CS 189 Introduction to Machine Learning
Fall 2017

HW10

This homework is due **Monday, November 6 at 10pm.**

1 Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, "HW[n] Write-Up"
2. Submit all code needed to reproduce your results, "HW[n] Code".
3. Submit your test set evaluation results, "HW[n] Test Set".

After you've submitted your homework, be sure to watch out for the self-grade form.

(a) Before you start your homework, write down your team. Who else did you work with on this homework? List names and email addresses. In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

(b) Please copy the following statement and sign next to it:

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

2 Clip Loss

In lecture, you saw the example of different loss functions like the squared-error loss and the hinge-loss. This question explores a different loss function.

Let $S = \{(x_1, y_1), \dots (x_n, y_n)\}$ be a set of n points sampled i.i.d. from a distribution $\mathscr{D}$. This is the training set with $x_i \in \mathbb{R}^d$ being the features and $y_i \in \{-1, 1\}$ being the labels.

We are thinking about a linear classifier that is going to look at the sign of $w^T x$ to make a decision as to whether the label is $+1$ or -1.

Define the *clip loss* of a linear classifier $w \in \mathbb{R}^d$ as

$$\text{loss}(w^T x, y) = \text{clip}(y w^T x)$$

Where clip is the function

$$\text{clip}(z) = \begin{cases} 1 & \text{if } z < 0 \\ 0 & \text{if } z \geq 1 \\ 1 - z & \text{otherwise.} \end{cases}$$

For any d-dimensional vector w, define the *risk* of w as

$$R[w] = \mathbb{E}_{\mathscr{D}}[\text{loss}(w^T x, y)]\,,$$

and the *empirical risk* of w as

$$R_S[w] = \frac{1}{n} \sum_{i=1}^{n} \text{loss}(w^T x_i, y_i)\,.$$

(a) Draw the clip loss function. **Is the function** clip **convex?** Justify your answer.

Solution: It is not convex. Drawing the function shows that the line from $(-1, 1)$ to $(1, 0)$ lies below the graph of the clip function.

(b) **Interpret the loss function. Why is it reasonable to look at $y w^T x$?**

(c) **Prove that if $R_S[w] = 0$ and $\|w\|_2^2 < 1$, then the hyperplane defined by w has a classification margin greater than 1 on this training set.**

Solution: The margin of the hyperplane is defined as

$$\frac{\min_{1 \leq i \leq n}(y_i(w^T x))}{\|w\|_2^2}$$

If $R_S[w] = 0$, then the numerator is greater than or equal to 1 for all i. Moreover, if $\|w\|_2^2 < 1$, the denominator is less than 1. Hence, the margin is greater than 1.

(d) **Prove that $\mathbb{E}_S[R_S[w]] = R[w]$.** Here, the outer expectation is being taken over the randomly drawn training set.

Solution:

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\text{loss}(w^T x_i, y_i)\right] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\left[\text{loss}(w^T x_i, y_i)\right] = \frac{1}{n}\sum_{i=1}^{n} R[w] = R[w]$$

(e) **Prove that** $\text{Var}(R_S[w]) \leq \frac{1}{n}$.

Solution:

$$\begin{aligned}
\text{Var}(R_S[w]) &= \mathbb{E}\left[(R_S[w] - R[w])^2\right] \\
&= \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbb{E}(\text{loss}(w^T x_i, y_i) - R[w])(\text{loss}(w^T x_j, y_j) - R[w]) \\
&= \frac{1}{n^2}\sum_{i=1}^{n}\mathbb{E}\left[(\text{loss}(w^T x_i, y_i) - R[w])^2\right] \\
&= \frac{1}{n}\mathbb{E}\left[(\text{loss}(w^T x, y) - R[w])^2\right] \\
&\leq \frac{1}{n}
\end{aligned}$$

Here, the first line is the definition of variance, the second line expands the square, the third line follows because (x_i, y_i) and (x_j, y_j) are independent. The fourth line follows because the (x_i, y_i) are identically distributed. The last line follows because the clip loss is nonnegative and bounded above by 1.

Alternate proof of first 4 steps:

$$\begin{aligned}
\text{Var}(R_S[w]) &= \text{Var}(\frac{1}{n}\sum_{i=1}^{n}\text{loss}(w^T x_i, y_i)) \\
&= \frac{1}{n^2}\text{Var}(\sum_{i=1}^{n}\text{loss}(w^T x_i, y_i)) \\
&= \frac{1}{n^2}\sum_{i=1}^{n}\text{Var}(\text{loss}(w^T x_i, y_i), \text{ by i.i.d} \\
&= \frac{1}{n}\text{Var}(\text{loss}(w^T x, y))
\end{aligned}$$

(f) **Is it possible to have an** S **and** w **such that** $R_S[w] = 0$**, but** $R[w] > 0$**?** Justify your answer.

Solution: Yes. Consider the case when $n = 1$. Then it is possible to classify the single data point correctly while classifying all of the opposite class incorrectly.

3 Gradient Descent Methods Convergence

In this problem we will examine the convergence of different gradient descent methods that we have seen so far. We aim to solve a linear least squares problem on the dataset gradient_descent_data.mat. Remember that the linear least squares problem can be written as:

$$\min_w \frac{1}{2}||Aw-y||_2^2 = \min_w \Sigma_{i=1}^n \frac{1}{2}(A_i^\top w - y_i)^2 = \min_w \sum_{i=1}^n \ell_i(w)$$
.

Here $A_i^\top$ denotes the ith row of matrix A and ℓ_i is the loss of the training example i. We implement a variant of SGD, called randomized Kaczmarz SGD as follows:

$$w^t = w^{t-1} - \alpha_J \nabla \ell_J(w^{t-1})$$

with learning rates $\alpha_j = \frac{1}{||A_j^\top||_2^2}$ for each $j = 1, 2, \ldots, n$. Above, J is a random index between 1 and n such that the probability of choosing $J = j$ is given by $p(J = j) = \frac{||A_j^\top||_2^2}{||A||_F^2}$ for $j \in \{1, \ldots, n\}$. The superscript t on the weights w^t tells us which iteration we are considering.

For this problem, only use these python libraries: numpy, matplotlib, scipy.io

(a) **Write code to apply a batch gradient descent algorithm 10 times over all the samples with learning rate $\alpha = 0.05$.** Initialize the weight vector with zeros. **Plot the mean square error as a function of number of iterations.**

Solution:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io

N = 1000
M = 2
num_iter = 10
W = np.zeros((M, num_iter+1))

data = scipy.io.loadmat('gradient_descent_data.mat')
X = data['x']
y = np.squeeze(data['y'])

plt.figure()
plt.xlabel('Number of Iterations')
plt.ylabel('Mean Squared Error')
plt.title("MSE of different gradient descent sampling schemes")

# batch gradient descent
error = np.zeros(num_iter*N)
alpha = 0.05
k = 0
for i in range(num_iter):
    w = W[:,i].reshape(M,)
    error[k:k+N] = np.linalg.norm(y - X.dot(w))**2/N
    wnext = w + alpha * X.T.dot(y - X.dot(w))/N
    W[:,i+1] = wnext.reshape(M,)
    k = k+N
plt.plot(error, label="Batch GD")
```

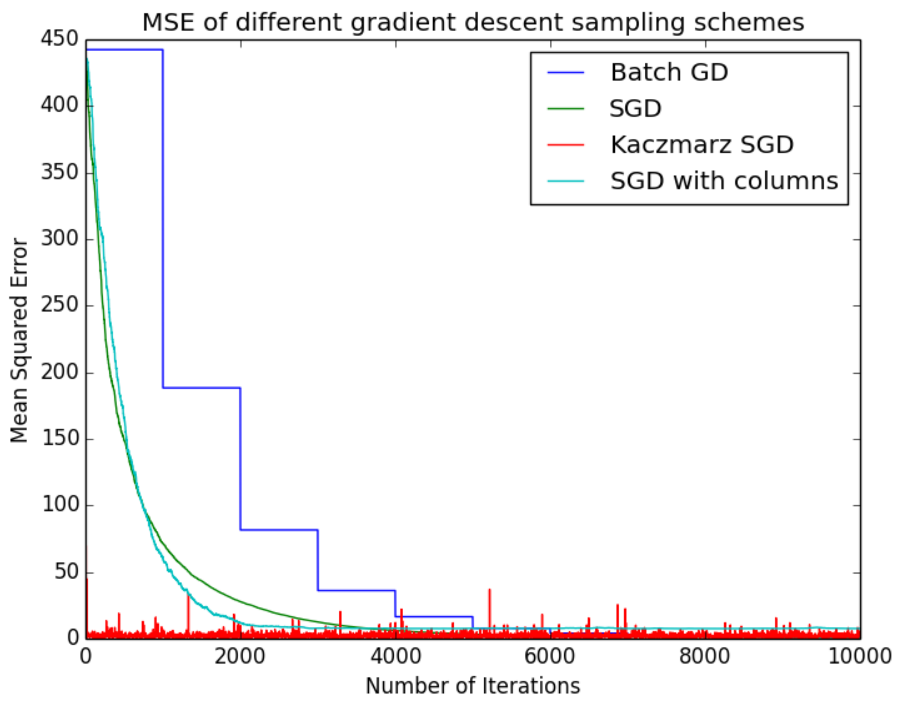


Figure ?? shows the resulting comparison between this part and the subsequent 3 parts.

(b) **Write code to apply a standard stochastic gradient descent algorithm with minibatch size of one with fixed learning rate $\alpha = 5 \times 10^{-5}$.** Initialize the weight vector with zeros. Do a number of iterations that corresponds to 10 times over the samples. **Plot the mean square error as a function of number of iterations. What happens if we increase the learning rate? Compare the results with previous part.** To be fair in our comparisons from a computational perspective, consider each time we have processed n samples to be the equivalent of one iteration in gradient descent. So we have "fractional" iterations here.

Solution:

```
W = np.zeros((M,N*num_iter+1))
error = np.zeros(num_iter*N)
alpha = 0.05/N
k = 0
for j in range(num_iter):
    for i in range(N):
        indx = np.random.choice(range(N))
        w = W[:,k].reshape(M,)
        x = X[indx,:].reshape(1,M)
        error[k] = np.linalg.norm(y - X.dot(w))**2/N
        wnext = w + alpha * x.T.dot(y[indx] - x.dot(w))
        W[:,k+1] = wnext.reshape(M,)
        k = k+1

plt.plot(error, label="SGD")
```

(c) Stochastic gradient descent looks at the training data in small minibatches (or one at a time) instead of looking at the whole big data set each time. Another thing that we could do is to look at the *features* one at a time. This is sometimes called "coordinate descent" in optimization because you work with one coordinate at a time – changing just the weight on that one feature. (Leaving all the other weights unchanged for this iteration.)

Do all three possibilities for coordinate descent for this problem. In each case at each iteration, uniformly pick one of the features and only adjust its weight during this iteration. During each iteration: (1) Do a full least-squares solution, but assuming that the only thing you can change is this single weight (think about how easy least squares is when you only have a single unknown to fit). (2) Do a full gradient descent, but assuming that only this one weight can be changed this iteration. (3) Do stochastic gradient descent using a minibatch size of one, but assuming that for this iteration, only this one weight can be changed.

Write code to uniformly randomly subsample from the columns (the two features). Plot the mean square error as a function of number of iterations for each of these possibilities. Compare the results with the previous parts.

Solution:

As we can see in the plots, randomized Kaczmarz SGD is sensitive to noise and choice of model and will have a high variance.

```
# Stochastic gradient descent with subsampled columns (features)
W = np.zeros((M,N*num_iter+1))
error = np.zeros(num_iter*N)
alpha = 0.05/N
k = 0
for j in range(num_iter):
    for i in range(N):
        indx = np.random.choice(range(N))
        indy = np.random.choice(range(M))
        w0 = W[:,k].reshape(M,)
        w = W[indy,k]
        x = X[indx,indy]
        error[k] = np.linalg.norm(y - X.dot(w0))**2/N
        w0[indy] = w
        wnext = w0 + alpha * x * (y[indx] - x * w)
        W[:,k+1] = wnext.reshape(M,)
        k = k+1

plt.plot(error, label="SGD with columns")
```

(d) **Write code to apply randomized Kaczmarz stochastic gradient descent. Plot the mean square training error as a function of number of iterations. Compare the results with the previous parts.**

Solution:

```
alpha = np.zeros((N,1))
prob = []
Xf = np.linalg.norm(X,'fro')**2
```

```
for i in range(N):
    alpha[i] = 1/(np.linalg.norm(X[i,:].reshape(1,M)))**2
    prob = prob + (1/(alpha[i]*Xf)).tolist()

W = np.zeros((M,N*num_iter+1))
error = np.zeros(num_iter*N)
k = 0
for j in range(num_iter):
    for i in range(N):
        indx = np.random.choice(N, 1, p=prob)
        w = W[:,k].reshape(M,)
        x = X[indx,:].reshape(1,M)
        error[k] = np.linalg.norm(y - X.dot(w))**2/N
        wnext = w + alpha[indx] * x.T.dot(y[indx] - x.dot(w))
        W[:,k+1] = wnext.reshape(M,)
        k = k+1

plt.plot(error, label="Kaczmarz SGD")
```

(e) **Now add some nonlinearity to the data by adding $0.1 \times a_2^3$ to y, where a_2 is the second column of the data matrix A. Redo the previous parts and compare the results.**

Solution:

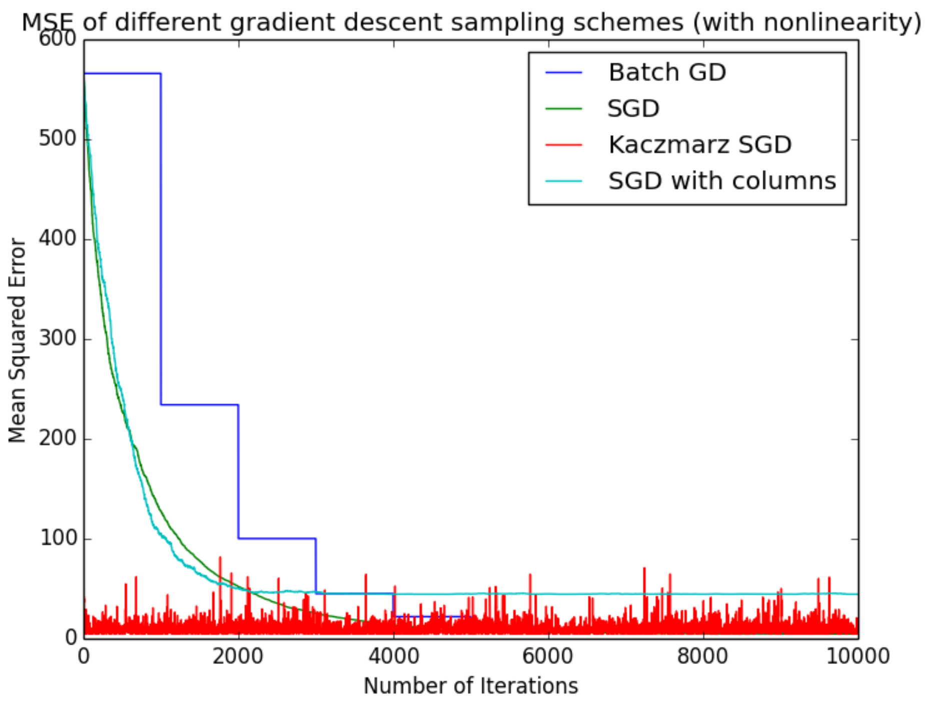
As we can see in the plots, randomized Kaczmarz SGD is sensitive to noise and choice of model and will have a high variance.

```
# Add non-linearity to data
y = y + 0.1 * np.power(X[:,1],3)
plt.title("MSE of different gradient descent sampling schemes (with nonlinearity)
    ")

# Randomized Kaczmarz SGD
alpha = np.zeros((N,1))
prob = []
Xf = np.linalg.norm(X,'fro')**2
for i in range(N):
    alpha[i] = 1/(np.linalg.norm(X[i,:].reshape(1,M)))**2
    prob = prob + (1/(alpha[i]*Xf)).tolist()

W = np.zeros((M,N*num_iter+1))
error = np.zeros(num_iter*N)
k = 0
for j in range(num_iter):
    for i in range(N):
        indx = np.random.choice(N, 1, p=prob)
        w = W[:,k].reshape(M,)
        x = X[indx,:].reshape(1,M)
        error[k] = np.linalg.norm(y - X.dot(w))**2/N
        wnext = w + alpha[indx] * x.T.dot(y[indx] - x.dot(w))
        W[:,k+1] = wnext.reshape(M,)
        k = k+1

plt.plot(error, label="Kaczmarz SGD")
```



4 Linear Methods on Fruits and Veggies

In this problem, we will use the dataset of fruits and vegetables that was collected in HW5 and HW6. The goal is to accurately classify the produce in the image. Instead of operating on the raw pixel values, we operate on extracted HSV histogram features from the image. HSV histogram features extract the color spectrum of an image, so we expect these features to serve well for distinguishing produce like bananas from apples. Denote the input state $x \in \mathbb{R}^{729}$, which is an HSV histogram generated from an RGB image with a fruit centered in it. Each data point will have a corresponding class label, which corresponds to their matching produce. Given 25 classes, we can denote the label as $y \in \{0, ..., 24\}$.

Better features would of course give better results, but we chose color spectra for an initial problem for ease of interpretation.

Classification here is still a hard problem because the state space is much larger than the amount of data we obtained in the class – we are trying to perform classification in a 729 dimensional space with only a few hundred data points from each of the 25 classes. In order to obtain higher accuracy, we will examine how to perform hyper-parameter optimization and dimensionality reduction. We will first build out each component and test on a smaller dataset of just 3 categories: apple, banana, eggplant. Then we will combine the components to perform a search over the entire dataset.

Note all python packages needed for the project, will be imported already. **DO NOT import new Python libraries.**

(a) Before we classify our data, we will study how to reduce the dimensionality of our data. We will project some of the dataset into 2D to visualize how effective different dimensionality reduction procedures are. The first method we consider is a random projection, where a matrix is randomly created and the data is linearly projected along it.

For random projections **fill in the function get_rand_proj in projection.py** to produce a matrix, $A \in \mathbb{R}^{2\times 729}$ where each element A_{ij} is sampled independently from a normal distribution (i.e. $A_{ij} \sim N(0,1)$). **Run the code projection.py and report the resulting plot of the projection.**

Solution:

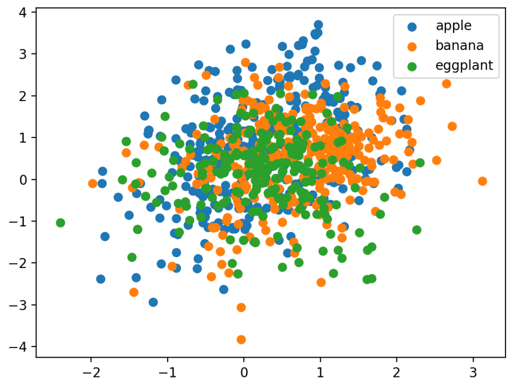


Figure 1: A random projection of the high-dimensional HSV Histograms

See code for solution. You plot may look different based on the random values you generated, but the message to take away is similar.

(b) We will next examine how well PCA performs as a dimensionality-reduction tool. PCA projects the data into the subspace with the most variance, which is determined via the covariance matrix Σ_{XX}. We can compute the principal components via the singular value decomposition $U\Lambda U^T = \Sigma_{XX}$. Collecting the first two column vectors of U in the matrix U_2, we can project our data as follows:

$$\bar{x} = U_2^T x$$

Fill in the function get_pca_proj. Note, you can use the functions imported frum utils to compute the covariance matrix from data and perform mean subtraction.

Solution:

See code for solutions

(c) Finally, we will project our data into the Canonical Variates. In order to perform CCA, we must first turn our labels y into a one-hot encoding vector $\bar{y} \in \{0,1\}^J$, where each element corresponds to the label. Note J is the number of class labels, which is $J = 3$ for this part. Next we need to compute the canonical correlation matrix $\Sigma_{XX}^{-\frac{1}{2}}\Sigma_{XY}\Sigma_{YY}^{-\frac{1}{2}}$ and compute the singular value decomposition $U\Lambda V^T = \Sigma_{XX}^{-\frac{1}{2}}\Sigma_{XY}\Sigma_{YY}^{-\frac{1}{2}}$.

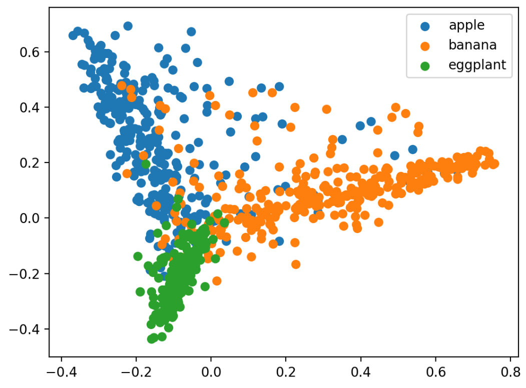


Figure 2: A PCA projection of the high-dimensional HSV Histograms

We can then project to the canonical variates by using the first k columns in U, or U_k. The projection can be written as follows:

$$\bar{x} = U_k^T \Sigma_{XX}^{-\frac{1}{2}} x$$

Fill in get_cca_proj and report the resulting plot for $k = 2$. Note you can use the given function create_one_hot_label.

Solution:

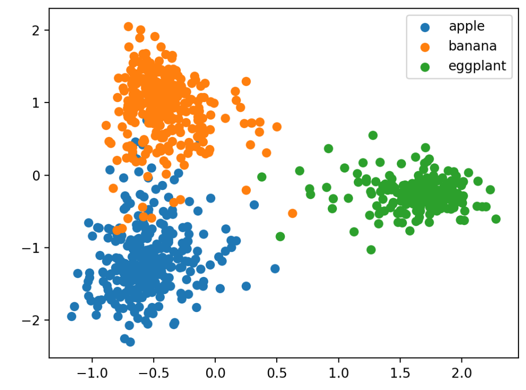


Figure 3: A CCA projection of the high-dimensional HSV Histograms

See code for solutions.

(d) We will now examine ways to perform classification using this smaller projected space from CCA as our features. One technique is to regress to the class labels and then greedily choose the model's best guess. In this problem, we will use ridge regression to learn a mapping from the HSV histogram features to the one-hot encoding $\bar{y}$ described in the previous problem. Solve the following Ridge Regression problem:

$$\min_{w} \sum_{n=1}^{N} ||\bar{y} - w^T x_n||_2^2 + \lambda ||w||_F^2$$

Then we will make predictions with the following function:

$$y = \underset{j \in 0,..,J-1}{\text{argmax}} \ (w^T x)_j$$

where $(w^T x)_j$ considers the j-th coordinate of the predicted vector. **Implement the skeleton class Ridge_Model**. Note that you are allowed to use the imported SKlearn's Ridge Regression method.

Once your model is implemented **run linear_classification.py**. It will output a confusion matrix, a matrix that compares the actual label to the predicted label of the model. The higher the numerical value on the diagonal, the higher the percentage of correct predictions made by the model, thus the better model. **Report the Ridge Regression confusion matrix for the training data and validation data**.

Solution:

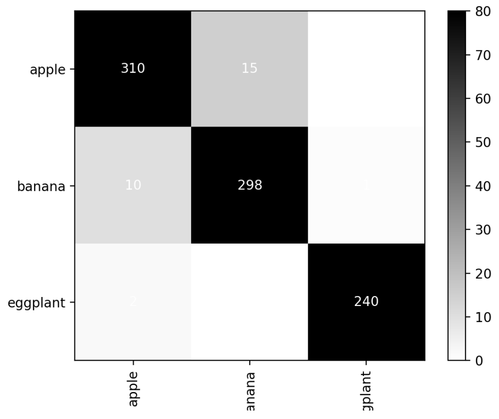


Figure 4: A Confusion Matrix for Training Data with Ridge

See code for solutions.

(e) Instead of performing regression, we can potentially obtain better performance by using algorithms that explicitly model a classification procedure. LDA (Linear Discriminant Analysis) approaches the problem by assuming each $p(x|y = j)$ is a normal distribution with mean μ_j and covariance Σ. Notice that the covariance matrix is assumed to be the same for all the class labels.

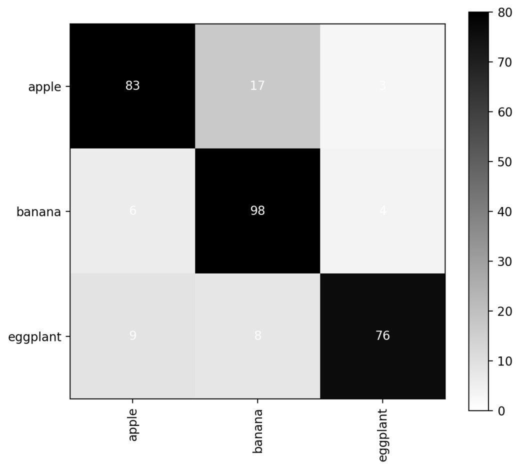


Figure 5: A Confusion Matrix for Validation Data with Ridge

LDA works by fitting μ_j and Σ on the dimensionality-reduced dataset. During prediction, the class with the highest likelihood is chosen.

$$y = \underset{j \in 0,\ldots,J-1}{\operatorname{argmax}} \; -(x-\mu_j)^T \Sigma^{-1} (x-\mu_j)$$

Fill in the class LDA_Model. Then **run linear_classification.py and report the LDA confusion matrix for the training and validation data**.

Solution:

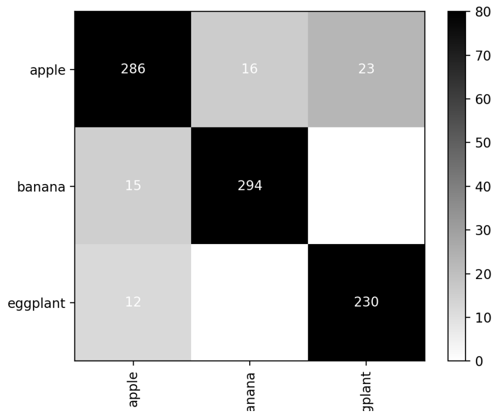


Figure 6: A Confusion Matrix for Training Data with LDA

See code for solutions.

(f) LDA makes an assumption that all classes have the same covariance matrix. We can relax this assumption with QDA. In QDA, we will now parametrize each conditional distribution (still normal) by μ_j and Σ_j. The prediction function is then computed as

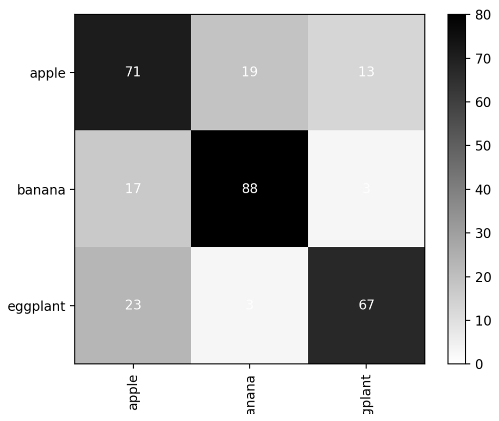


Figure 7: A Confusion Matrix for Validation Data with LDA

$$y = \underset{j \in 0,..,J-1}{\operatorname{argmax}} \; -(x-\mu_j)^T \Sigma_j^{-1}(x-\mu_j) - \log(|\Sigma_j|)$$

Fill in the class QDA_Model. Then **run linear_classification.py and report the QDA confusion matrix for the training data and validation data**.

Solution:

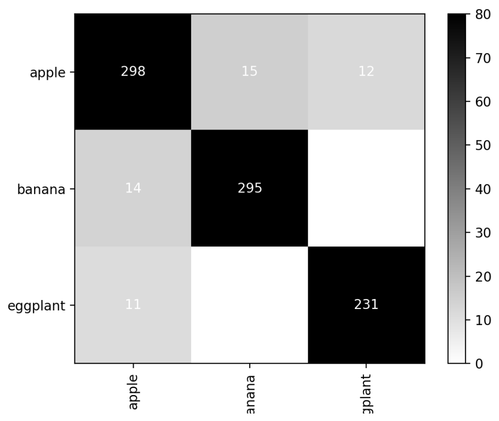


Figure 8: A Confusion Matrix for Training Data with QDA

See code for solutions.

(g) Finally, let us examine the Linear SVM, which fits a hyperplane to separate the dimensionality-reduced data. For this problem, you can use SKlearn's implementation. **Fill in the class SVM_Model**. Then **run linear_classification.py and report the Linear SVM confusion matrix for the training data and validation data**.

Solution:

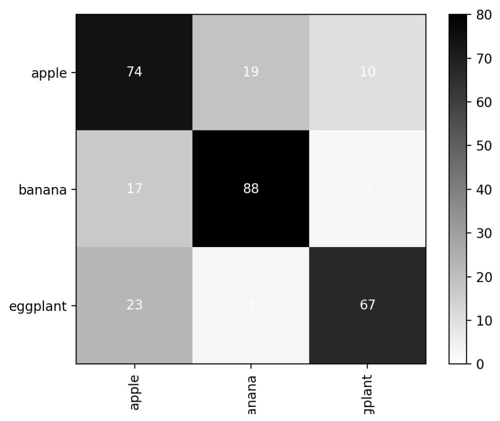


Figure 9: A Confusion Matrix for Validation Data with QDA

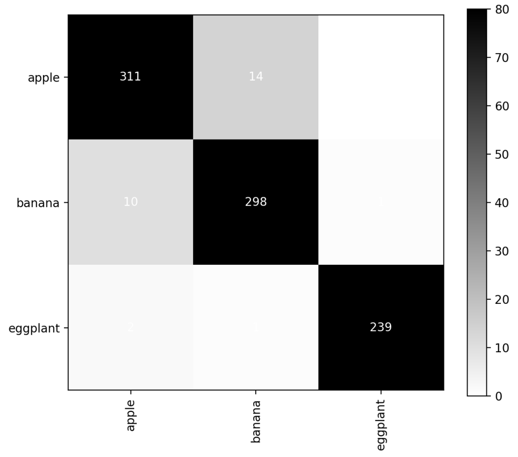


Figure 10: A Confusion Matrix for Training Data with SVM

(h) We will finally train on the full dataset and compare the different models. We will perform a grid search over the following parameters:

(a) The regularization term λ in Ridge Regression.

(b) The weighting on slack variables, C in the linear SVM.

(c) The number of dimensions, k we project to using CCA.

The file hyper_search.py contains the parameters that will be swept over. If the code is correctly implemented in the previous steps, this code should perform a sweep over all parameters and give the best model. **Run hyper_search.py, report the model parameters chosen, report the plot of the models's validation error, and report the best model's confusion matrix for validation data.** WARNING: This can take up to an hour to run.

Solution:

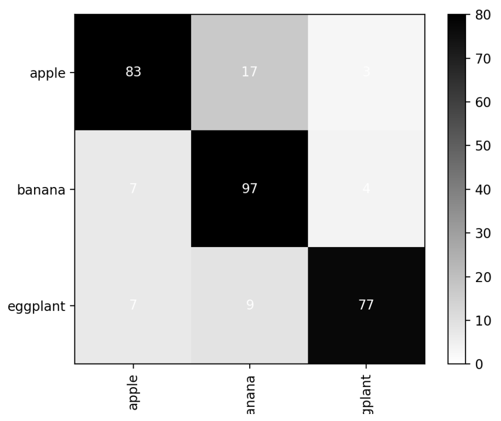


Figure 11: A Confusion Matrix for Validation Data with SVM

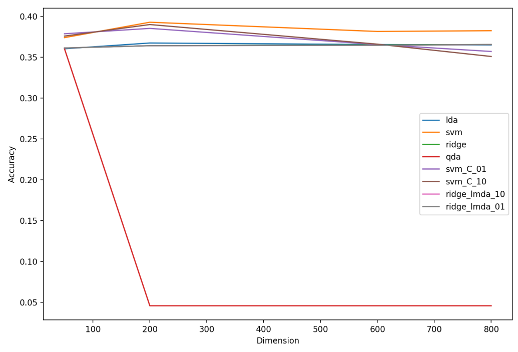


Figure 12: Comparison of Models Across Validation Error

5 Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't

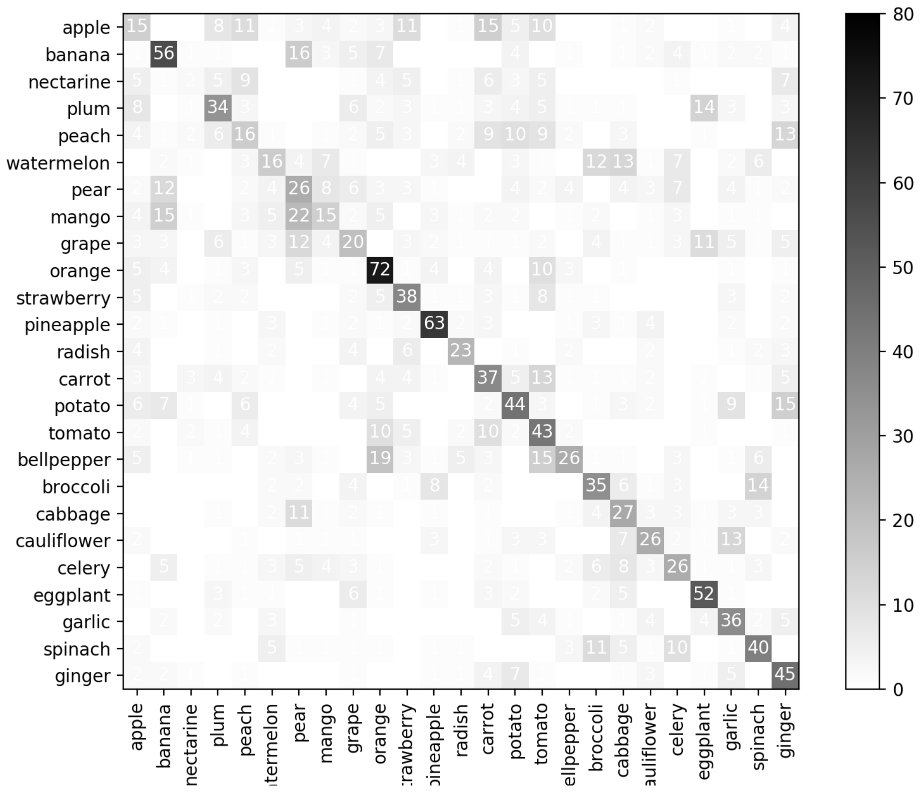


Figure 13: A Confusion Matrix for Validation Data with SVM

have to achieve this every week. But unless you try every week, it probably won't happen ever.