Homework 1
CS 5787 Deep Learning
Spring 2021

Yuxin Zhang - yz2726@cornell.edu

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: [1]

$$\begin{aligned}
\text{softmax}\left(\mathbf{a} + c\mathbf{1}\right) &= \frac{\exp\left(\mathbf{a} + c\mathbf{1}\right)}{\sum_{j=1}^{K}\exp\left(\mathbf{a} + c\mathbf{1}\right)_j} \\
&= \frac{\exp\left(\mathbf{a}\right)\exp\left(c\mathbf{1}\right)}{\sum_{j=1}^{K}\exp\left(\mathbf{a}\right)_j\exp\left(c\mathbf{1}\right)} \\
&= \frac{\exp\left(\mathbf{a}\right)\exp\left(c\mathbf{1}\right)}{\exp\left(c\mathbf{1}\right)\sum_{j=1}^{K}\exp\left(\mathbf{a}\right)_j} \\
&= \frac{\exp\left(\mathbf{a}\right)}{\sum_{j=1}^{K}\exp\left(a_j\right)} = \text{softmax}\left(\mathbf{a}\right)
\end{aligned} \tag{1}$$

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:
When implementing a neural network, the training of network sometimes suffers from numerical stability problem due to output value jumping up and down. With the invariant property of softmax function, we can then subtracting the maximum element from all elements of a, which achieves numerical stability of the NN and reduce the computational amount.

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 hyper-parameters, 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:
In building the softmax classifier, the weight(for each feature) is initialized from Gaussian distribution with mean of 0 and variance of 0.01. We also add bias term by appending a initial zero column in the weight matrix. In preprocessing input data for softmax classifier, we added a column of value 1 at index 0 to ensure correct matrix calculation involving the bias term.
For current batch size of n, input batched_x of shape (n, m+1),batched_y of shape (n, # of classes), W of shape(# of classes, m+1), we have predicted probability =softmax(W.batched_x^T), which is of shape(# of classes, n)
Then we have loss = $\frac{-\sum(\text{batched\_y}*\log(\text{predicted probability}^T)+\alpha\sum W^2}{n}$
Then $\frac{\delta\text{Loss}}{\delta W} = \frac{\delta\text{Cross entropy Loss}}{\delta W} + \frac{\delta\text{Regularization Loss}}{\delta W}$
and $\frac{\delta\text{Cross entropy Loss}}{\delta W} = \frac{(\text{predicted probability}-\text{batched\_y}^T).\text{batched\_x}}{n}$
and $\frac{\delta\text{Regularization Loss}}{\delta W} = \frac{2\alpha W}{n}$
$\delta W = \eta\frac{\delta\text{Loss}}{\delta W} + \text{momentum} * \delta W$
$W = W - \delta W$(Weight updating procedure for each data batch)

The test accuracy(average per-class accuracy) achieved by the one-layer model is 0.8513888888888889, with hyperparameters learning rate = 0.01, batch size = 10, momentum = 0.8, lambda(L2 regularization parameter) = 0. The parameters are tuned by holding out 20% of training data as development set and choose the best parameters combination that produces the highest average per-class accuracy on the development set. The training and test cross-entropy loss and accuracy is trained and plotted using the best parameter combination found above.
Early stopping would not help improve accuracy on the test data in this case though we observe the degrading average-class accuracy rate with more epochs on the test dataset. However, the early stopping decision should be made when the performance on the development set degrade with more epochs, which is not attained in here. Besides, the cross-entropy loss on the training and testing dataset all decreases with more epochs, suggesting non-overfitting in this case.

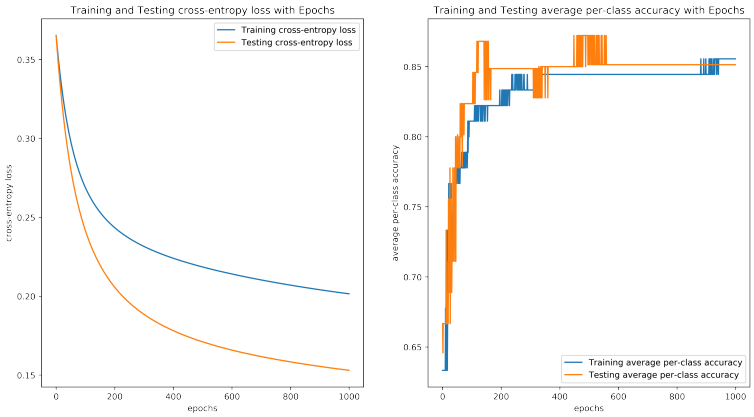


Figure 1: Training, Testing cross-entropy loss and average per-class accuracy with Epochs

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 hyper-parmeters? 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:
[2] After loading the Training and Testing dataset, prepossessing them by normalizing each image vector to range of -1 and 1, and encoding the labels. Hyper-parameters(Learning rate, Batch size, Momentum, Weight decay) is tuned by taking the one that entails the highest average per-class accuracy on the development set(20% from the training dataset). Best hyper-parameters are learning rate=0.05, batch size = 5000, weight decay =0.1, momentum = 0.5. Running the NN with selected hyper-parameters, the test set accuracy achieved is 0.3998. The confusion matrix is normalized and plotted on the test dataset (class 1 to 10), and from the coloring density, we can see that class 3, 4, 5, 6 are where most of the classification error comes from. This indicates that our one-layer softmax NN failed to capture some complex features in those label classes for differentiation. Further, the model seems to confuse class 1 and class 9, class 5 and class 7,class 3 and class 7, class 2 and class 10, class 4 and class 6 the most.

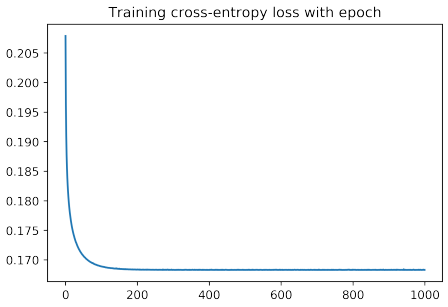


Figure 2: Training cross-entropy loss with Epochs

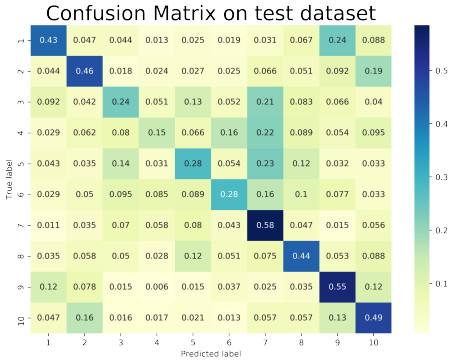


Figure 3: Confusion matrix on test dataset

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:
The range of the feature variables of both training and testing dataset is from -14861.69535 to 65735.77953. We can normalize it by subtract the feature mean and divide it by the its standrad deviation. In this way, all feature can be converted to distribute in the range of 0 and 1.
Year collection from both traning and testing dataset: Year collection: [1922, 1924, 1925, 1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933, 1934, 1935, 1936, 1937, 1938, 1939, 1940, 1941, 1942, 1943, 1944, 1945, 1946, 1947, 1948, 1949, 1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011].

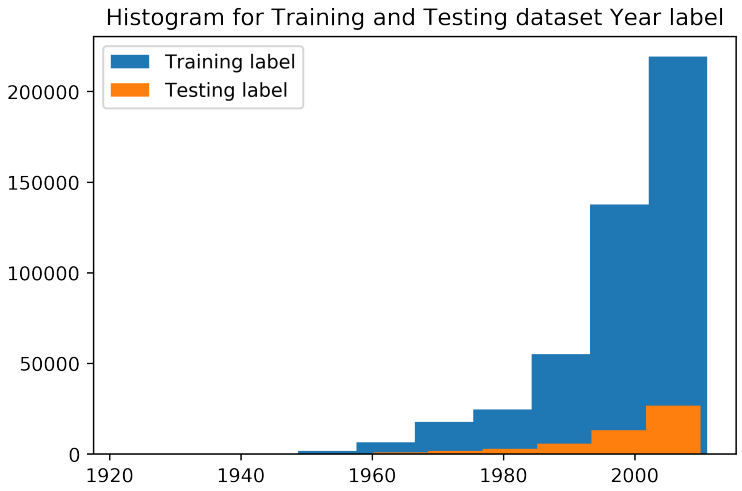


Figure 4: Histogram of training and testing year label

The left-skewed histogram shows a consistent under representation of years before 1990 and an over representation of years after 2000, both in the training and testing dataset.
The most common year in the dataset is 2007, plugging it into the `musicMSE` function with `testYears`, we have the test set MSE = 190.08239236117836
Plugging the mean year 1998 into the `musicMSE` function with `testYears`, we have the test set MSE = 118.0090062171951.

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:
Let X = Current batch input feature data with shape (N, M+1) (M is the of feature dimensions, in this dataset, M = 90)
W = Weight for the linear model with shape (1, M+1)
$\frac{\delta Loss}{\delta W} = -2X^T(Y - XW^T) + 2\lambda W^T$
$W = W - \eta \frac{\delta Loss}{\delta W}$

Using stochastic gradient descent with mini-batches to minimize the Ridge loss, and we evaluate the final training and testing loss using musicMSE we implemented in last part. Also, we use average loss in the weight-updating procedure to ensure the training and testing result has comparable losses.
Before putting training data into model training and hyper-parameter tuning, we normalized all the feature inputs columns in both the training and testing set for efficient gradient descent and fast training purpose (Except for the bias term) Also substract from the minimal Year from the training and testing data (does not affect the loss calculation). We tune hyper-parameters learning rate, batch size, and α(The strength of L2 regularization). We split 20% of the training set as the development set, and we use the development set loss as the standard for hyper-parameter selection. We use epoch = 100 since we observed that regressions are quite quick in converging. Eventually, the development set achieves average set loss of, with learning rate = 0.01, batch size = 10000, epochs = 100, and α = 0.3. Using such hyper-parameters to train the Ridge Regression again on the entire training set and apply the model on the test set, we eventually attain a testing set rounded MSE loss of 90.519184211036.

Figure 5: Training, Testing MSE loss with Epochs

Using the pseudoinverse approach, $W = (X^T(XX^T + \lambda I)^{-1}Y)^T$.
Applying such formula using the above tuned $\alpha(0.3)$ and training dataset(randomly sampled 50% due to RAM explosion issue in dot product calculation), we calculate the model weight. Thus, we attain from this approach, the test dataset MSE loss of 91.1956382793283. The test set MSE using the pseudoinverse approach a bit larger(worse) than the one attained by the SGD due to subsampling reasons.

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

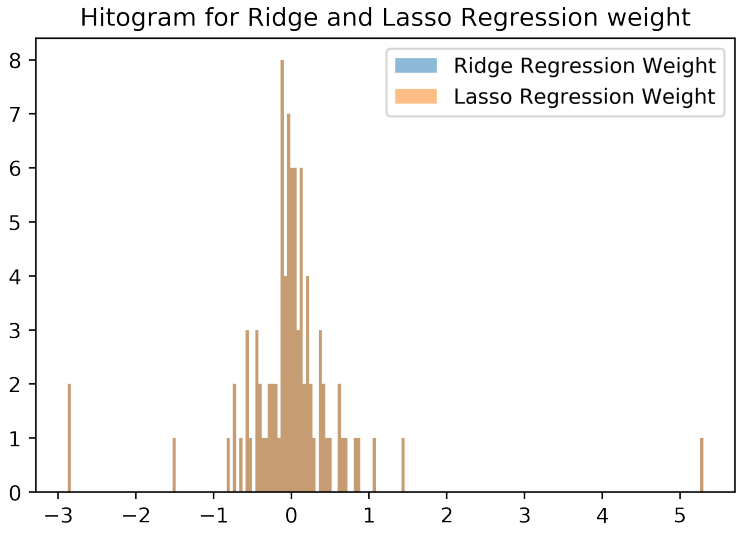
$$L = \sum_{j=1}^{N} \left\| \mathbf{w}^T \mathbf{x}_j - y_j \right\|_2^2 + \alpha \left\| \mathbf{w} \right\|_1^1 .$$

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:
Let X = Current batch input feature data with shape (N, M) (M is the of feature dimensions, in this dataset, M = 90)
W = Weight for the linear model with shape (1, M)

$\frac{\delta Loss}{\delta W} = 2(XW^T - Y)^T X + \lambda \frac{W}{|W|}$
$W = W - \eta \frac{\delta Loss}{\delta W}$
Using stochastic gradient descent with mini-batches to minimize the Ridge loss, and we evaluate the final training and testing loss using musicMSE we implemented in last part. We tune hyper-parameters learning rate, batch size, and α(The strength of L1 regularization). We split 20% of the training set as the development set, and we use the development set loss as the standard for hyper-parameter selection. Eventually, the development set achieves average set loss of 91.49767637449726, with learning rate = 0.01, batch size = 10000, and $\alpha = 0.3$. Using such hyper-parameters to train the Lasso Regression again on the entire training set and apply the model on the test set, we eventually attain a testing set average MSE loss of 90.51914547461796.

We plot the histogram of all weights from Ridge Regression and Lasso Regression (excluding the bias) to see the effect of different regularization terms on the distribution of trained weights.
And the weights shows highly similarity, with most of the weights pushed to zero or very close to zero (average difference is in e-07). This shows that for our dataset, both L1 and L2 regularization demonstrated their power by pushing most weights to 0.



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 $\mathbf{x}$.

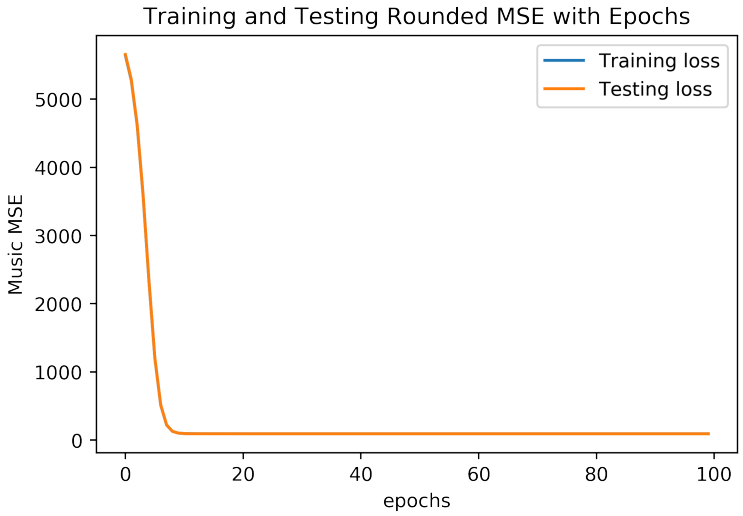
The loss is given by

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

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:
Let X = Current batch input feature data with shape (N, M) (M is the of feature dimensions, in this dataset, M = 90)
W = Weight for the linear model with shape (1, M)
$\frac{\delta Loss}{\delta W} = X^T(e^{XW^T} - Y)$
Using stochastic gradient descent with mini-batches to minimize the Ridge loss, and we evaluate the final training and testing loss using musicMSE we implemented above.
exp term in the loss gradient makes the code exposed to overflow issue. So, we redesign the weight initialization to avoid this problem. After trail-and-error, we initialize it following a Gaussian distribution with 0 mean and 0.01.
We tune hyper-parameters learning rate, batch size. We split 20% of the training set as the

development set, and we use the development set loss as the standard for hyper-parameter selection. Eventually, the development set achieves average set loss of 91.49322320822057, with learning rate =0.01 , batch size =10000 . Using such hyper-parameters to train the Poisson Regression again on the entire training set and apply the model on the test set, we eventually attain a training set average MSE of 91.39489211022311, and a testing set average MSE loss of 90.78770506091301 .



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.

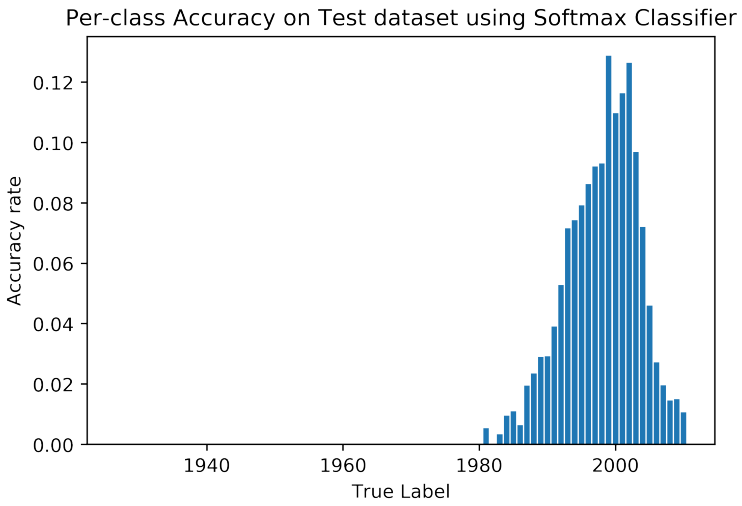
Solution:
Before we call the Softmax classifier implemented before, we need to re-adjust the distribution of feature input of training and testing dataset to -1 and 1, and encode the label in the dataset for classification purpose. And this can be done by calling the function pre_process_softmax_data implemented before. There are in total 89 classes (Years).
We tune the parameters(learning rate, batch size, momentum, L2 regularization) by holding 20% of the training dataset as the development set and attempt attain the hyper-parameter combination with highest average per-class accuracy rate on the development set. And the best hyper-parameter combination is learning rate = 0.01, batch size = 10000, momentum =0.5, (α)L2 regularization =0.5
Running the Softmax classifier on training dataset with the tuned hyper-parameters, and

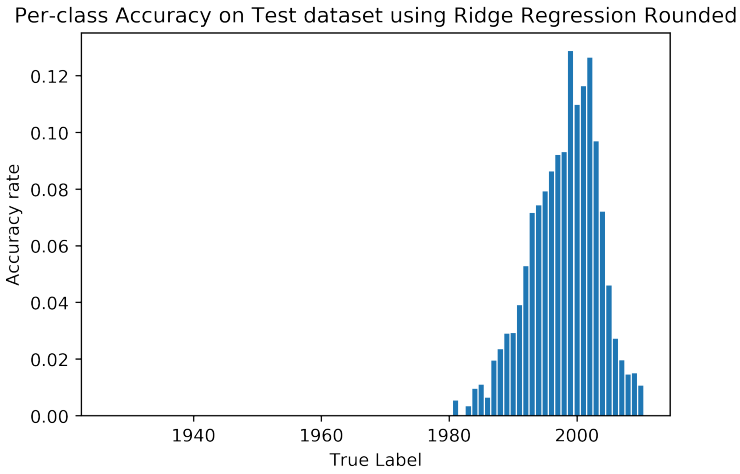
measure its performance on both training and testing dataset by musicMSE cost function implemented before. And it shows the training loss of 188.52020857683831, and test loss of 181.96730646317135, which are much higher than all the three regression approaches discussed above.
Pros: Classification gives discrete integer in model prediction directly, eliminating the rounding error in the regression approach. Also, classification isn't sensitive to outliers like the regression
Cons: Classification trains model based on a limited number of classes in the training dataset and thus predicts that limited number of classes, which has no extending power to predict out-of-training class.

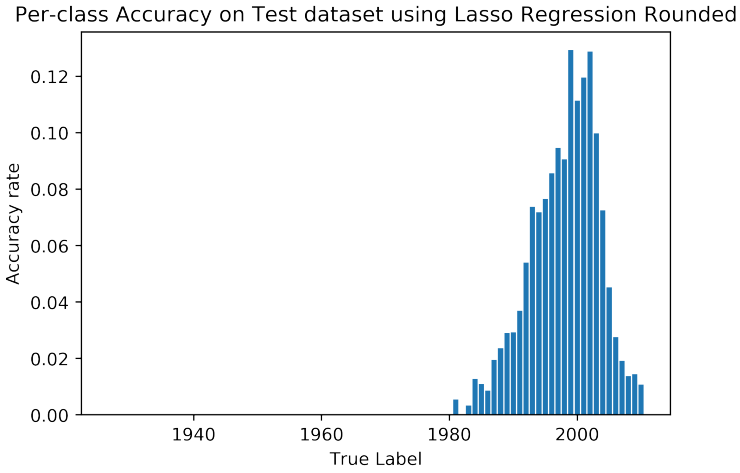
Part 6 - Model Comparison (10 points)

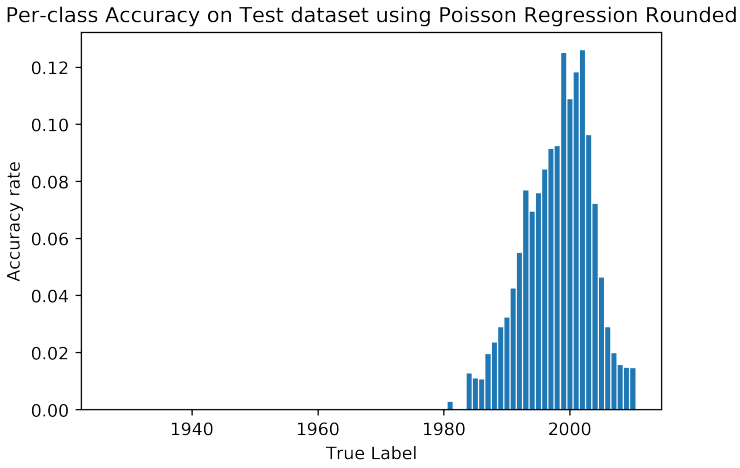
Discuss and compare the behaviors of the models. Are there certain periods (ranges of years) in which models perform better than others? Where are the largest errors across models. Did L_2 regularization help for some models but not others?

Solution:
We compare the behavior of the models by plotting the per-class accuracy in prediction(or classification) on the testing dataset.




Per-class Accuracy on Test dataset using Ridge Regression Rounded
Accuracy rate
0.00
0.02
0.04
0.06
0.08
0.10
0.12
1940
1960
1980
2000
True Label


Per-class Accuracy on Test dataset using Lasso Regression Rounded
Accuracy rate
0.00
0.02
0.04
0.06
0.08
0.10
0.12
1940
1960
1980
2000
True Label



There are range of years across models that performs better than the other years.All models achieve conparatively high accuracy around the year 2000. More detail, the years having classification accuracy over 10% is the year [1999, 2000, 2001, 2002].
Considering years with accuracy rate under 2% as bad performance, then [1927, 1928, 1929, 1930, 1931, 1934, 1936, 1937, 1940, 1941, 1942, 1943, 1944, 1945, 1947, 1948, 1949, 1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980, 2007, 2008, 2009, 2010] are the common poor performance years for all models.
Apparently, the performance of models on range of years is consistent with the data distribution, with under-sampled years having lower test set accuracy. The L2 regularization is applied to the softmax classifier as well as the Ridge Regression, and since the Ridge Regression has better performance(measured by MSE), the regularization helps with Ridge Regression more.

Softmax Classifier Code Appendix

0.1 Part 1

```
import pandas as pd
import numpy as np
import random
import matplotlib.pyplot as plt
%matplotlib inline
def load_data(file_name):
    file = open(file_name)
    data = file.readlines()
```

```
    data_new =[]
    for i in data:
        data_new.append(i.strip().split())
    data = np.array(data_new)
    data = data.astype('float')
    file.close()
    return data
train_data = load_data("iris-train.txt")
test_data = load_data("iris-test.txt")
import numpy as np
import matplotlib.pylab as plt
from sklearn.preprocessing import OneHotEncoder
class SoftmaxClassifier:
    def __init__(self, learning_rate=0.01, n_iterations=2500,
    batch_size = 1,lamdba  = 0,momentum = 0):
        self.parameters = {}
        self.n = 0
        self.learning_rate= learning_rate
        self.epoch = n_iterations
        self.batch_size = batch_size
        self.momentum = momentum
        self.l2_regularization = lamdba

    def softmax(self, x):
        e_x = np.exp(x - np.max(x))
        return e_x / e_x.sum(axis=0, keepdims=True)
    def fit(self, X, Y, X_test, Y_test):
        np.random.seed(1)
        self.n = X.shape[0]
        self.weight = np.random.normal(0, 0.01,(Y.shape[1], X.shape[1]-1))
        self.weight = np.insert(self.weight, 0, 0, axis=1)
        self.train_entropy_loss = []
        self.test_entropy_loss =[]
        self.train_accuracy =[]
        self.test_accuracy =[]
        self.delta_w = np.zeros((Y.shape[1], X.shape[1]))
        for loop in range(self.epoch):
            batches = create_mini_batches(X, Y,self.batch_size)
            for i in range(len(batches)-1):
                batch  = batches[i]
                cur_x = batch[0]
```

```
            cur_y = batch[1]
            cur_predicted = self.softmax(self.weight.dot(cur_x.T))
            derivative_entropy = (cur_predicted - cur_y.T).dot(cur_x)
            / self.batch_size
            derivative_regularization = 2* self.l2_regularization
            * self.weight/ self.batch_size
            self.delta_w =  self.learning_rate *(derivative_entropy +
            derivative_regularization) + self.momentum*self.delta_w
            self.weight -=  self.delta_w

        A_train = self.softmax(self.weight.dot(X.T))
        self.train_entropy_loss.append(-np.mean(Y * np.log(A_train.T+ 1e-8)))
        A_test = self.softmax(self.weight.dot(X_test.T))
        self.test_entropy_loss.append(-np.mean(Y_test * np.log(A_test.T+ 1e-8)))
        self.train_accuracy.append(self.mean_per_class_accuracy(X, Y))
        self.test_accuracy.append(self.mean_per_class_accuracy(X_test, Y_test))
def predict(self,X):
    predicted =self.softmax(self.weight.dot(X.T))
    return np.argmax(predicted, axis=0)
def mean_per_class_accuracy(self,X, Y):
    y_hat = self.predict(X)
    Y = np.argmax(Y, axis=1)
    label_collection = np.unique(Y)
    accu_collection = []
    for i in label_collection:
        indice = np.where(Y == i)
        cur_accu = len(np.where(y_hat[indice] == i)[0]) / len(indice[0])
        accu_collection.append(cur_accu)
    return np.mean(np.array(accu_collection)) #return a scalar
def plot_cost_accuracy(self):
    fig, ax = plt.subplots(1, 2, figsize= (15,8), dpi = 400)
    ax[0].plot(np.arange(len(self.train_entropy_loss)),self.train_entropy_loss,
    label = "Training cross-entropy loss")
    ax[0].plot(np.arange(len(self.test_entropy_loss)),self.test_entropy_loss,
    label = "Testing cross-entropy loss")
    ax[0].legend()
    ax[0].set_xlabel("epochs")
    ax[0].set_ylabel("cross-entropy loss")
    ax[0].set_title("Training and Testing cross-entropy loss with Epochs")
    ax[1].plot(np.arange(len(self.train_accuracy)),  self.train_accuracy,
    label = "Training average per-class accuracy")
```

```
        ax[1].plot(np.arange(len(self.test_accuracy)),  self.test_accuracy,
        label = "Testing average per-class accuracy")
        ax[1].legend()
        ax[1].set_title("Training and Testing average per-class accuracy with Epochs")
        ax[1].set_xlabel("epochs")
        ax[1].set_ylabel("average per-class accuracy")
        plt.show()
def pre_process_softmax_data(train_x, train_y, test_x, test_y):
    #Normalize in between -1 and 1, add bias
    #encode y
    all_feature = np.vstack((train_x,test_x))
    for i in range(all_feature.shape[1]):
        all_feature[:,i] = 2.*(all_feature[:,i] - np.min(all_feature[:,i]))
        /np.ptp(all_feature[:,i])-1
    bias_all_feature = np.insert(all_feature, 0, 1, axis=1)
    processed_train_x = bias_all_feature[:len(train_x)]
    processed_test_x = bias_all_feature[len(train_x):]
    enc = OneHotEncoder(sparse=False, categories='auto')
    train_y = enc.fit_transform(train_y.reshape(len(train_y), -1))
    test_y = enc.transform(test_y.reshape(len(test_y), -1))
    label_array = enc.inverse_transform(test_y).ravel()
    return processed_train_x, train_y, processed_test_x, test_y, label_array
def create_mini_batches(X, y, batch_size):
        mini_batches = []
        data = np.hstack((X, y))
        np.random.shuffle(data)
        n_minibatches = data.shape[0] // batch_size
        for i in range(n_minibatches + 1):
            mini_batch = data[i * batch_size:(i + 1)*batch_size, :]
            X_mini = mini_batch[:, :X.shape[1]]
            Y_mini = mini_batch[:, X.shape[1]:]
            mini_batches.append((X_mini, Y_mini))
        if data.shape[0] % batch_size != 0:
            mini_batch = data[i * batch_size:data.shape[0]]
            X_mini = mini_batch[:, :X.shape[1]]
            Y_mini = mini_batch[:, X.shape[1]:]
            mini_batches.append((X_mini, Y_mini))
        return mini_batches
from sklearn.model_selection import train_test_split
lr = [0.001, 0.01, 0.05, 0.1]
batch_size = [10,30, 50, 72]
```

```
momentum = [0, 0.1, 0.3, 0.5, 0.8]
lambda_lst = [0, 0.01, 0.1, 0.3, 0.5, 0.8]
def tune_parameter(lr,batch_size,momentum,lambda_lst, train_x, train_y):
    #holdout 20% from the training as the development set
    train_x, develop_x,  train_y, develop_y = train_test_split(train_x,
    train_y, test_size=0.2, random_state=42)
    print("train_x's shape: " + str(train_x.shape))
    print("develop_x's shape: " + str(develop_x.shape))
    print("train_y's shape: " + str(train_y.shape))
    print("develop_y's shape: " + str(develop_y.shape))
    best_comb = None
    best_develop_accuracy = 0
    for i in lr:
        for j in batch_size:
            for l in momentum:
                for p in lambda_lst:
                    ann = SoftmaxClassifier(learning_rate=i, n_iterations=1000,
                    batch_size = j, lamdba =p, momentum =l)
                    ann.fit(train_x, train_y,develop_x, develop_y)
                    cur_develop_accuracy = ann.test_accuracy[-1]
                    print(cur_develop_accuracy)
                    if cur_develop_accuracy > best_develop_accuracy:
                        best_develop_accuracy = cur_develop_accuracy
                        best_comb = [i,j,l, p]
    print("Best Hyperparameter combination:\nlearning rate = %s\nbatch size = %s\nmomentum
    %s\nlambda(L2 regularization parameter) = %s\nWith best average per-class accuracy on d
    %s"%(best_comb[0],best_comb[1],best_comb[2],best_comb[3],best_develop_accuracy))
train_data = load_data("iris-train.txt")
test_data = load_data("iris-test.txt")
train_x, train_y, test_x, test_y,label_array = pre_process_softmax_data(train_data[:,1:],
train_data[:,0].astype("int"), test_data[:,1:], test_data[:,0].astype("int"))
tune_parameter(lr,batch_size,momentum,lambda_lst,train_x, train_y)
if __name__ == '__main__':
    train_data = load_data("iris-train.txt")
    test_data = load_data("iris-test.txt")
    train_x, train_y, test_x, test_y, labels = pre_process_softmax_data(train_data[:,1:],
    train_data[:,0].astype("int"), test_data[:,1:], test_data[:,0].astype("int"))
    print("train_x's shape: " + str(train_x.shape))
    print("test_x's shape: " + str(test_x.shape))
    print("train_y's shape: " + str(train_y.shape))
    print("test_y's shape: " + str(test_y.shape))
```

```
    ann = SoftmaxClassifier(learning_rate=0.01, n_iterations=1000,
    batch_size = 10, lamdba =0, momentum = 0.5)
    ann.fit(train_x, train_y,test_x, test_y)
    print("Train Accuracy:", ann.mean_per_class_accuracy(train_x, train_y))
    print("Test Accuracy:", ann.mean_per_class_accuracy(test_x, test_y))
    ann.plot_cost_accuracy()
```

0.2 Part 2

```
def plot_graph(train_data,test_data):
    plt.figure(dpi = 400)
    # Plot the decision boundary. For that, we will assign a color to each
    cdict = {1: 'red', 2: 'blue', 3: 'green'}
    ann = SoftmaxClassifier()
    train_x, train_y, test_x, test_y, labels = pre_process_softmax_data(
    train_data[:,1:],train_data[:,0].astype("int"), test_data[:,1:],
    test_data[:,0].astype("int"))
    ann = SoftmaxClassifier(learning_rate=0.01, n_iterations=1000,
    batch_size = 10, lamdba =0, momentum = 0.5)
    ann.fit(train_x, train_y,test_x, test_y)
    min1, max1 = train_x[:, 0].min()-1, train_x[:, 0].max()+1
    min2, max2 = train_x[:, 1].min()-1, train_x[:, 1].max()+1
    x1grid = np.arange(min1, max1, 0.01)
    x2grid = np.arange(min2, max2, 0.01)
    xx, yy = np.meshgrid(x1grid, x2grid)
    r1, r2 = xx.flatten(), yy.flatten()
    r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))

    grid = np.hstack((r1,r2))
    bias_grid = np.insert(grid, 0, 1, axis=1)
    yhat = ann.predict(bias_grid) +1
    zz = yhat.reshape(xx.shape)
    plt.contour(xx, yy, zz)
    for g in np.unique(train_data[:,0]):
        ix = np.where(train_data[:,0] == g)
        plt.scatter(train_x[ix,1],train_x[ix,2], c = cdict[g],
        label = "class "+str(int(g)), s = 10)
    plt.title("Decision boundary and Scatter Plot")
    plt.legend()
    plt.show()
plot_graph(train_data, test_data)
```

0.3 Problem 3

```
def pre_process_image_data(train_x, train_y, test_x, test_y):
    # Normalize in between -1 and 1
    train_x = ((train_x/255)-0.5)*2
    test_x = ((test_x/255)-0.5)*2
    # encode the y label to have shape N X (# of classes)
    enc = OneHotEncoder(sparse=False, categories='auto')
    train_y = enc.fit_transform(train_y.reshape(len(train_y), -1))
    test_y = enc.transform(test_y.reshape(len(test_y), -1))
    return train_x, train_y, test_x, test_y
import pickle
np.seterr(divide='ignore', invalid='ignore')
def unpickle(file):
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict
data_dic = unpickle("data_batch_1")
test_dic= unpickle("test_batch")
train_x = np.array(data_dic[b'data'])
test_x = np.array(test_dic[b'data'])
train_y = np.array(data_dic[b'labels'])
test_y = np.array(test_dic[b'labels'])

train_x, train_y, test_x, test_y=pre_process_image_data(train_x, train_y,
test_x, test_y)
lr = [0.005, 0.01, 0.05, 0.1]
batch_size = [100, 1000,5000, 8000]
momentum = [0, 0.5,0.9]
lambda_lst = [ 0.1, 0.5, 0.8]
tune_parameter(lr,batch_size,momentum,lambda_lst, train_x, train_y)
model_best = SoftmaxClassifier(learning_rate=0.05, n_iterations=1000,
batch_size = 5000, lamdba =0.1, momentum =0.5)
model_best.fit(train_x, train_y,test_x, test_y)
plt.figure(dpi = 400)
plt.plot(np.arange(len(model_best.train_entropy_loss)),
model_best.train_entropy_loss)
plt.title("Training cross-entropy loss with epoch")
plt.show()
print("Test dataset average per-class accuracy:%s"%(model_best.test_accuracy[-1]))
from sklearn.metrics import confusion_matrix
```

```
import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt
y = np.argmax(test_y, axis=1)
y_pred = model_best.predict(test_x)
cf_matrix = confusion_matrix(y, y_pred, normalize = 'true')

df_cm = pd.DataFrame(cf_matrix ,index = [i for i in range(1,11)],
                  columns = [i for i in range(1,11)])
plt.figure(figsize = (10,7), dpi = 400)
ax = plt.axes()

sn.heatmap(df_cm, ax = ax, annot=True, cmap="YlGnBu")
ax.set_xlabel("Predicted label")
ax.set_ylabel("True label")
ax.set_title('Confusion Matrix on test dataset', fontsize = 25)
plt.show()
```

Linear Models for Regression Code Appendix

0.4 Part 1

```
def loadMusicData(fname, addBias = True):
    file = open(fname, "r")
    data = file.readlines()
    data_new = []
    count = 0
    for i in data:
        cur_row = i.strip().split(",")
        data_new.append(cur_row)
    training = np.array(data_new[:463714])
    testing = np.array(data_new[463714:])
    trainFeat = training[:,1:]
    trainYears= training[:,0].reshape(-1,1)
    testFeat = testing[:,1:]
    testYears = testing[:,0].reshape(-1,1)
    trainFeat = trainFeat.astype("float")
    testFeat = testFeat.astype("float")
    trainYears = trainYears.astype("int")
    testYears = testYears.astype("int")
```

```
    # add the bias
    if addBias:
        trainFeat = np.append(np.ones([len(trainFeat),1]),trainFeat,1)
        testFeat = np.append(np.ones([len(testFeat),1]),testFeat,1)
    file.close()
    return trainYears, trainFeat, testYears, testFeat
def musicMSE(pred, gt):
    pred = np.rint(pred)
    return (np.square(pred - gt)).mean()
def dataset_assessment(trainYears, trainFeat, testYears, testFeat):
    all_feature = np.concatenate([trainFeat, testFeat])
    print("Range of feature variables: %s ~ %s"%(np.min(all_feature),
    np.max(all_feature)))
    all_label = np.concatenate([trainYears,testYears])
    print("Year collection: ",sorted(np.unique(all_label.flatten())))
    plt.figure(dpi = 400)
    plt.hist(trainYears.flatten(), label = "Training label")
    plt.hist(testYears.flatten(), label = "Testing label")
    plt.legend()
    plt.title("Histogram for Training and Testing dataset Year label")
    plt.show()
    most_freq_year = np.argmax(np.bincount(all_label.flatten()))
    print("If the classifier always outputs the most common year %s in the dataset,
    the test set MSE = %s" %(most_freq_year, musicMSE(most_freq_year, testYears)))
    print("If the classifier always outputs the year 1998 in the dataset,
    the test set MSE = %s"%musicMSE(1998,testYears ))
trainYears, trainFeat, testYears, testFeat = loadMusicData("YearPredictionMSD.txt",
addBias = True)
dataset_assessment(trainYears, trainFeat, testYears, testFeat)
```

0.5 Part 2

```
class RidgeRegression:
    def __init__(self,learning_rate=0.01, epochs = 1000, batch_size = 1, alpha = 0):
        self.n = 0
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size
        self.alpha = alpha
    def predict(self,X):
        return X.dot(self.weight.T )
```

```
    def fit(self, X, Y, X_test, Y_test):
        np.random.seed(1)
        self.n = X.shape[0]
        self.weight = np.random.normal(0, 0.01,(Y.shape[1], X.shape[1]-1))

        self.weight = np.insert( self.weight,0,0).reshape(1, X.shape[1])
        self.train_loss = []
        self.test_loss =[]
        self.delta_w = np.zeros((Y.shape[1], X.shape[1]))
        for loop in range(self.epochs):
            batches = create_mini_batches(X, Y, self.batch_size)
            for i in range(len(batches)-1):
                batch  = batches[i]
                cur_x = batch[0]
                cur_y = batch[1]
                predicted = self.predict(cur_x)
                self.delta_w =  ( - ( 2 * ( cur_x.T).dot(cur_y - predicted ) ) +
                2 *self.alpha* self.weight.T )/self.batch_size
                self.weight -=  self.learning_rate* self.delta_w.T
            predict_train = self.predict(X)
            cur_loss = musicMSE(predict_train, Y)
            self.train_loss.append(cur_loss)
            predict_test = self.predict(X_test)
            self.test_loss.append(musicMSE(predict_test, Y_test))
    def plot_cost(self):
            plt.figure(dpi = 400)
            plt.plot(np.arange(len(self.train_loss)),self.train_loss,
            label = "Training loss")
            plt.plot(np.arange(len(self.test_loss)),self.test_loss,
            label = "Testing loss")
            plt.legend()
            plt.xlabel("epochs")
            plt.ylabel("Music MSE")
            plt.title("Training and Testing Rounded MSE with Epochs")
            plt.show()
def create_mini_batches(X, y, batch_size):
        mini_batches = []
        data = np.hstack((X, y))
        np.random.shuffle(data)
        n_minibatches = data.shape[0] // batch_size
        for i in range(n_minibatches + 1):
```

```
            mini_batch = data[i * batch_size:(i + 1)*batch_size, :]
            X_mini = mini_batch[:, :X.shape[1]]
            Y_mini = mini_batch[:, X.shape[1]:]
            mini_batches.append((X_mini, Y_mini))
        if data.shape[0] % batch_size != 0:
            mini_batch = data[i * batch_size:data.shape[0]]
            X_mini = mini_batch[:, :X.shape[1]]
            Y_mini = mini_batch[:, X.shape[1]:]
            mini_batches.append((X_mini, Y_mini))
        return mini_batches
def normalize(feature):
    for i in range(feature.shape[1]):
        feature[:,i] = (feature[:,i] - np.mean(feature[:,i]))
        / np.std(feature[:,i])
    return feature
all_feature = np.concatenate([trainFeat, testFeat])
all_label = np.concatenate([trainYears, testYears])
all_feature[:,1:] = normalize(all_feature[:,1:])
all_label = all_label -np.min(all_label)
train_x = all_feature[:len(trainFeat)]
test_x = all_feature[len(trainFeat):]
train_y = all_label[: len(trainFeat)]
test_y = all_label[len(trainFeat): ]
from sklearn.model_selection import train_test_split
lr = [0.01, 0.05, 0.1]
batch_size = [500,1000, 5000, 10000, 20000]
lambda_lst = [0, 0.1,0.5, 0.8]
#select the best hyperparameters based on smallest development set loss
def tune_parameter(lr,batch_size,lambda_lst, train_x, train_y):
    #holdout 20% from the training as the development set
    train_x, develop_x,  train_y, develop_y =
    train_test_split(train_x, train_y, test_size=0.2, random_state=42)
    print("train_x's shape: " + str(train_x.shape))
    print("develop_x's shape: " + str(develop_x.shape))
    print("train_y's shape: " + str(train_y.shape))
    print("develop_y's shape: " + str(develop_y.shape))
    best_comb = None
    best_develop_loss= 10000000
    for i in lr:
        for j in batch_size:
```

```
            for p in lambda_lst:
                ann = RidgeRegression(learning_rate=i, epochs = 100,
                batch_size = j, alpha = p)
                ann.fit(train_x, train_y,develop_x, develop_y)
                cur_develop_loss = ann.test_loss[-1]
                print(cur_develop_loss)
                if cur_develop_loss < best_develop_loss:
                    best_develop_loss = cur_develop_loss
                    best_comb = [i,j, p]
    print("Best Hyperparameter combination:\nlearning rate = %s\nbatch size = %s
    \nlambda(L2 regularization parameter) = %s\nWith best development set loss =
    %s"%(best_comb[0],best_comb[1],best_comb[2],best_develop_loss))
tune_parameter(lr,batch_size,lambda_lst, train_x, train_y)
best_model = RidgeRegression(learning_rate=0.01, epochs = 100,
batch_size = 10000, alpha = 0.3)
best_model.fit(train_x, train_y, test_x, test_y)
best_model.plot_cost()
print("The test set MSE loss is %s"%best_model.test_loss[-1])
from numpy.linalg import inv
stack = np.hstack((train_x,train_y))
idx = np.random.randint(len(train_x), size=15000)
train_x_selected = stack[idx, :-1]
train_y_selected = stack[idx, -1].reshape(15000, 1)
a1 = train_x_selected.dot(train_x_selected.T)+0.3 * np.identity(len(train_x_selected))
a2 = inv(a1).dot(train_y_selected)
a3 = train_x_selected.T.dot(a2)
w = a3.T
predicted_test = test_x.dot(w.T )
print("The test set loss is :%s"%(musicMSE(predicted_test,test_y)))
```

0.6 Part 3

```
class LassoRegression:
    def __init__(self,learning_rate=0.01, epochs = 1000, batch_size = 1, alpha = 0):
        self.n = 0
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size
        self.alpha = alpha
    def predict(self,X):
        return X.dot(self.weight.T )
```

```
    def fit(self, X, Y, X_test, Y_test):
        np.random.seed(1)
        self.n = X.shape[0]
        self.weight = np.random.normal(0, 0.01,(Y.shape[1], X.shape[1]-1))
        self.weight = np.insert( self.weight,0,0).reshape(1, X.shape[1])
        self.train_loss = []
        self.test_loss =[]
        self.delta_w = np.zeros((Y.shape[1], X.shape[1]))
        for loop in range(self.epochs):
            batches = create_mini_batches(X, Y, self.batch_size)
            for i in range(len(batches)-1):
                batch  = batches[i]
                cur_x = batch[0]
                cur_y = batch[1]
                predicted = self.predict(cur_x)
                self.delta_w =    (2 * (cur_x.dot(self.weight.T) - cur_y).T.dot
                (cur_x)  +  self.alpha* ((self.weight+1e-8) / np.absolute(self.weight+1e-8)
                /self.batch_size
                self.weight -=  self.learning_rate* self.delta_w
            predict_train = self.predict(X)
            self.train_loss.append(musicMSE(predict_train, Y))
            predict_test = self.predict(X_test)
            self.test_loss.append(musicMSE(predict_test, Y_test))
    def plot_cost(self):
            plt.figure(dpi = 400)
            plt.plot(np.arange(len(self.train_loss)),self.train_loss,
            label = "Training loss")
            plt.plot(np.arange(len(self.test_loss)),self.test_loss,
            label = "Testing loss")
            plt.legend()
            plt.xlabel("epochs")
            plt.ylabel("Music MSE")
            plt.title("Training and Testing Rounded MSE with Epochs")
            plt.show()
def create_mini_batches(X, y, batch_size):
        mini_batches = []
        data = np.hstack((X, y))
        np.random.shuffle(data)
        n_minibatches = data.shape[0] // batch_size
        for i in range(n_minibatches + 1):
            mini_batch = data[i * batch_size:(i + 1)*batch_size, :]
```

```
            X_mini = mini_batch[:, :X.shape[1]]
            Y_mini = mini_batch[:, X.shape[1]:]
            mini_batches.append((X_mini, Y_mini))
        if data.shape[0] % batch_size != 0:
            mini_batch = data[i * batch_size:data.shape[0]]
            X_mini = mini_batch[:, :X.shape[1]]
            Y_mini = mini_batch[:, X.shape[1]:]
            mini_batches.append((X_mini, Y_mini))
        return mini_batches
from sklearn.model_selection import train_test_split
lr = [0.01, 0.05, 0.1]
batch_size = [500,1000, 5000, 10000, 20000]
lambda_lst = [0, 0.1,0.5, 0.8]
def tune_Lasso_parameter(lr,batch_size,lambda_lst, train_x, train_y):
    #holdout 20% from the training as the development set
    train_x, develop_x,  train_y, develop_y =
    train_test_split(train_x, train_y, test_size=0.2, random_state=42)
    print("train_x's shape: " + str(train_x.shape))
    print("develop_x's shape: " + str(develop_x.shape))
    print("train_y's shape: " + str(train_y.shape))
    print("develop_y's shape: " + str(develop_y.shape))
    best_comb = None
    best_develop_loss= 10000000
    for i in lr:
        for j in batch_size:
            for p in lambda_lst:
                ann = LassoRegression(learning_rate=i, epochs = 100,
                batch_size = j, alpha = p)
                ann.fit(train_x, train_y,develop_x, develop_y)
                cur_develop_loss = ann.test_loss[-1]
                print(cur_develop_loss)
                if cur_develop_loss < best_develop_loss:
                    best_develop_loss = cur_develop_loss
                    best_comb = [i,j, p]
    print("Best Hyperparameter combination:\nlearning rate = %s\nbatch size =
    %s\nlambda(L1 regularization parameter) = %s\nWith best development set loss = %s"
    %(best_comb[0],best_comb[1],best_comb[2],best_develop_loss))
tune_parameter(lr,batch_size,lambda_lst, train_x, train_y)
best_lasso_model = LassoRegression(learning_rate=0.01, epochs = 100,
batch_size = 10000, alpha = 0.3)
best_lasso_model.fit(train_x, train_y, test_x, test_y)
```

```
best_lasso_model.plot_cost()
print("The test set MSE loss is %s"%best_lasso_model.test_loss[-1])
weight_l2 = best_model.weight[:,1:]
weight_l1 = best_lasso_model.weight[:,1:]
plt.figure(dpi = 400)
plt.hist(weight_l2.flatten(),bins = 200, label = "Ridge Regression Weight",alpha=0.5)
plt.hist(weight_l2.flatten(),bins = 200, label = "Lasso Regression Weight",alpha=0.5)
plt.title("Hitogram for Ridge and Lasso Regression weight")
plt.legend()
plt.show()
```

0.7 Part 4

```
class PoissonRegression:
    def __init__(self,learning_rate=0.01, epochs = 1000, batch_size = 1):
        self.n = 0
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size

    def predict(self,X):
        return np.exp(X.dot(self.weight.T ))
    def fit(self, X, Y, X_test, Y_test):
        np.random.seed(1)
        self.n = X.shape[0]
        self.weight = np.random.normal(0, 0.01, (Y.shape[1], (X.shape[1]-1) ))
        self.weight = np.insert( self.weight,0,0).reshape(1, X.shape[1])
        self.train_loss = []
        self.test_loss =[]
        self.delta_w = np.zeros((Y.shape[1], X.shape[1]))
        for loop in range(self.epochs):
            batches = create_mini_batches(X, Y, self.batch_size)
            for i in range(len(batches)-1):
                batch  = batches[i]
                cur_x = batch[0]
                cur_y = batch[1]
                predicted = self.predict(cur_x)
                self.delta_w =   cur_x.T.dot(np.exp(cur_x.dot(self.weight.T))
                - cur_y)/self.n
                self.weight -=  self.learning_rate* self.delta_w.T
            predict_train = self.predict(X)
```

```
            self.train_loss.append(musicMSE(predict_train, Y))
            predict_test = self.predict(X_test)
            self.test_loss.append(musicMSE(predict_test, Y_test))
    def plot_cost(self):
            plt.figure(dpi = 400)
            plt.plot(np.arange(len(self.train_loss)),self.train_loss,
            label = "Training loss")
            plt.plot(np.arange(len(self.test_loss)),self.test_loss,
            label = "Testing loss")
            plt.legend()
            plt.xlabel("epochs")
            plt.ylabel("Music MSE")
            plt.title("Training and Testing Rounded MSE with Epochs")
            plt.show()
def create_mini_batches(X, y, batch_size):
        mini_batches = []
        data = np.hstack((X, y))
        np.random.shuffle(data)
        n_minibatches = data.shape[0] // batch_size
        for i in range(n_minibatches + 1):
            mini_batch = data[i * batch_size:(i + 1)*batch_size, :]
            X_mini = mini_batch[:, :X.shape[1]]
            Y_mini = mini_batch[:, X.shape[1]:]
            mini_batches.append((X_mini, Y_mini))
        if data.shape[0] % batch_size != 0:
            mini_batch = data[i * batch_size:data.shape[0]]
            X_mini = mini_batch[:, :X.shape[1]]
            Y_mini = mini_batch[:, X.shape[1]:]
            mini_batches.append((X_mini, Y_mini))
        return mini_batches
from sklearn.model_selection import train_test_split
lr = [0.001,0.01,  0.05, 0.1]
batch_size = [500,1000, 5000, 10000, 20000]
def tune_Poisson_parameter(lr,batch_size, train_x, train_y):
    train_x, develop_x,  train_y, develop_y =
    train_test_split(train_x, train_y, test_size=0.2, random_state=42)
    print("train_x's shape: " + str(train_x.shape))
    print("develop_x's shape: " + str(develop_x.shape))
    print("train_y's shape: " + str(train_y.shape))
    print("develop_y's shape: " + str(develop_y.shape))
    best_comb = None
```

```
    best_develop_loss= 10000000
    for i in lr:
        for j in batch_size:
            ann = PoissonRegression(learning_rate=i, epochs = 100, batch_size = j)
            ann.fit(train_x, train_y,develop_x, develop_y)
            cur_develop_loss = ann.test_loss[-1]
            print(cur_develop_loss)
            if cur_develop_loss < best_develop_loss:
                best_develop_loss = cur_develop_loss
                best_comb = [i,j]
    print("Best Hyperparameter combination:\nlearning rate = %s\nbatch size = %s
    \nWith best development set loss = %s"%(best_comb[0],best_comb[1],best_develop_loss))
tune_Poisson_parameter(lr,batch_size, train_x, train_y)
best_model = PoissonRegression(learning_rate=0.001, epochs = 100, batch_size = 20000)
best_model.fit(train_x, train_y, test_x, test_y)
best_model.plot_cost()
print("The music MSE loss on the training set is %s"%(best_model.train_loss[-1]))
print("The music MSE loss on the testing set is %s"%(best_model.test_loss[-1]))
```

0.8 Part 5

```
trainYears, trainFeat, testYears, testFeat = loadMusicData("YearPredictionMSD.txt",
addBias = False)
train_x_class, train_y_class, test_x_class, test_y_class, label_array_class =
pre_process_softmax_data(trainFeat,trainYears,  testFeat, testYears)
lr = [0.01, 0.05, 0.1]
batch_size =  [1000, 5000, 10000, 20000]
momentum = [0,0.3, 0.5]
lambda_lst = [ 0.1, 0.5, 0.8]
def tune_softmax_parameter(lr,
batch_size,momentum,lambda_lst, train_x, train_y):
    #holdout 20% from the training as the development set
    train_x, develop_x,  train_y, develop_y = train_test_split(train_x, train_y,
    test_size=0.2, random_state=42)
    print("train_x's shape: " + str(train_x.shape))
    print("develop_x's shape: " + str(develop_x.shape))
    print("train_y's shape: " + str(train_y.shape))
    print("develop_y's shape: " + str(develop_y.shape))
    best_comb = None
    best_develop_accuracy = 0
```

```
    for i in lr:
        for j in batch_size:
            for l in momentum:
                for p in lambda_lst:
                    ann = SoftmaxClassifier(learning_rate=i, n_iterations=1000,
                    batch_size = j, lamdba =p, momentum =l)
                    ann.fit(train_x, train_y,develop_x, develop_y)
                    cur_develop_accuracy = ann.test_accuracy[-1]
                    print(cur_develop_accuracy)
                    if cur_develop_accuracy >
                    best_develop_accuracy:
                        best_develop_accuracy = cur_develop_accuracy
                        best_comb = [i,j,l, p]
    print("Best Hyperparameter combination:\nlearning rate = %s\nbatch size = %s
    \nmomentum = %s\nlambda(L2 regularization parameter) = %s\nWith best average per-class
    accuracy on development = %s"%(best_comb[0],best_comb[1],
    best_comb[2],best_comb[3],best_develop_accuracy))
tune_softmax_parameter(lr,batch_size,momentum,lambda_lst, train_x_class, train_y_class)
best_softmax_model = SoftmaxClassifier( learning_rate=0.01, n_iterations=1000,
batch_size = 10000, lamdba =0.5, momentum =0.5)
best_softmax_model.fit(train_x_class, train_y_class,test_x_class, test_y_class)
train_predicted_index = best_softmax_model.predict(train_x_class)
predicted_train_y =label_array_class[train_predicted_index]
test_predicted_index = best_softmax_model.predict(test_x_class)
predicted_test_y =label_array_class[test_predicted_index]
print(musicMSE(predicted_train_y, trainYears.flatten()))
print(musicMSE(predicted_test_y, testYears.flatten()))
best_softmax_model.plot_cost_accuracy()
```

0.9 Part 6

```
from sklearn.metrics import confusion_matrix
import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt
def confusion_matrix_plot(name, trained_model, x_test, x_test_class, y_test):
    if name == "Softmax Classifier":
        y_pred = label_array_class[trained_model.predict(test_x_class)]
    else:
        y_pred = np.rint(trained_model.predict(x_test)) + np.min(all_label)
    cf_matrix = confusion_matrix(testYears.flatten(), y_pred, normalize = 'true')
```

```
    label_array_true = np.unique(y_test)+ np.min(all_label)
    total_label = np.unique(np.append(y_pred,y_test))
    df_cm = pd.DataFrame(cf_matrix ,index = total_label,
                         columns = total_label)
    mask = np.isin(total_label, label_array_true)
    chosem_cm = df_cm.values[mask, :][:,mask]
    #calculate the per-class accuracy
    plt.figure(dpi = 400)
    class_accuracy = []
    print(label_array_true)
    for i in range(len(label_array_true )):
        class_accuracy.append(chosem_cm[i][i])
    plt.bar(label_array_true,np.array(class_accuracy))
    plt.xlabel("True Label")
    plt.ylabel("Accuracy rate")
    plt.title("Per-class Accuracy on Test dataset using %s"%(name))
    plt.show()
    return class_accuracy
confusion_matrix_plot("Softmax Classifier", best_softmax_model, test_x, test_x_class, test_
confusion_matrix_plot("Ridge Regression Rounded", best_ridge_model, test_x, test_x_class,te
confusion_matrix_plot("Lasso Regression Rounded", best_lasso_model, test_x, test_x_class,te
confusion_matrix_plot("Poisson Regression Rounded", best_poisson_model,  test_x, test_x_cla
```

References

[1] URL: https://tex.stackexchange.com/questions/8936/how-to-break-a-long-equation.

[2] URL: https://medium.com/@dtuk81/confusion-matrix-visualization-fc31e3f30fea.