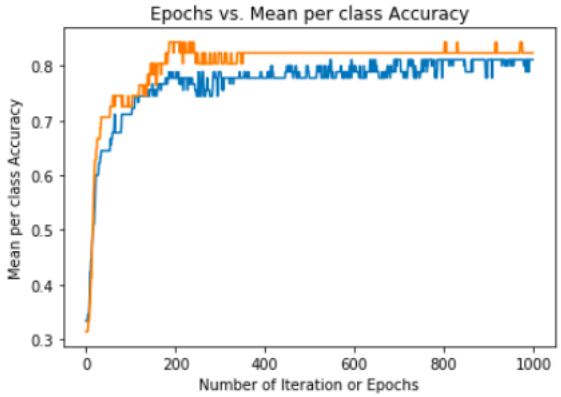
```
Epochs = 1000
Learning Rate = 0.15
Batch size = 10
Regression Strength = 0.001
Momentum = 0.75
```

ACCURACY RESULTS:

```
Final Train Acc:  0.8111111111111111
Final Test Acc:  0.8235294117647058
```

```
Text(0, 0.5, 'Cross Entropy Loss')
```

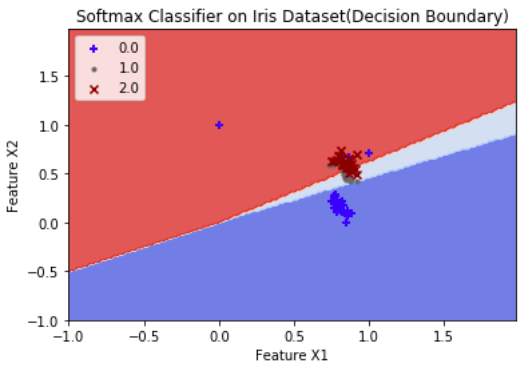


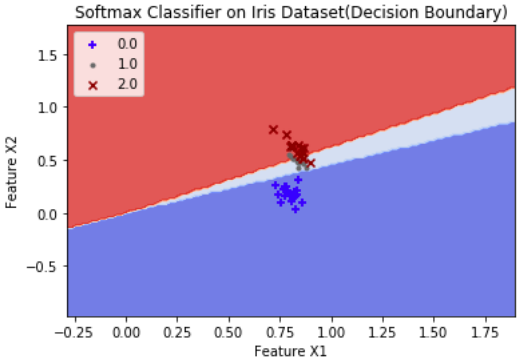


Part 2 - Displaying Decision Boundaries (10 points)

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:





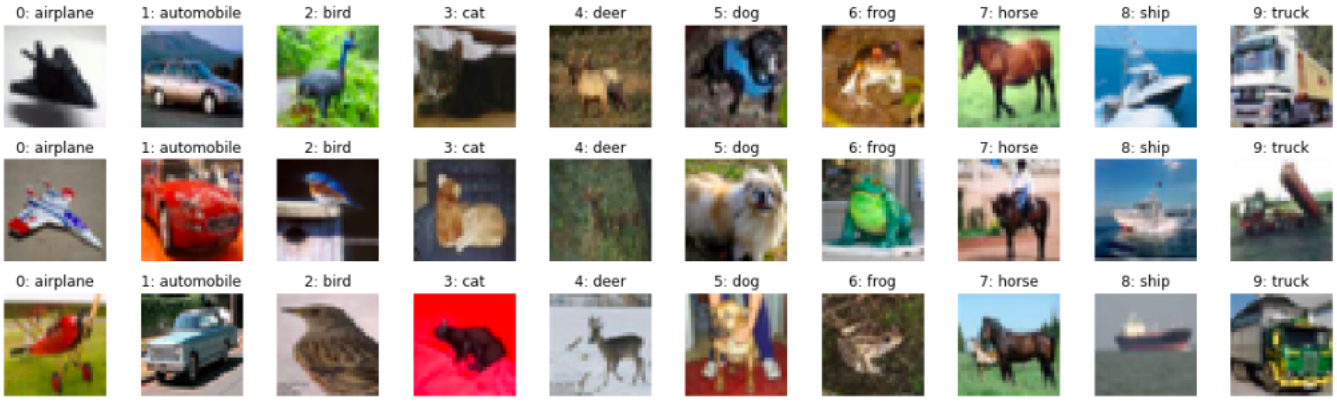
Problem 3 - Visualizing and Loading CIFAR-10 (5 points)

The CIFAR-10 dataset contains 60,000 RGB images from 10 categories. Download it from here: `https://www.cs.toronto.edu/~kriz/cifar.html`
Read the documentation.

Write a function to load the dataset, e.g.,
`trainLabels, trainFeat, testLabels, testFeat = loadCIFAR10()` where `trainLabels` has the categories for the training data, `trainFeat` has the 3072 dimensional image features from the training data, etc. Each of the returned variables should be NumPy arrays.

Using the first CIFAR-10 training batch file, display the first three images from each of the 10 categories as a 3×10 image array. The images are stored as rows, and you will need to reshape them into $32 \times 32 \times 3$ images if you load up the raw data yourself. It is okay to use the PyTorch toolbox for loading them or you can make your own.

Solution:

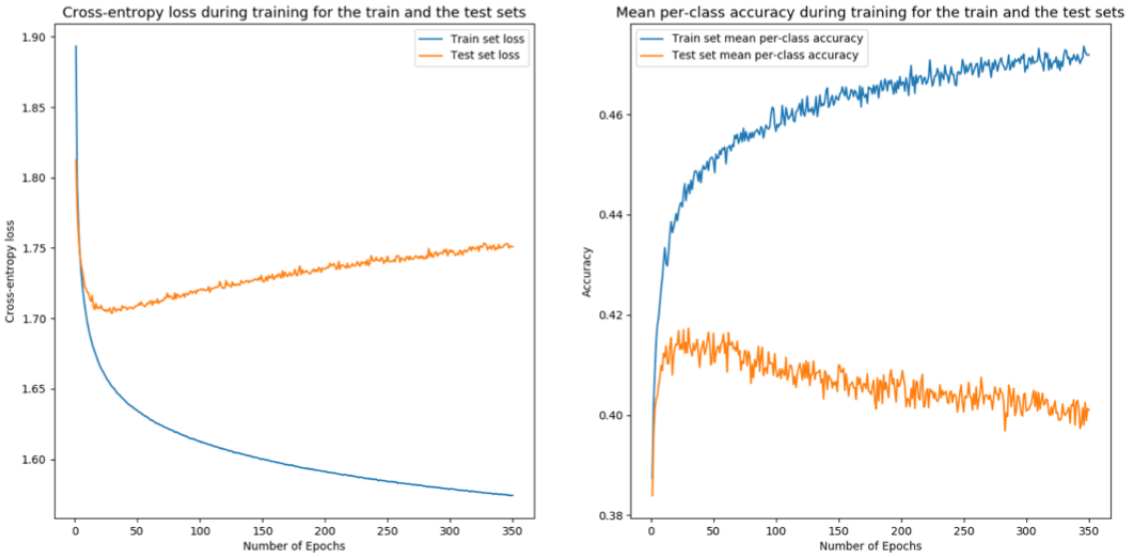


Problem 4 - Classifying Images (10 points)

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:



The best accuracy that the model achieved was 0.30. The hyperparameters used were:

```
learning rate = 0.00005,
num_batches = 32,
regStrength = 0.0001
epochs = 300,
momentum = 0.0001
```

The confusion matrix is attached on the next page. The x-axis represents the predicted label, while the y-axis represents the true label.

```
Normalized confusion matrix
[[0.59  0.01  0.    0.001 0.011 0.035 0.116 0.005 0.169 0.063]
 [0.203 0.059 0.    0.003 0.014 0.044 0.358 0.009 0.132 0.178]
 [0.322 0.008 0.013 0.002 0.032 0.083 0.418 0.017 0.048 0.057]
 [0.231 0.004 0.001 0.01  0.019 0.161 0.418 0.024 0.041 0.091]
 [0.172 0.007 0.006 0.003 0.057 0.1   0.524 0.023 0.042 0.066]
 [0.233 0.001 0.002 0.004 0.024 0.258 0.34  0.026 0.056 0.056]
 [0.172 0.004 0.001 0.    0.007 0.061 0.642 0.028 0.016 0.069]
 [0.22  0.005 0.    0.003 0.055 0.096 0.295 0.05  0.073 0.203]
 [0.283 0.016 0.    0.    0.009 0.069 0.096 0.    0.403 0.124]
 [0.201 0.018 0.001 0.    0.016 0.024 0.177 0.006 0.169 0.388]]
```

Problem 5 - 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:

```
[ ] range_of_variables = np.max(trainYears) - np.min(trainYears)
    print("Min year: ", np.min(trainYears))
    print("Max training year: ", np.max(trainYears))
    print("Range of training years: ", range_of_variables)
```

```
Min year:  1922
Max training year:  2011
Range of training years:  89
```

From the statistics in the picture above, we observe that range of the years in the dataset is approximately 90. The minimum year in the training set is 1922, while the maximum year in the training set is 2011.

```
[ ] print("Min average timbre: ", np.min(trainFeat[:, :12]))
    print("Max average timbre: ", np.max(trainFeat[:, :12]))
    print("Min timbre covariance: ", np.min(trainFeat[:, 12:]))
    print("Max timbre covariance: ", np.max(trainFeat[:, 12:]))
    print("Mode of training years: ", mode(trainYears))
```

```
Min average timbre:  -337.0925
Max average timbre:  384.06573
Min timbre covariance:  -14861.69535
Max timbre covariance:  65735.77953
Mode of training years:  ModeResult(mode=array([2007]), count=array([35374]))
```

The minimum average timbre in the training set is approximately -337, while the maximum average timbre in the training set is approximately 384. The minimum timbre covariance is approximately -14861, while the maximum timbre covariance is around 65735. The

most frequently occurring year is 2007.

```
[ ] print("Test MSE if classifier always outputs the most common year in the dataset (2007): ", musi
    print("Test MSE if classifier always outputs the rounded mean of the years (1998): ", musicMSE(
```

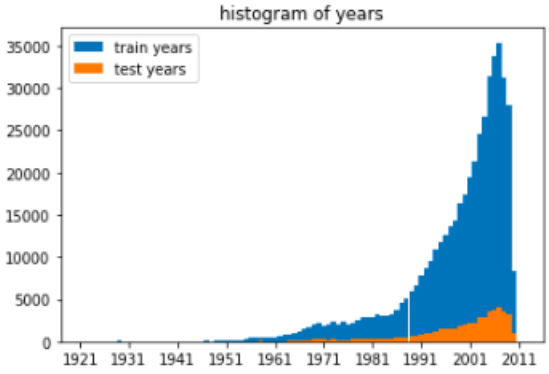
```
Test MSE if classifier always outputs the most common year in the dataset (2007):  190.086073987
Test MSE if classifier always outputs the rounded mean of the years (1998):  118.00972302924656
```

The test MSE if the classifier always outputs the most common year in the dataset (2007) is 190.08607398799148. The test MSE if the classifier always outputs the rounded mean of the years (1998) is 118.00972302924656.

```
[ ] data = np.arange(np.min(trainYears) - 1, np.max(trainYears) + 1) + 0.5
    data2 = np.arange(np.min(trainYears) - 1, np.max(trainYears) + 1, 10)

    plt.hist(trainYears, data)
    plt.hist(testYears, data)
    plt.xticks(data2)
    plt.title('histogram of years')
    plt.legend(["train years", "test years"])
```

```
<matplotlib.legend.Legend at 0x7f3253d25ef0>
```



From the histogram generated above, we can observe that more recent years are generally represented more heavily. The time frame from 1920-1950 appears to have very little data based off the histogram.

We can normalize the data by taking the mean and standard deviation of the different features and then subtracting the mean from the actual value while dividing by the standard deviation. To bound the values even further to be between 1 and -1, we can subtract 1/2 and multiply by 2.

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

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

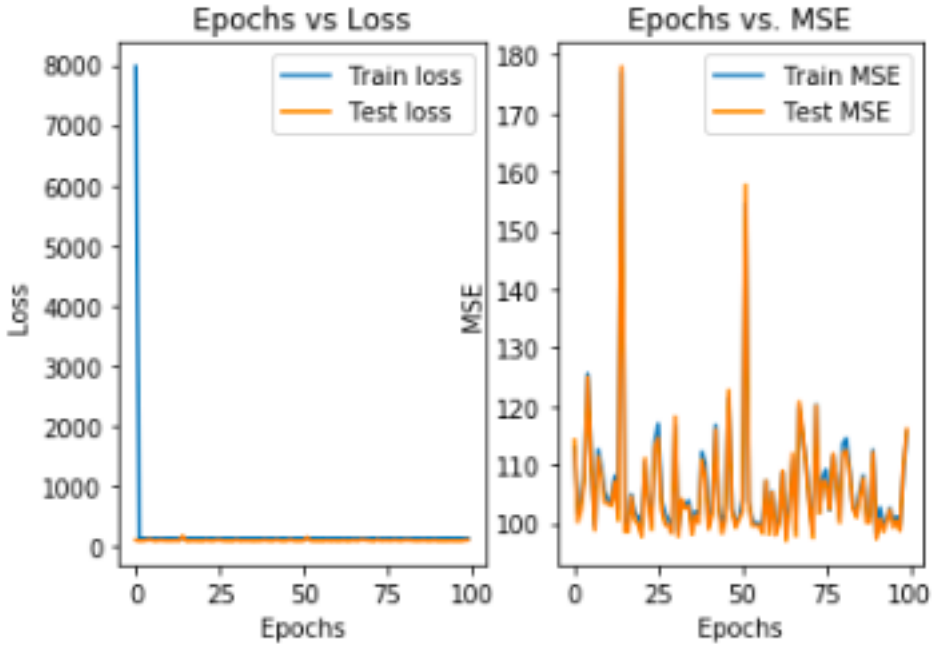
where $\alpha > 0$ is a hyperparameter, N is the total number of samples in the dataset, and y_j is the j-th ground truth year in the dataset. Differentiate the loss with respect to $\mathbf{w}$ to get the gradient descent learning rule and give it here. Use stochastic gradient descent with mini-batches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



Ridge yielded an average MSE of around 120.

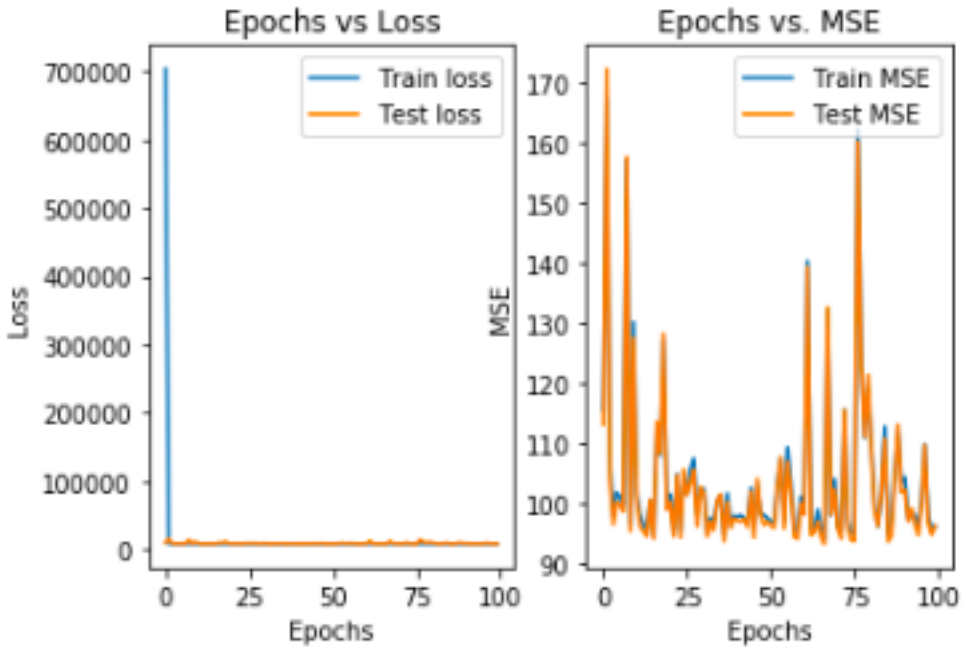
Part 3 - L_1 Weight Decay (10 points)

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:




```
[148] lasso_Regression = Regress(weights=params, epochs=100, lr=0.03, batchsize=100, regType="L
      train_lasso_losses, test_lasso_losses, train_lasso_err, test_lasso_err =
      lasso_Regression.fit(trainFeat, trainYears, testFeat, testYears)

      pred = lasso_Regression.predict(testFeat)
      test_mse = musicMSE(pred, testYears)
      print("Final test MSE for Lasso: ", test_mse)
```

```
Final test MSE for Lasso:  96.14013170637226
```

```
[149] exact_weights = lasso_Regression.pseudoinvSol(testFeat, testYears)
      pred = testFeat @ exact_weights
      actual_mse = musicMSE(pred, testYears)
      print("Analytical solution from pseudoinverse: ", actual_mse)
```

```
Analytical solution from pseudoinverse:  89.96902963393376
```

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

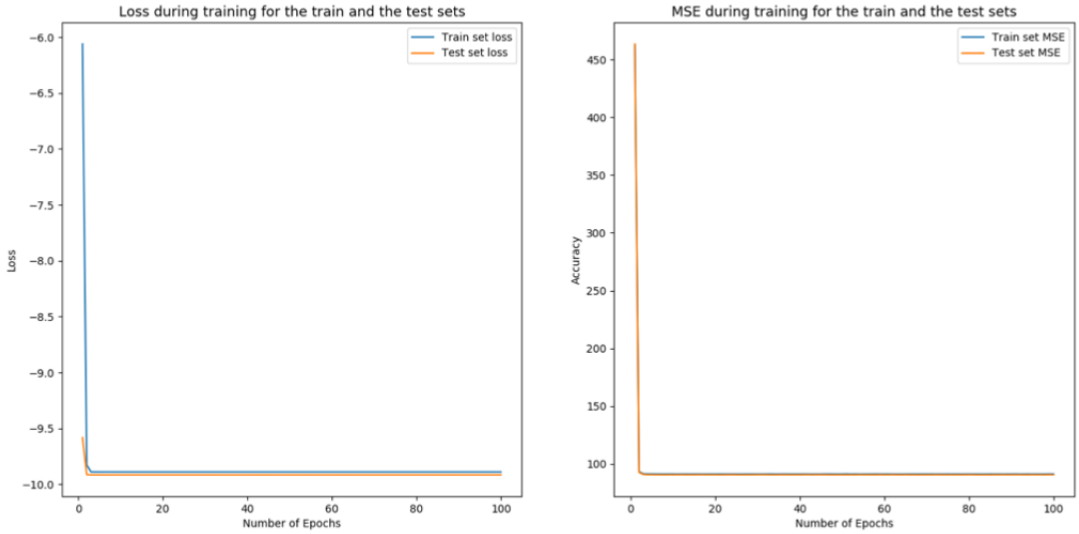
A potentially interesting way to do this problem is to treat it as a counting problem. In this case, the prediction is given by $\hat{y} = \exp\left(\mathbf{w}^T\mathbf{x}\right)$, where we again assume the bias is incorporated using the trick of appending a constant to $\mathbf{x}$.

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:

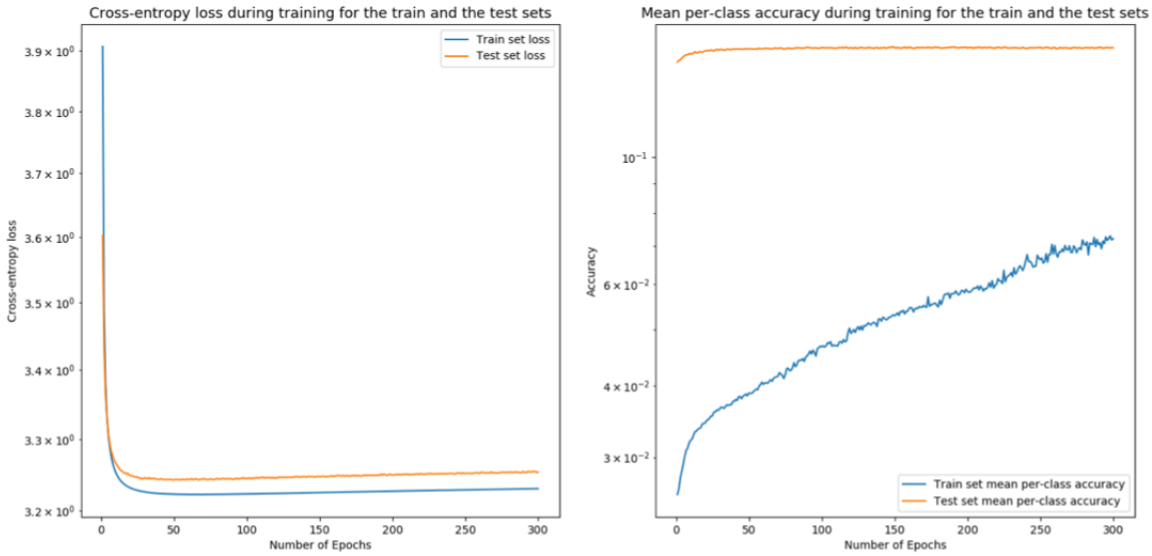


Poisson yielded an MSE of around 90 for both training and testing.

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:



Using the classifier, we achieve an MSE = 122 for training and MSE = 130 for testing.

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 three regression techniques (ridge, lasso, poisson) yielded fairly similar MSE results that almost matched the pseudoinverse solution. However, in order to achieve optimal results, we would want to create a model that accounts for datasets that are unbalanced, like the music dataset we dealt with in this problem. As we saw from the histogram on year representation, years closer to 1920 have very little data, so its hard to make accurate predictions on them. This tends toward the trend that the model is more accurate on recent years than older years. L2 regularization seemed to helped for the linear model instead of softmax.

Softmax Classifier Code Appendix

```
# Code implementation inspired by https://github.com/karan6181/Softmax-Classifier/blob/
master/softmaxClassifier.py

class Softmax:
    def __init__(self, weights, epochs, lr, batchSize, regStrength, momentum, velocity):
        self.weights = weights
        self.epochs = epochs
```

```
    self.lr = lr
    self.batchSize = batchSize
    self.regStrength = regStrength
    self.momentum = momentum
    self.velocity = velocity

def train(self, xTrain, yTrain, xTest, yTest):
    yTrainOneHot = np.asarray(pd.get_dummies(yTrain.reshape(-1)))
    yTestOneHot = np.asarray(pd.get_dummies(yTest.reshape(-1)))
    trainLosses, trainAcc, testLosses, testAcc = list(), list(), list(), list()

    for _ in range(self.epochs):
        trainLoss = self.SGDWithMomentum(xTrain, yTrainOneHot)
        testLoss, _ = self.cross_entropy_loss(xTest, yTestOneHot)
        trainAcc.append(self.meanAccuracy(xTrain, yTrain))
        testAcc.append(self.meanAccuracy(xTest, yTest))
        trainLosses.append(trainLoss)
        testLosses.append(testLoss)

    print("Final Train Acc: ", trainAcc[len(trainAcc) - 1])
    print("Final Test Acc: ", testAcc[len(testAcc) - 1])
    return trainLosses, testLosses, trainAcc, testAcc

def get_minibatches(self, X, y):
  rows = X.shape[0]
  randomIndices = random.sample(range(rows), rows)
  X, y = X[randomIndices], y[randomIndices]
  minibatches = list()

  for i in range (0, rows, self.batchSize):
    X_batch = X[i:i + self.batchSize]
    y_batch = y[i:i + self.batchSize]
    minibatches.append((X_batch,y_batch))
  return minibatches

def SGDWithMomentum(self, x, y):
    losses = list()
    batches = self.get_minibatches(x, y)
    for batch in batches:
      loss, grad = self.cross_entropy_loss(batch[0], batch[1])
      self.velocity = (self.momentum * self.velocity) + (self.lr * grad)
```

```
            self.weights = self.weights - self.velocity
            losses.append(loss)
        return np.sum(losses) / len(losses)

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

    def cross_entropy_loss(self, X, y):
        size = X.shape[0]
        A = X @ self.weights
        prob = self.softmaxEquation(A)

        loss = -np.log((prob)) * y
        weight_Tot = np.sum(self.weights * self.weights)
        regLoss = (1/2) * self.regStrength * weight_Tot

        totalLoss = (np.sum(loss) / size) + regLoss
        grad = self.back_prop(X, y, prob)
        return totalLoss, grad

    def back_prop(self, X, y, prob):
      size = X.shape[0]
      mtx = X.T @ (y - prob)
      gradient = ((-1 / size) * mtx) + (self.regStrength * self.weights)
      return gradient

    def meanAccuracy(self, X, y):
      dot_Prod = X @ self.weights
      predY = np.argmax(dot_Prod, 1).reshape((-1, 1))
      return np.mean(np.equal(y, predY))
```

Linear Models for Regression Code Appendix

```
class Regress:
    def __init__(self, weights=None, epochs=100, lr=0.0001, batchsize=200, regType=""):
        self.weights = weights
```

```
    self.epochs = epochs
    self.lr = lr
    self.batchSize = batchSize
    self.regStrength = 0.001
    self.regType = regType

def train(self, xTrain, yTrain, xTest, yTest):
    trainLosses, trainAcc, testLosses, testAcc = list(), list(), list(), list()

    for _ in range(self.epochs):
        trainLoss = self.SGD(xTrain, yTrain)
        if (self.regType == 'Ridge'):
          test_loss, _ = self.compute_loss_ridge(xTest, yTest)
        elif (self.regType == 'Lasso'):
          test_loss, _ = self.compute_loss_lasso(xTest, yTest)
        elif (self.regType == 'Poisson'):
          test_loss, _ = self.compute_loss_poisson(xTest, yTest)

        train_pred = self.predict(xTrain)
        test_pred = self.predict(xTest)
        trainAcc.append(musicMSE(train_pred, yTrain))
        testAcc.append(musicMSE(test_pred, yTest))
        trainLosses.append(trainLoss)
        testLosses.append(testLoss)

    return trainLosses, testLosses, trainAcc, testAcc

def predict(self, X):
    return X @ self.weights

def SGD(self, X, y):
    ind = random.sample(range(X.shape[0]), X.shape[0])
    X, y = X[ind], y = y[ind]
    num_samples = X.shape[0]
    loss_arr = list()

    for i in range(0, num_samples, self.batchSize):
        batch_X = X[i:i + self.batchSize]
        batch_Y = y[i:i + self.batchSize]

        if (self.regType == 'Ridge'):
```

```
            loss, dw = self.compute_loss_ridge(batch_X, batch_Y)
          elif (self.regType == 'Lasso'):
            loss, dw = self.compute_loss_lasso(batch_X, batch_Y)
          elif (self.regType == 'Poisson'):
            loss, dw = self.compute_loss_poisson(batch_X, batch_Y)

          self.weights = self.weights + dw * self.lr
          loss_arr.append(loss)
      return np.sum(loss_arr)/len(loss_arr)

  def compute_loss_ridge(self, X, y):
      num_samples = X.shape[0]
      A = X @ self.weights
      dot_Weights = self.weights.T @ self.weights
      dot_A = (A - y).T @ (A - y)
      loss = dot_A + self.regStrength * dot_Weights
      grad = (-1 / num_samples) * 2 * X.T @ (A - y) + (2 * self.regStrength * self.weight
      total_loss = np.sum(loss) / num_samples
      return total_loss, grad

  def compute_loss_lasso(self, X, y):
      num_samples = X.shape[0]
      A = X @ self.weights
      dot_A = (A - y).T @ (A - y)
      loss = dot_A + self.regStrength * (abs(self.weights))
      grad = (-1 / num_samples) * 2 * X.T @ (A - y) + self.regStrength
      total_loss = np.sum(loss) / num_samples
      return total_loss, grad

  def compute_loss_poisson(self, X, y):
      A = X @ self.weights
      A = np.exp(A)
      A = A.reshape(-1)
      mean_l = np.sum(A - y * np.log(res)) / len(X)
      mult_Weights = self.weights * self.weights
      total_loss = mean_l + self.c_reg * np.sum(mult_Weights)
      return total_loss, A

  def compute_loss_poisson(self, X, y):
      A = X @ self.weights
      A = np.exp(A)
```

```
        A = A.reshape(-1)
        mean_l = np.sum(grad - y * np.log(res)) / len(X)
        mult_Weights = self.weights * self.weights
        total_loss = mean_l + self.c_reg * np.sum(mult_Weights)
        return total_loss, grad

    def pseudoinvSol(self, X, y):
        mat_dim = X.shape[1]
        I = np.identity(mat_dim)
        dot_X = np.dot(X.T, X)
        inv = np.linalg.inv(dot_X + self.regStrength * I)
        X_dot_inv = inv @ X.T
        X_dot_inv_y = X_dot_inv @ y

def musicMSE(pred, actual):
  return np.mean((pred.round() - actual)**2)

def loadMusicData(fname, addBias):
  datafile = "YearPredictionMSD.txt"
  data = np.loadtxt(datafile, delimiter=',')

  trainData = data[:463714]
  testData = data[-51630:]

  trainYears = trainData[:,0].reshape((-1,1))
  testYears = testData[:,0].reshape((-1,1))
  trainFeat = np.copy(trainData)[:, 1:]
  testFeat = np.copy(testData)[:, 1:]

  mu = np.mean(trainFeat, axis=0)
  sigma = np.std(trainFeat, axis=0)

  trainFeat = (trainFeat - mu) / sigma
  testFeat = (testFeat - mu) / sigma

  num_samples_training = trainFeat.shape[0]
  num_samples_test = testFeat.shape[0]
  training_ones = np.ones((num_samples_training, addBias))
  testing_ones = np.ones((num_samples_test, addBias))

  trainFeat = np.hstack((training_ones, trainFeat))
```

```
  testFeat = np.hstack((testing_ones, testFeat))
  return trainYears, trainFeat, testYears, testFeat

fname = "YearPredictionMSD.txt"
trainYears, trainFeat, testYears, testFeat = loadMusicData(fname, 1)

n_samples, n_features = trainFeat.shape
params = np.random.standard_normal((n_features+1,1))
initPred = testFeat @ params
init_mse = musicMSE(initPred, testYears)
print("initial MSE using random weights: ", init_mse)

lasso_Regression = Regress(weights=params, epochs=100, lr=0.03, batchsize=100, regType="Las
train_lasso_losses, test_lasso_losses, train_lasso_err, test_lasso_err =
lasso_Regression.train(trainFeat, trainYears, testFeat, testYears)

pred = lasso_Regression.predict(testFeat)
test_mse = musicMSE(pred, testYears)
print("Final test MSE for Lasso: ", test_mse)

exact_weights = lasso_Regression.pseudoinvSol(testFeat, testYears)
pred = testFeat @ exact_weights
actual_mse = musicMSE(pred, testYears)
print("Analytical solution from pseudoinverse: ", actual_mse)

ridge_Regression = Regress(weights=params, epochs=100, lr=0.03, batchsize=100, regType="Rid
train_ridge_losses, test_ridge_losses, train_ridge_err, test_ridge_err = ridge_Regression.f
y_pred = ridge_Regression.predict(testFeat)
test_ridge_mse = musicMSE(y_pred, testYears)
print("Final test MSE for Ridge: ", test_ridge_mse)

exact_weights = ridge_Regression.pseudoinvSol(testFeat, testYears)
pred = testFeat @ exact_weights
actual_mse = musicMSE(pred, testYears)
print("Analytical solution from pseudoinverse: ", actual_mse)

poisson_Regression = Regress(weights=params, epochs=100, lr=0.03, batchsize=100, regType="P
train_losses, test_losses, train_err, test_err = poisson_Regression.fit(X_train, y_train, X
y_pred = poisson_Regression.predict(X_test)
test_mse = musicMSE(y_pred,y_test)
print("Final test MSE for Poisson: ", test_mse)
```