Homework 1
CS 5787 Deep Learning
Spring 2019

Submitted by Eric Nguyen, en274@cornell.edu

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:

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

softmax(a + c) $= \frac{exp(a+c)}{\sum_{i=1}^{K} exp(a_j+c)}$

$= \frac{exp(a)exp(c)}{\sum_{i=1}^{K} exp(a_j)exp(c)}$

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

Part 2 (3 points)

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

Solution:

It's important because it allows you to not need to stack a bias term on to your feature inputs. Since softmax is invariant to constants, you can just use the input features as is, and do not need to use the "bias trick" when coming up with the dimensions of the Weights matrix.

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.

```python
def load_data():
    # load data
    train_data = loadtxt('iris-train.txt')
    x_train = train_data[:,1:]
    y_train = train_data[:,0].astype(int)-1  # make sure to minus 1 for label
    y_train = y_train.reshape((-1, 1))  # convert to column vector
    test_data = loadtxt('iris-test.txt')
    x_test = test_data[:,1:]
    y_test = test_data[:,0].astype(int)-1  # make sure to minus 1 for label
    y_test = y_test.reshape((-1, 1))   # convert to column vector
    return x_train, y_train, x_test, y_test
```

Listing 1: Python example

Part 1 - Implementation & Evaluation (20 points)

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

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

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:

See appendix for Softmax Classifier code.

The best test accuracy was **0.804**.

The hyperameters I used were:

Epochs: 1000
Learning rate: [0.1, 0.01, 0.001]
Regularization: [1, 0.1, 0.01, 0.001]
batch size: [1, 2, 10, 12, 100]
momentum: 0-1

Early stopping would have improved accuracy on the test data, yes. I grabbed the max accuracy over all epochs, and would save the weights from this epoch next time. This occurs because the model is overfitting to the training data, and not generalizing to the test data. Therefore, early stopping is one way to get better results than using the weights at the end.

Softmax Classifier: by far, the simplest model to train was the softmax classifier on the iris dataset. The fewer dimensions and sample size allowed for rapid experimentation. The model ran in a matter of a few seconds so I could constantly try different hyperparameters, and could get the accuracy up to about 80%

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:

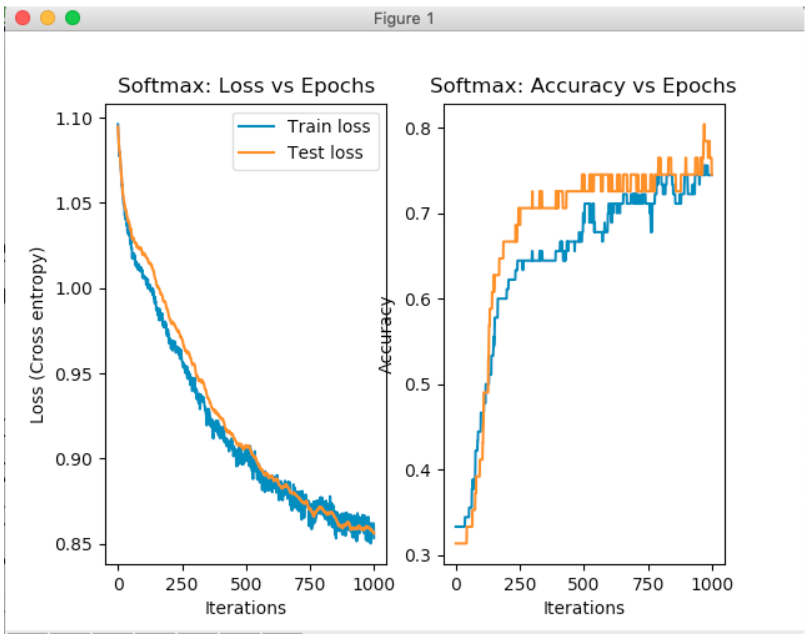

Figure 1: Softmax classifier on Iris dataset

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.

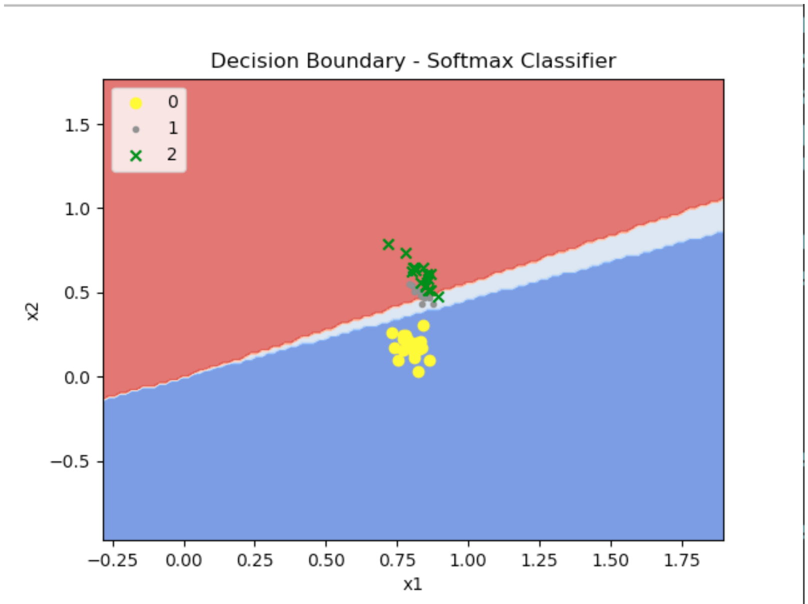

Figure 2: Softmax Decision Boundary on Iris training dataset

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.

Cifar specific code:

```
def flatten(self, x):
    d1, d2, d3, d4 = x.shape
    # for each row, flatten the rest of the dims
    x = x.reshape((d1, -1))
    return x

classes = np.unique(trainLabels)
disp_data = []
for clas in classes:
    class_count = 0
    for i in range(len(trainLabels)):
        if class_count == 3:
            break  # from inner for loop
        if trainLabels[i] == clas:
            tup = (trainFeat[i], trainLabels[i])
            disp_data.append(tup)
            class_count += 1

```

Figure 3: CIFAR10 Samples Images

```
# plot cifar images
rows = 3
cols = 10
scale = 125

fig, axes = plt.subplots(rows, cols, figsize=(11, 9))
curr_ind = 0
for col in range(cols):
    for row in range(rows):
        image, label_index = disp_data[curr_ind]
        curr_ind += 1
        axes[row][col].set_title('label {}'.format(label_index))
        axes[row][col].imshow(image)
        axes[row][col].axis('off')
plt.savefig('./cifar10')

def loadCIFAR10():
    trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True)
    testset = torchvision.datasets.CIFAR10(root='./data', train=False,
```

```
    download=True)
    trainFeat = trainset.train_data
    trainLabels = np.asarray(trainset.train_labels)
    testFeat = testset.test_data
    testLabels = np.asarray(testset.test_labels)
    return trainLabels, trainFeat, testLabels, testFeat
```

Listing 2: Python example

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 test accuracy was: 40.3% (with the last accuracy being 40.0%)`

`The softmax classifier on the CIFAR data did not work that well in general. The dataset was`

The best hyperparameters used were:

epochs = 100
learning rate = 0.0001 tried [0.1, 0.01, 0.001. 0.0001, 0.000001]
batch size = 8 tried powers of 2
regularization = 0.01 L2 weight decay, range [1, 0.1, 0.01, 0.001]
momentum = 0.10 started with 0 to 1

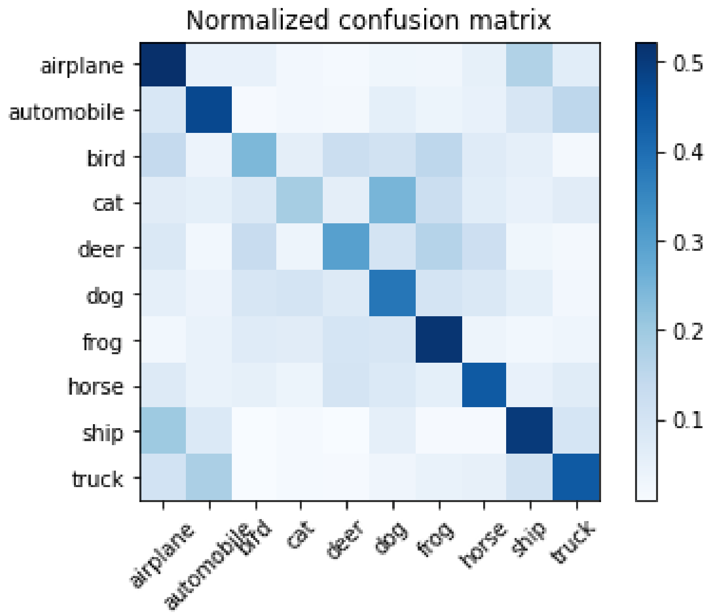

Figure 4: CIFAR10 Normalized Confusion Matrix

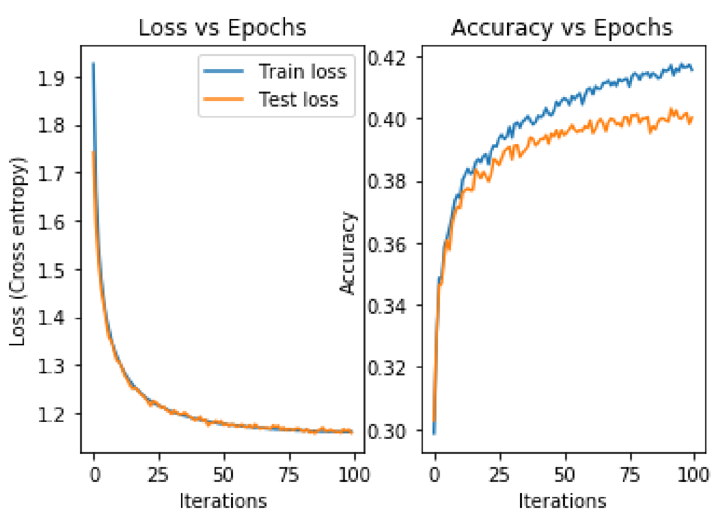

Figure 5: CIFAR10 Loss and Accuracy

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.

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?

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

Solution:

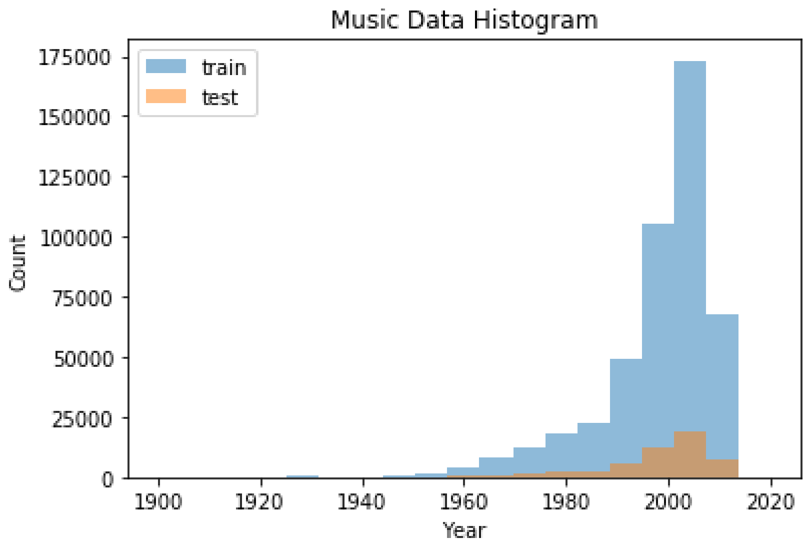

Figure 6: Music Data Histogram of Data Counts vs Year

The range of the x feature values is very large, ranging from -14,000 to 65,000, with a wide range in scales too, from 1000s to decimals, so we'll need to normalize. I normalized the features by subtracting the mean (feature-wise) and dividing by the standard deviation (feature-wise) of the training data, and applying those same statistics to the test data.

For the y labels, they're in years, from 1922-2011, and roughly the same in the test, though slightly wider range. They're not identical ranges, so that is necessary to keep in mind.

The 90's and 2000's are much more over represented than the rest of the years. The early 1900's are very underrepresented. In general, the earlier the music, the lower the count.

The most common year, 2007 had a loss = 193.87.
The year 1998 had a loss = 119.82

```
def normalize_feat(self, x, mean=None, std=None):
    # normalize the feature data.  test data must pass mean and std
    # calc feature-wise mean
    if mean is None:
        mean = np.mean(x, axis=0)
    # calc feature-wise std
    if std is None:
        std = np.std(x, axis=0)
```

```python
    # sub the mean per column
    x_norm = x - mean
    # div by the standard dev.
    x_norm = x_norm / std
    return x_norm, mean, std

def load_data(self, fname, bias=1):
    data = loadtxt(fname, delimiter=',')
    # loads data, normalizes, and appends a bias vector to the data
    TRAIN_NUM = 463714  # training data up to this point
    # process training data
    x_train = data[:TRAIN_NUM,1:].astype(float)  # parse train
    x_train, train_mean, train_std = self.normalize_feat(x_train)  # normalize data
    # create a col vector of ones
    col_bias = np.ones((x_train.shape[0], 1))
    # append bias with hstack
    x_train = np.hstack((x_train, col_bias))
    # convert label vals to int and to vector
    y_train = data[:TRAIN_NUM,0].astype(int)
    y_train = y_train.reshape((-1, 1))
    # ----------------------
    # process test data
    x_test = data[TRAIN_NUM:,1:].astype(float)  # parse test
    x_test, _, _ = self.normalize_feat(x_test, train_mean, train_std)  # normalize data
    # create a col vector of ones
    col_bias = np.ones((x_test.shape[0], 1))
    # append bias with hstack
    x_test = np.hstack((x_test, col_bias))
    # convert label vals to int and to vector
    y_test = data[TRAIN_NUM:,0].astype(int)
    y_test = y_test.reshape((-1, 1))  # convert to column vector
    return x_train, y_train, x_test, y_test

def musicMSE(self, pred, gt):
    # make sure to floor by converting to int()
    diff = pred - gt
    mse = (np.square(diff)).mean()
    return mse
```

Listing 3: Python example

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: If you don't use a constant, 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:

Note: see the appendix for ridge regression code.

Results Discussion: For ridge regression to work, I needed to subtract the y label means from all the training and test labels, otherwise the gradients would blow up. With that "trick" I was able to get it to work if using a very low learning rate of about 0.00000001.

Derivative of the Ridge Regression Loss:

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

The pseudo inverse closed form is more accurate as expected, since it doesn't need to tune hyperparameters, but solve directly.

Pseudo inverse closed form MSE: 90.44
Best achieve through iterative: 93.3

```

# note, code might be completed on next page
def closed_form(self, x, yt):
    # yt is regular labels
```

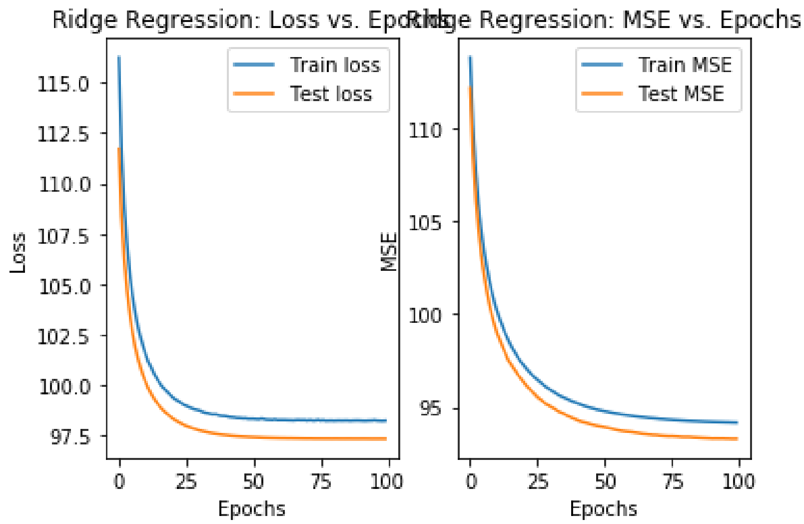

Figure 7: Ridge Regression Training and Evaluation Loss and MSE

```python
    # returns the weights w that allow you to find the prediction
    xt = np.transpose(x)
    alpha_identity = self.alpha * np.identity(len(xt))
    theInverse = np.linalg.inv(np.dot(xt, x) + alpha_identity)
    w = np.dot(np.dot(theInverse, xt), yt)
    return w
```

Listing 4: Python example

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:

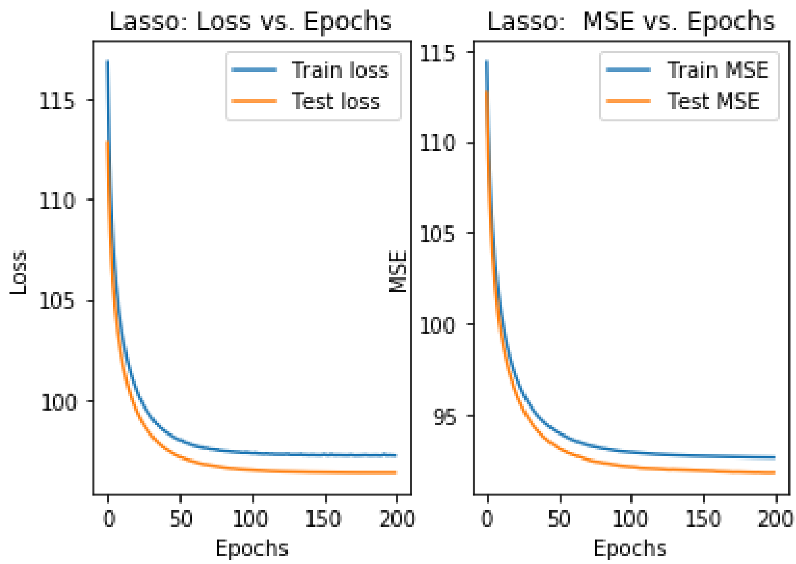

Figure 8: L1 Regression Training and Evaluation Loss and MSE

Results discussion: The L1 weight decay results were slightly better with an MSE of 91.83, vs 93.3 for L2 (Ridge) regression. I had a more difficult time training the L1 regression, and it took longer to converge, so I ran it for twice as long up to 200 epochs. However, the curve is roughly the same as ridge.

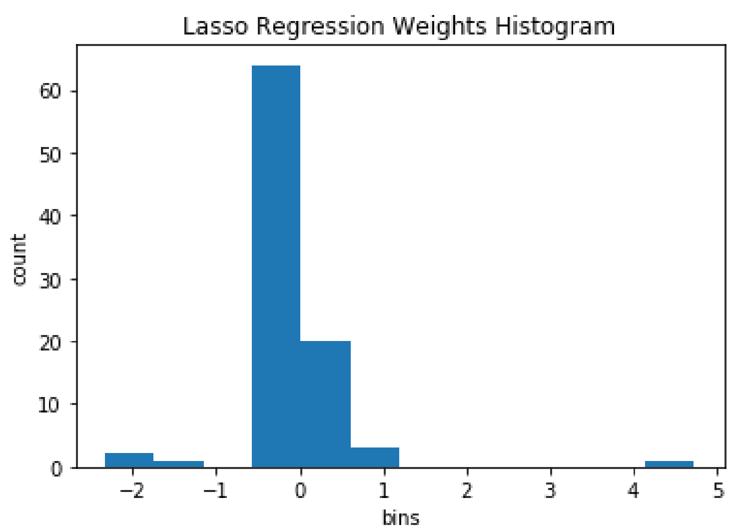

Figure 9: Lasso regression weights histogram

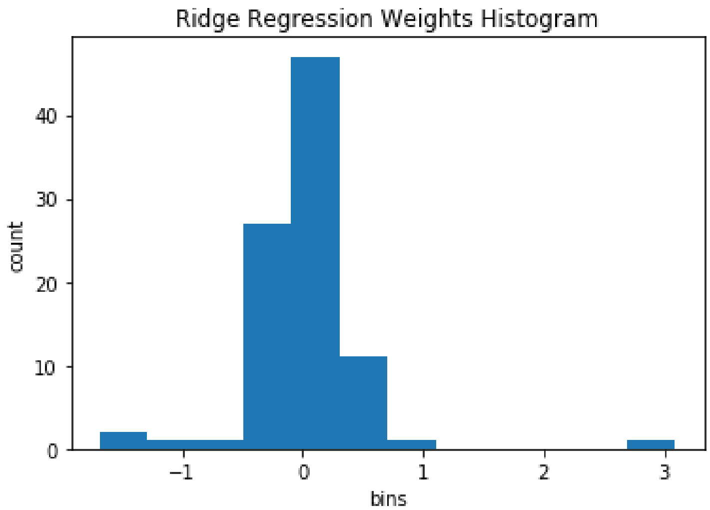

Figure 10: Ridge regression weights histogram

Results comparison: the weights of the Lasso are more consolidated and closer to 0. They seem to be more sparse elsewhere, with some values at the tail ends farther out. However, the L2 weights are slightly more spread around 0 (wider range), and the further values at the tail ends are not as far in magnitude compared to L1. This is due to the property of the L1 norm, which produces sparse coefficients. The sparsity in weights makes it more computationally efficient, since many weights are 0.

Modified code specific for L1 regression:

```
def loss(self, x, yt_sm):
    # calc the cost
    # yt = true label, sub mean label
    n_samples = x.shape[0]
    pred_y = np.dot(x, self.weights)
    residual = np.linalg.norm(pred_y - yt_sm, ord=2, axis=0)
    sq_residual = np.square(residual)
    loss = (sq_residual / n_samples) + self.alpha * np.linalg.norm(self.weights, ord=1, axis=0)
    return loss

def gradient(self, x, yt_sm):
    n_samples = x.shape[0]
    pred_y = np.dot(x, self.weights)
    residual = pred_y - yt_sm
    dW = 2 * (np.dot(x.T, residual) / n_samples) + self.alpha * np.sign(self.weights)
```

```python
    return dW
```

Listing 5: Python example

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:

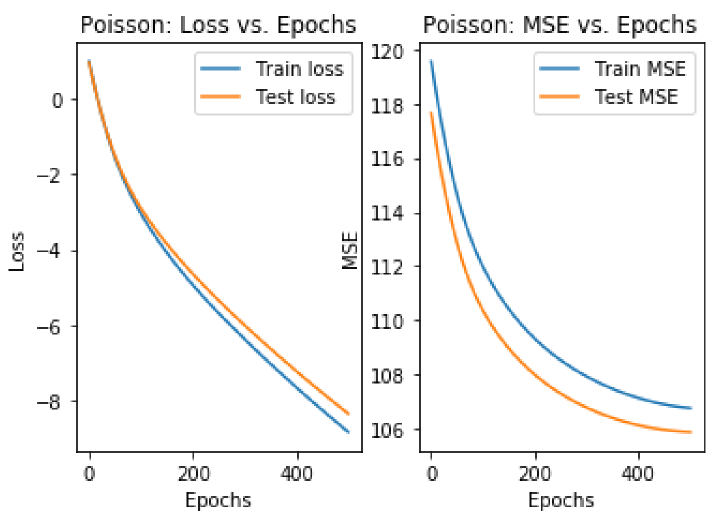

Figure 11: Poisson regression loss and MSE

Results discussion: The Poisson regression was much more difficult to train as the MSE kept "blowing up". I had to use all sorts of hacking tricks to keep the training to behave properly. This involved trying very small learning rates, and keeping not dividing the gradient by the batch size (to get the average), as I had done as usual for the other models. The smaller learning rate made up for this, even though in theory, they are countering each other way. Either I can lower the learning rate, and increase the gradient magnitude, or vice versa. This performance was reflected in my MSE, which I could only get to approximately 112, fairly high.

Poisson Specific code:

```python
def loss(self, x, yt_sm):
    # calc the cost
    # yt = true label, sub mean label
    n_samples = x.shape[0]
    # predict
    pred_y = np.exp(np.dot(x, self.weights))
    # (x dot w)
    x_dot_w = np.dot(x, self.weights)
    # calc y dot times x_dot_w
    x_prod_y = x_dot_w * yt_sm
    # calc the diff, and divide
    loss = np.sum((pred_y - x_prod_y)) / n_samples
    return loss

def gradient(self, x, yt_sm):
    n_samples = x.shape[0]
    y_pred = np.exp(np.dot(x, self.weights))
    dW = np.dot(x.T, (y_pred - yt_sm).reshape(-1)).reshape(-1, 1)
    return dW
```

Listing 6: Python example

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:

To use the softmax classifier on the music data, I had to specifically create new one hot encoding functions to handle all years in the training and testing. Also, I offset the label data to make sure that learning was done properly.

Some pros of treating this as a classification problem are that classification is easier to compute the loss, so it can be done faster. The output state is fixed, which can be a good

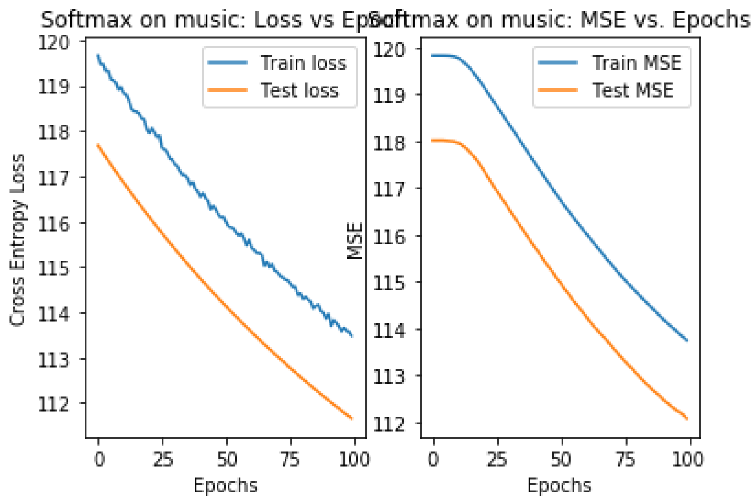

Figure 12: Softmax classifier applied to music data cross entropy loss and MSE

or bad thing depending on the application. However, when you get an incorrect class, there's no relation of getting close to the correct answer by distance, such as 1998 vs 1999. For classification, it can't tell. But for regression, there will be less penalty, relative to getting something wrong by a larger distance, as opposed to a larger confidence in the form of a probability on that class.

Softmax on music data specific code:

```
def calc_mse(self, probs, yt_off):
    preds = np.argmax(probs, 1).reshape(-1, 1)
    diff = preds - yt_off.reshape(-1, 1)
    mse = (np.square(diff)).mean()

def offset_labels(y):
    OFFSET = 1900 # starting the index 0 with year 1923
    return y - OFFSET

def one_hot_vary(y_train, y_test):
    # create one hot on y labels for years
    train_size = len(y_train)
    test_size = len(y_test)

    stacked = np.vstack((y_train, y_test))
    one_h = OneHotEncoder().fit_transform(stacked).toarray()

    y_train = one_h[0:train_size, :]
    y_test = one_h[train_size:, :]
```

```python
    return y_train, y_test  # new one hots
```

Listing 7: Python example

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:

Model Comparison:

Note, results discussions for each individual model were discussed in their sections. The discussion is related to inter-model and inter-data comparisons.

MSE Performance:
Ridge: 93.3
Lasso: 91.8
Poisson: 112.3
Softmax Classifier: 110.1

For each of the models, it was interesting to note that the only to get them to train was to subtract the label mean from each label. This allowed the model to predict something in the range of the true label, but not have the loss too large as to have the gradient explode. Even with trying to use a small learning rate was not enough to counter to avoid this.

The largest error was on the Poisson regression, and it proved to be the most difficult to train as well. The regression by learning the count seemed to be distinctly perform different than the other models. The softmax classifer also performed poorly as well, and this was the most unique of all the "regression" models.

Across all the models, the 2000's years performed the best (had the most true positives), which is due to the fact that most of the training data is around these years, so the model is expected to predict something near this range. The higher count in these years "nudges" the model closer to these range each time it sees a sample from this range.

Model specific comparisons:

1. The ridge regression model on the music data performed reasonably well with an MSE of 93.3, second of all the regression models I trained. Having the L2 regularization was tricky in getting the gradient and loss to be in reasonable range. This was important because

initially the gradients kept exploding, and I needed to try a wider range of hyperparameters, especially the learning rate, which I brough down very low. The square term on the loss can rapidly cause the weights to increase. The L2 norm also causes the model to try and fit outliers more, and therefore, is more sensitive to data with more variance.

This model was the first "difficult" model to train in the assignment, where tuning the hyperparameters was extremely important, and I could see the effect of each change.

2. The L1 Lasso regression model performed the best of all the regression with an MSE of 91.8. One possible explaination for the better performance is the fact the L1 norm is much gentler on the outliers, allowing the model to converge easier compared to L2 ridge regression. It did converge slower, since the gradient took smaller steps.

3. The Poisson regression model was one of the more difficult models to train. The regression by counting prediction kept having the behavior of first decreasing the MSE, and then increasing MSE, even though the loss was going down constantly (into negative values). This was remedied by removing the division of the gradient calculation, and increasing the learning rate. Doing the reverse of this, dividing the gradient and lowering the learning rate, did not work for me. This is a strange behavior of the Poisson model that I could not fully rationalize, even with discussion from other students. The MSE could only go down to 112, relatively high compared to the other models.

4. The softmax classifier on the music data, which as a relatively large dataset, was not as effective as the linear regression models. It was second most difficult model to train, as the losses were not behaving properly when trying to train and debug. It was flatlining while I debugged for 6 hours, likely due to the requirement of having to one hot encode in a specialized way across the training and test data sets. I needed to encode for the entire range of both data sets. It had the largest error across all the models. The MSE could only get down to approximately 110, where as the linear regression models could achieve an MSE of 91.8. This is likely due to the fact that each class, or year, retains no relation in terms of distance. Given a correct class of say 2003, a wrong prediction of 1998 is the same as a wrong prediction of 2008, even though one year is closer to the correct class.

Softmax Classifier Code Appendix

Softmax on music data specific code:

```
def __init__(self, epochs, learning_rate, batch_size, regularization,
    momentum):
    self.epochs = epochs
    self.learning_rate = learning_rate
    self.batch_size = batch_size
    self.regularization = regularization
```

```
    self.momentum = momentum
    self.velocity = None
    self.weights = None

def one_hot(self, y):
    # get a vector of labels, convert into 1 hot
    num_classes = 3  # needs to be fixed
    y = np.asarray(y, dtype='int32')  # convert type to int
    y = y.reshape(-1)  # convert into a list of numbers
    y_one_hot = np.zeros((len(y), num_classes))  # init shape of len y, and out 3 (num of classes)
    y_one_hot[np.arange(len(y)), y] = 1  # set the right indexes to 1, based on y (a list)
    return y_one_hot  # shape N by num_classes (3)

def calc_accuracy(self, x, y):
    #  predict the class, then compare with the correct label.  return the average correct %
    f = x.dot(self.weights)
    probs = self.softmax(f)
    pred = np.argmax(probs, 1)  # predict
    pred = pred.reshape((-1, 1))  # convert to column vector
    return np.mean(np.equal(y, pred))  # return average over all the 1's (over the total)

def softmax(self, x):
    # calc the softmax
    exp_x = np.exp(x - np.max(x))  # make sure it doesn't blow up by sub max

    # make sure sum along columns, and keep dims keeps the exact same dim when summing
    # ie keep cols, instead of converting to rows
    y = np.sum(exp_x, axis=1, keepdims=True)
    return exp_x / y

def loss_and_gradient(self, x, y):
    # calc the loss and gradient.  forward prop, get softmax, calc the neg loss loss, and total loss.
    # calc dW by taking the residual, then dot with X,  + regularization
    # find average for both
    n_samples = x.shape[0]  # num of examples
    # forward prop
    f = np.dot(x, self.weights)  # mult X by W
    probs = self.softmax(f)  # pass f to softmax

    # take neg log of the highest prob. for that row
    neg_log_loss = -np.log(probs[np.arange(n_samples), np.argmax(probs, axis=1)])
    # neg_log_loss = -np.log(probs[np.arange(n_samples), y])
    loss = np.sum(neg_log_loss)  # sum to get total loss across all samples
```

```
        # calc the regularization loss too
        reg_loss = 0.5 * self.regularization * np.sum(self.weights * self.weights)
        total_loss = (loss / n_samples) + reg_loss  # sum to get total, divide for avg

        # calc dW
        y_one_hot = self.one_hot(y)  # need one hot
        # calc derivative of loss (including regularization derivative)
        dW = x.T.dot( (probs - y_one_hot) ) + (self.regularization * self.weights)
        dW /= n_samples  # compute average dW
        return total_loss, dW

    def train_phase(self, x_train, y_train):
        # shuffle data together, and forward prop by batch size, and add momentum
        num_train = x_train.shape[0]
        losses = []
        # Randomize the data (using sklearn shuffle)
        x_train, y_train = shuffle(x_train, y_train)

        # get the next batch (loop through number of training samples, step by batch size)
        for i in range(0, num_train, self.batch_size):
            # grab the next batch size
            x_train_batch = x_train[i:i + self.batch_size]
            y_train_batch = y_train[i:i + self.batch_size]

            # forward prop
            loss, dW = self.loss_and_gradient(x_train_batch, y_train_batch)  # calc loss and dW
            # calc velocity
            self.velocity = (self.momentum * self.velocity) + (self.learning_rate * dW)
            self.weights -= self.velocity  # update the weights
            losses.append(loss)  # save the losses
        return np.average(losses)  # return the average

    def test_phase(self, x, y_test):
        # extra, but more explicit calc of loss and gradient during testing (no back prop)
        loss, _ = self.loss_and_gradient(x, y_test)  # calc loss and dW (don't need)
        return loss

    def run_epochs(self, x_train, y_train, x_test, y_test):
        # start the training/valid by looping through epochs
        num_dim = x_train.shape[1]  # num of dimensions
        n_classes = 3  # num output
        # create weights array/matrix size (num features x output)
```

```python
    self.weights = 0.001 * np.random.rand(num_dim, n_classes)
    self.velocity = np.zeros(self.weights.shape)

    # store losses and accuracies here
    train_losses = []
    test_losses = []
    train_acc_arr = []
    test_acc_arr = []

    for e in range(self.epochs): # loop through epochs
        # print('Ephoch {} / {}...'.format(e + 1, self.epochs))

        # calc loss and accuracies
        train_loss = self.train_phase(x_train, y_train)
        test_loss = self.test_phase(x_test, y_test)
        train_acc = self.calc_accuracy(x_train, y_train)
        test_acc = self.calc_accuracy(x_test, y_test)

        # append vals to lists
        train_losses.append(train_loss)
        test_losses.append(test_loss)
        train_acc_arr.append(train_acc)
        test_acc_arr.append(test_acc)
    return train_losses, test_losses, train_acc_arr, test_acc_arr  # return all the vals

def plot_graph(self, train_losses, test_losses, train_acc, test_acc):
    # plot graph
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label="Train loss")
    plt.plot(test_losses, label="Test loss")
    plt.legend(loc='best')
    plt.title("Softmax: Loss vs Epochs")
    plt.xlabel("Iterations")
    plt.ylabel("Loss (Cross entropy)")

    plt.subplot(1, 2, 2)
    plt.plot(train_acc, label="Train Accuracy")
    plt.plot(test_acc, label="Test Accuracy")
    # plt.legend(loc='best')
    plt.title("Softmax: Accuracy vs Epochs")
    plt.xlabel("Iterations")
    plt.ylabel("Accuracy")
    plt.show()

def make_mesh_grid(self, x, y, h=0.02):
    # make a mesh grid for the decision boundary
    x_min, x_max = x[:, 0].min() - 1, x[:, 0].max() + 1
    y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1
```

```
    x_x, y_y = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max
, h))
    return x_x, y_y  # matrix of x-axis and y-axis

def plot_contours(self, plt, x_x, y_y, **params):
    # plot contours
    array = np.array([x_x.ravel(), y_y.ravel()])
    f = np.dot(array.T, self.weights)
    prob = self.softmax(f)
    Q = np.argmax(prob, axis=1) + 1
    Q = Q.reshape(x_x.shape)
    plt.contourf(x_x, y_y, Q, **params)  # takes in variable number of params

def plot_decision_boundary(self, x, y):
    # plot decision boundary
    markers = ('o', '.', 'x')
    colors = ('yellow', 'grey', 'green')
    cmap = ListedColormap(colors[:len(np.unique(y))])
    x_x, y_y = self.make_mesh_grid(x, y)
    self.plot_contours(plt, x_x, y_y, cmap=plt.cm.coolwarm, alpha=0.8)

    # plot training points
    for idx, cl in enumerate(np.unique(y)):
        xBasedOnLabel = x[np.where(y[:,0] == cl)]
        plt.scatter(x=xBasedOnLabel[:, 0], y=xBasedOnLabel[:, 1], c=cmap(idx)
,
                    cmap=plt.cm.coolwarm, marker=markers[idx], label=cl)
    plt.xlim(x_x.min(), x_x.max())
    plt.ylim(y_y.min(), y_y.max())
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.title("Decision Boundary - Softmax Classifier")
    plt.legend(loc='upper left')
    plt.show()

def load_data():
    # load data
    train_data = loadtxt('iris-train.txt')
    x_train = train_data[:,1:]
    y_train = train_data[:,0].astype(int)-1  # make sure to minus 1 for label
    y_train = y_train.reshape((-1, 1))  # convert to column vector

    test_data = loadtxt('iris-test.txt')
    x_test = test_data[:,1:]
    y_test = test_data[:,0].astype(int)-1  # make sure to minus 1 for label
    y_test = y_test.reshape((-1, 1))   # convert to column vector
    return x_train, y_train, x_test, y_test
```

Listing 8: Python example

Linear Models for Regression Code Appendix

Linear model for regression code. Note: I put the linear model for ridge regression here, however, each regression model has specific loss and gradient functions that I put in their respective sections above.

Other Selected code Ridge Regression, used for other regression models too:

```
def __init__(self, feat_dims=0):
    # alpha is weight decay hyperparameter
    self.learning_rate = 0.00001
    self.epochs = 100
    self.batch_size = 10000
    self.feat_dims = feat_dims
    self.output_classes = 1

    # create weights array/matrix size (num features x output)
    self.weights = 0.001 * np.random.rand(self.feat_dims, self.output_classes)
    self.alpha = 0.2  # regularization strength
    self.y_mean = None

def normalize_feat(self, x, mean=None, std=None):
    # normalize the feature data.  test data must pass mean and std
    # calc feature-wise mean
    if mean is None:
        mean = np.mean(x, axis=0)
    # calc feature-wise std
    if std is None:
        std = np.std(x, axis=0)
    # sub the mean per column
    x_norm = x - mean
    # div by the standard dev.
    x_norm = x_norm / std
    return x_norm, mean, std

def load_data(self, fname, bias=1):
    data = loadtxt(fname, delimiter=',')
    # loads data, normalizes, and appends a bias vector to the data
    TRAIN_NUM = 463714  # training data up to this point
    # process training data
    x_train = data[:TRAIN_NUM,1:].astype(float)  # parse train
    x_train, train_mean, train_std = self.normalize_feat(x_train)  # normalize data

    # create a col vector of ones
    col_bias = np.ones((x_train.shape[0], 1))

    # append bias with hstack
    x_train = np.hstack((x_train, col_bias))
```

```
    # convert label vals to int and to vector
    y_train = data[:TRAIN_NUM,0].astype(int)
    y_train = y_train.reshape((-1, 1))

    # ----------------------------------------

    # process test data
    x_test = data[TRAIN_NUM:,1:].astype(float)  # parse test
    x_test, _, _ = self.normalize_feat(x_test, train_mean, train_std)  # normalize data

    # create a col vector of ones
    col_bias = np.ones((x_test.shape[0], 1))

    # append bias with hstack
    x_test = np.hstack((x_test, col_bias))

    # convert label vals to int and to vector
    y_test = data[TRAIN_NUM:,0].astype(int)
    y_test = y_test.reshape((-1, 1))  # convert to column vector
    return x_train, y_train, x_test, y_test

def musicMSE(self, pred, gt):
    # make sure to floor by converting to int()
    diff = pred - gt
    mse = (np.square(diff)).mean()
    return mse

def label_sub_mean(self, label):
    # find the mean
    self.y_mean = np.mean(label)
    # sub mean
    temp = label - self.y_mean
    return temp

def train_loss(self, x, yt_sm):
    # calc the cost
    # yt = true label, sub mean label
    n_samples = x.shape[0]
    pred_y = np.dot(x, self.weights)
    residual = np.linalg.norm(pred_y - yt_sm, ord=2, axis=0)
    sq_residual = np.square(residual)
    loss = (sq_residual / n_samples) + self.alpha * np.square( np.linalg.norm(self.weights, ord=2, axis=0) )
    return loss

def test_loss(self, x, yt_sm):
    # calc the cost at test time
    # yt = true label, is regular label
```

```
    n_samples = x.shape[0]
    # need to add the mean back to label
    yt = yt_sm + self.y_mean

    # predict
    pred_y = np.dot(x, self.weights)

    # need to add the y mean back
    pred_y = pred_y + self.y_mean

    residual = np.linalg.norm(pred_y - yt, ord=2, axis=0)
    sq_residual = np.square(residual)

    loss = (sq_residual / n_samples) + self.alpha * np.square( np.linalg.norm(self.weights, ord=2, axis=0) )
    return loss

def gradient(self, x, yt_sm):
    n_samples = x.shape[0]
    pred_y = np.dot(x, self.weights)
    residual = pred_y - yt_sm
    dW = 2 * (np.dot(x.T, residual) / n_samples) + 2 * self.weights * self.alpha
    return dW

def calc_mse(self, x, y_sm):
    # preprocesses (adds the y_mean back to both x and y, and calls musicMSE)
    # predict
    pred_y = np.dot(x, self.weights)
    # add the y mean to the pred and convert to int to round
    pred_y += self.y_mean

    # convert to int to round
    pred_y = pred_y.astype(int)

    # add the y mean back to the labels
    y_labels = y_sm + self.y_mean
    # convert to int to round
    y_labels = y_labels.astype(int)
    # calc the MSE
    mse = self.musicMSE(pred_y, y_labels)
    return mse, pred_y

def train_phase(self, x_train, y_train_sm):
    # shuffle data together, and forward prop by batch size, and add momentum

    num_train = x_train.shape[0]
    losses = []
    # Randomize the data (using sklearn shuffle)
    x_train, y_train_sm = shuffle(x_train, y_train_sm)
```

```
    # get the next batch (loop through number of training samples, step by batch size)
    for i in range(0, num_train, self.batch_size):
        # grab the next batch size
        x_train_batch = x_train[i:i + self.batch_size]
        y_train_batch_sm = y_train_sm[i:i + self.batch_size]
        # calc loss
        loss = self.train_loss(x_train_batch, y_train_batch_sm)
        dW = self.gradient(x_train_batch, y_train_batch_sm)
        self.weights -= dW * self.learning_rate  # update the weights
        losses.append(loss)  # save the losses
    return np.average(losses)  # return the average

def test_phase(self, x, y_sm):
    # extra, but more explicit calc of loss and gradient during testing (no back prop)
    # calc loss
    loss = self.test_loss(x, y_sm)
    return loss

def run_epochs(self, x_train, y_train_sm, x_test, y_test_sm):
    # start the training/valid by looping through epochs
    # store losses and accuracies here
    train_losses = []
    test_losses = []
    train_mse_arr = []
    test_mse_arr = []

    for e in range(self.epochs): # loop through epochs
        print('Epoch {} / {}...'.format(e + 1, self.epochs))
        # calc loss and accuracies
        train_loss = self.train_phase(x_train, y_train_sm)
        test_loss = self.test_phase(x_test, y_test_sm)
        train_mse, train_preds = self.calc_mse(x_train, y_train_sm)
        test_mse, test_preds = self.calc_mse(x_test, y_test_sm)
        # append vals to lists
        train_losses.append(train_loss)
        test_losses.append(test_loss)
        train_mse_arr.append(train_mse)
        test_mse_arr.append(test_mse)

#         return train_losses, test_losses
    # return all the vals
    return train_losses, test_losses, train_mse_arr, test_mse_arr, test_preds

def closed_form(self, x, yt):
    # yt is regular labels
    # returns the weights w that allow you to find the prediction
    xt = np.transpose(x)
```

```
    alpha_identity = self.alpha * np.identity(len(xt))
    theInverse = np.linalg.inv(np.dot(xt, x) + alpha_identity)
    w = np.dot(np.dot(theInverse, xt), yt)
    return w

def plot_graph(self, train_losses, test_losses, train_mse, test_mse):
    # plot graph
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label="Train loss")
    plt.plot(test_losses, label="Test loss")
    plt.legend(loc='best')
#         plt.title("Ridge Regr: Loss vs. Epochs")
    plt.title("Softmax on music: Loss vs Epoch")
    plt.xlabel("Epochs")
#         plt.ylabel("Loss")
    plt.ylabel("Cross Entropy Loss")

    plt.subplot(1, 2, 2)
    plt.plot(train_mse, label="Train MSE")
    plt.plot(test_mse, label="Test MSE")
    plt.legend(loc='best')
    plt.title("Softmax on music: MSE vs. Epochs")
#         plt.title("Ridge Regr: MSE vs. Epochs")
    plt.xlabel("Epochs")
    plt.ylabel("MSE")
    plt.show()

def make_mesh_grid(self, x, y, h=0.02):
    # make a mesh grid for the decision boundary
    x_min, x_max = x[:, 0].min() - 1, x[:, 0].max() + 1
    y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1
    x_x, y_y = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    return x_x, y_y  # matrix of x-axis and y-axis

def plot_contours(self, plt, x_x, y_y, **params):
    # plot contours
    array = np.array([x_x.ravel(), y_y.ravel()])
    f = np.dot(array.T, self.weights)
    prob = self.softmax(f)
    Q = np.argmax(prob, axis=1) + 1
    Q = Q.reshape(x_x.shape)
    plt.contourf(x_x, y_y, Q, **params)  # takes in variable number of params

def plot_decision_boundary(self, x, y):
    # plot decision boundary
    markers = ('o', '.', 'x')
    colors = ('yellow', 'grey', 'green')
    cmap = ListedColormap(colors[:len(np.unique(y))])
    x_x, y_y = self.make_mesh_grid(x, y)
```

```python
        self.plot_contours(plt, x_x, y_y, cmap=plt.cm.coolwarm, alpha=0.8)

        # plot training points
        for idx, cl in enumerate(np.unique(y)):
            xBasedOnLabel = x[np.where(y[:,0] == cl)]
            plt.scatter(x=xBasedOnLabel[:, 0], y=xBasedOnLabel[:, 1], c=cmap(idx)
    ,
                        cmap=plt.cm.coolwarm, marker=markers[idx], label=cl)
        plt.xlim(x_x.min(), x_x.max())
        plt.ylim(y_y.min(), y_y.max())
        plt.xlabel("x1")
        plt.ylabel("x2")
        plt.title("Decision Boundary - Softmax Classifier")
        plt.legend(loc='upper left')
        plt.show()

    def plot_weights(self):
        plt.hist(self.weights, bins=12)
        plt.xlabel('bins')
        plt.ylabel('count')
        plt.title('Ridge Regression Weights Histogram')
        plt.show()
```

Listing 9: Python example