Homework 10: Pet Classification Challenge

Due: 6:00pm, Wednesday, December 7th via Gradescope

Reading: Chapter 10. **Videos:** 10.1 - 10.3

Before we get started, make sure that you have downloaded and decompressed `petdataset.zip` and placed the resulting `catsfolder` and `dogsfolder` into your working MATLAB or Python directory. (You may run into trouble if your working directory is in cloud storage, rather than directly on your hard drive, but hopefully you resolved this issue in Homework 3.)

Let's begin with some basic definitions. We have access to a **dataset** consisting of n examples, $(\underline{X}_1, Y_1), (\underline{X}_2, Y_2), \dots, (\underline{X}_n, Y_n)$. The i^{th} sample $(\underline{X}_i, Y_i)$ consists of an **feature vector** $\underline{X}_i$ (a column vector) as well as a **label** Y_i, which we assume[1] is either 0 or 1. In other words, we get some examples for each hypothesis, and we have to decide what to do when faced with an example $\underline{X}$ that is not in our dataset. These examples are stacked together into an $n \times 4096$ matrix $\mathbf{X}$ and a length-n column vector $\underline{Y}$. The i^{th} row of $\mathbf{X}$ is our i^{th} example written as a row vector $\underline{X}_i^{\mathsf{T}}$ (where superscript T denotes transpose) and the i^{th} entry of $\underline{Y}$ is the label Y_i,

$$\mathbf{X} = \begin{bmatrix} \underline{X}_1^{\mathsf{T}} \\ \underline{X}_2^{\mathsf{T}} \\ \vdots \\ \underline{X}_n^{\mathsf{T}} \end{bmatrix}, \qquad \underline{Y} = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix}.$$

Our overall goal is to build a **binary classification function** $D(\underline{X})$ that takes as input a feature vector $\underline{X}$ and produces as output a guess for the label.

Overfitting. In this problem, we are combine coding ideas from Homeworks 3, 7, and 8 to explore the phenomenon of *overfitting* a dataset. The basic idea is that when we are dealing with high-dimensional datasets, we often have more parameters to fit than the total number of samples. If we are not careful, then our classifier will try to capture the training data perfectly, even if doing so harms its performance more generally.

For example, the pet classification dataset consists of $n = 2000$ samples, each of which is a 64×64 grayscale image, corresponding to $d = 4096 = 64 \times 64$ dimensions. Thus, we have more than twice as many dimensions as samples, and it is easy to overfit the data, even with a "simple" linear classifier. How can we notice and subsequently avoid overfitting in practice?

Training vs. Test Data. It is standard practice in machine learning to (randomly) split the provided dataset into non-overlapping **training data** and **test data.** The training data can be used to design the classifier $D(\underline{x})$ whereas the test data can *only* be used to determine its performance. Overall, we end up with a training dataset

$$(\underline{X}_{\text{train},1}, Y_{\text{train},1}), \dots, (\underline{X}_{\text{train},n_{\text{train}}}, Y_{\text{train},n_{\text{train}}})$$

[1] In binary classification, you will often see $+1$ and -1 labels, including in our videos. However, this homework uses 0 and 1 labels so as to make your previous code easier to reuse.

and a test dataset,

$$(\underline{X}_{\text{test},1}, Y_{\text{test},1}), \dots, (\underline{X}_{\text{test},n_{\text{test}}}, Y_{\text{test},n_{\text{test}}}).$$

We can also organize these into training and test data matrices $\mathbf{X}_{\text{train}}$ and $\mathbf{X}_{\text{test}}$ and training and test label vectors $\underline{Y}_{\text{train}}$ and $\underline{Y}_{\text{test}}$.

Once we have designed our classifier $D(\underline{X})$, we can evaluate its performance on both the training and test sets to get the **training error rate** and the **test error rate**,

$$\text{Training Error Rate} = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} g_{\text{error}}(D(\underline{X}_{\text{train},i}), Y_{\text{train},i}),$$
$$\text{Test Error Rate} = \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} g_{\text{error}}(D(\underline{X}_{\text{test},i}), Y_{\text{test},i})$$

where

$$g_{\text{error}}(y_{\text{guess}}, y_{\text{true}}) = \begin{cases} 1, & \text{if } y_{\text{guess}} \neq y_{\text{true}}, \\ 0, & \text{if } y_{\text{guess}} = y_{\text{true}}. \end{cases}$$

The provided code automatically manages error estimation process for you. Specifically, it first randomly permutes the dataset, then it partitions it into 5 non-overlapping segments, known as **folds.** It then loops over the folds, using the selected fold as test data and the remaining four folds as training data. It then averages the training and test error rates over all five folds, leading to better estimates than a single split.

Dimensionality Reduction. At the full dimension of $d = 4096$, we will often observe that the training error is very close to zero, while the test error remains relatively high. Roughly speaking, this is due to the fact that the classifier has "memorized" the training data so well that the resulting decision rule includes aspects of the training data that do not generalize to the test data. As an analogy, imagine studying for an upcoming exam by perfectly memorizing the numerical answers to the previous exams without understanding the problem-solving process. You would do well if the exam questions are repeated, but poorly otherwise.

To help the classifier avoid overfitting, it is often helpful to reduce the effective problem dimension so that it is smaller than the number of available samples. **Principal component analysis (PCA)** is a powerful method for discovering informative directions in a dataset. For a covariance matrix $\mathbf{\Sigma}$ with eigendecomposition $\mathbf{\Sigma} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{\mathsf{T}}$ where $\mathbf{V}$ is an orthonormal matrix whose columns are the eigenvectors and $\mathbf{\Lambda}$ is a diagonal matrix containing the real eigenvalues. The idea behind PCA is to think of the eigenvectors (the columns of $\mathbf{V}$) as a new basis for our dataset. The eigenvector corresponding to the largest eigenvalue (in absolute value) is the dimension along which the data varies the most. Therefore, one way of reducing the dimension is to only keep the directions corresponding to the k largest eigenvalues.

To keep the top k dimensions, we select the k columns of $\mathbf{V}$ corresponding to the k largest eigenvalues to get a new matrix $\mathbf{V}_k$. We can now project the original input vectors to the subspace spanned by the columns of $\mathbf{V}_k$. Let $\hat{\underline{\mu}}_{\text{train}}$ be the sample mean vector of the training dataset $\mathbf{X}_{\text{train}}$. Given an input data matrix $\mathbf{X}_{\text{run}}$, we first center it by first subtracting the sample mean vector from every row,

$$\mathbf{X}_{\text{centered}} = \mathbf{X}_{\text{run}} - \underline{1}\,\hat{\underline{\mu}}_{\text{train}}^{\mathsf{T}}$$

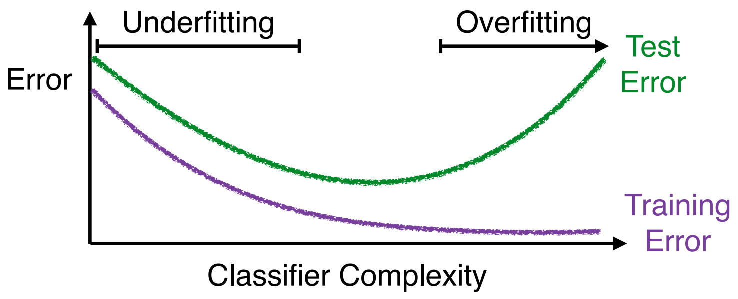
where $\underline{1}$ is a column vector with every entry equal to 1. Next, we multiply by the projection matrix $\mathbf{V}_k$ to obtain our reduced data matrix,

$$\mathbf{X}_{\text{reduced}} = \mathbf{X}_{\text{centered}} \mathbf{V}_k .$$

Problem 10.1 (Video 10.1, 10.2)

Your first task is to fill in the code for the function `dimensionality_reduction` that takes in a data matrix $\mathbf{X}_{\text{run}}$ and returns the k-dimensional PCA-reduced data matrix $\mathbf{X}_{\text{reduced}}$. Note that this task is nearly identical to the 2D data visualization task from Homework 7, except there you only retained the top 2 eigenvectors, rather than the top k. You should be able to adapt your Homework 7 code to this problem.

Our overall goal is to run two classification algorithms and observe the phenomenon of overfitting via a plot. At a high level, we expect to see something like the following illustration. As the complexity of the classifier increases, both the training and test error decrease. However, at a certain point, the test error starts to increase again while the training error continues to decrease. This is the regime where overfitting occurs. In our setting, the classifier complexity will be represented by the dimension k used in the dimensionality reduction algorithm.



For our classification algorithms, we will return to the ideas we explored in Homework 8. That is, we will first fit a Gaussian distribution to the dataset, and then employ the resulting ML rule. As a first step, we will need to estimate the mean vectors and covariance matrices under each label. Specifically, we want

$$n_{\text{train},0} = \#\text{ of 0 labels in } \underline{Y}_{\text{train}} \qquad n_{\text{train},1} = \#\text{ of 1 labels in } \underline{Y}_{\text{train}}$$

$$\underline{\hat{\mu}}_0 = \frac{1}{n_{\text{train},0}} \sum_{i:Y_{\text{train},i}=0} \underline{X}_i \qquad \underline{\hat{\mu}}_1 = \frac{1}{n_{\text{train},1}} \sum_{i:Y_{\text{train},i}=1} \underline{X}_i$$

$$\hat{\mathbf{\Sigma}}_0 = \frac{1}{n_{\text{train},0}-1} \sum_{i:Y_{\text{train},i}=0} (\underline{X}_i - \underline{\hat{\mu}}_0)(\underline{X}_i - \underline{\hat{\mu}}_0)^{\mathsf{T}} \qquad \hat{\mathbf{\Sigma}}_1 = \frac{1}{n_{\text{train},1}-1} \sum_{i:Y_{\text{train},i}=1} (\underline{X}_i - \underline{\hat{\mu}}_1)(\underline{X}_i - \underline{\hat{\mu}}_1)^{\mathsf{T}}$$

Problem 10.2 (Video 10.1, 10.2)

Your second task is to fill in the code for the function `labeled_mean_cov` that takes in a data matrix $\mathbf{X}$, vector of labels $\underline{Y}$, and a desired label. It returns the number of samples n_{label} with that label as well as the corresponding mean vector $\underline{\mu}_{\text{label}}$ and covariance matrix $\mathbf{\Sigma}_{\text{label}}$ under that label. Note that this task is similar to the vector averaging code from Homework 3, except that here you need to first extract the rows of $\mathbf{X}$ corresponding to the desired label. You should feel free to use built-in functions.

In Homework 8, we examined the maximum likelihood (ML) rule for vector Gaussian data. In our current machine learning notation, this approach assume that under the label $Y = 0$, $\underline{X} \sim \mathcal{N}(\underline{\mu}_0, \mathbf{\Sigma}_0)$, and, under the label $Y = 1$, $\underline{X} \sim \mathcal{N}(\underline{\mu}_1, \mathbf{\Sigma}_1)$. Then, the maximum likelihood (ML) decision rule is $D_{\text{ML}}(\underline{x}) = \begin{cases} 1 & f_{\underline{X}|Y}(\underline{x}|1) \geq f_{\underline{X}|Y}(\underline{x}|0) \\ 0 & \text{otherwise} \end{cases}$ where

$$f_{\underline{X}|Y}(\underline{x}|1) = \frac{1}{\sqrt{(2\pi)^d \det(\mathbf{\Sigma}_1)}} \exp\left(-\frac{1}{2}(\underline{x} - \underline{\mu}_1)^\mathsf{T} \mathbf{\Sigma}_1^{-1} (\underline{x} - \underline{\mu}_1) \right)$$
$$f_{\underline{X}|Y}(\underline{x}|0) = \frac{1}{\sqrt{(2\pi)^d \det(\mathbf{\Sigma}_0)}} \exp\left(-\frac{1}{2}(\underline{x} - \underline{\mu}_0)^\mathsf{T} \mathbf{\Sigma}_0^{-1} (\underline{x} - \underline{\mu}_0) \right)$$

We found that, for high-dimensional datasets, we need to use the log-likelihood form of the ML rule to avoid numerical issues:

$$D_{\text{ML}}(\underline{x}) = \begin{cases} 1 & -\frac{1}{2}\Big(\log(\det(\mathbf{\Sigma}_1)) + (\underline{x} - \underline{\mu}_1)^\mathsf{T} \mathbf{\Sigma}_1^{-1} (\underline{x} - \underline{\mu}_1) \Big) \geq -\frac{1}{2}\Big(\log(\det(\mathbf{\Sigma}_0)) + (\underline{x} - \underline{\mu}_0)^\mathsf{T} \mathbf{\Sigma}_0^{-1} (\underline{x} - \underline{\mu}_0) \Big) \\ 0 & \text{otherwise.} \end{cases}$$

Multiplying both sides of the inequality by $-\frac{1}{2}$, this can be equivalently written as

$$D_{\text{ML}}(\underline{x}) = \begin{cases} 1 & \log(\det(\mathbf{\Sigma}_1)) + (\underline{x} - \underline{\mu}_1)^\mathsf{T} \mathbf{\Sigma}_1^{-1} (\underline{x} - \underline{\mu}_1) \leq \log(\det(\mathbf{\Sigma}_0)) + (\underline{x} - \underline{\mu}_0)^\mathsf{T} \mathbf{\Sigma}_0^{-1} (\underline{x} - \underline{\mu}_0) \\ 0 & \text{otherwise.} \end{cases}$$

Linear Discriminant Analysis (LDA). In Homework 8, one of the special cases assumed that the mean vectors $\underline{\mu}_0$ and $\underline{\mu}_1$ were different and that the covariance matrices were the same $\mathbf{\Sigma}_0 = \mathbf{\Sigma}_1 = \mathbf{\Sigma}$. We estimated the shared covariance matrix by pooling our estimates under each label:

$$\hat{\mathbf{\Sigma}} = \frac{1}{n_{\text{train}} - 2}\Big((n_{\text{train},0} - 1)\hat{\mathbf{\Sigma}}_0 + (n_{\text{train},1} - 1)\hat{\mathbf{\Sigma}}_1 \Big) \ .$$

Note that, under this assumption, the $\log(\det(\hat{\mathbf{\Sigma}}))$ terms on either side of the inequality cancel out, and should be removed to avoid numerical issues. It is also possible to cancel out the $\underline{x}^\mathsf{T} \hat{\mathbf{\Sigma}}^{-1} \underline{x}$ terms. In the machine learning literature, the resulting decision is known as Linear Discriminant Analysis (LDA).

Problem 10.3 (Video 10.1, 10.2)

Fill in the code for the `LDA` function. This should be essentially the same as the code you wrote in Problem 8.3(f). Remember that to invert $\hat{\mathbf{\Sigma}}$ you should use the built-in function for the matrix pseudoinverse, not the matrix inverse. (MATLAB: `pinv`, Python: `np.linalg.pinv`) This pseudoinverse should be evaluated outside any of your for loops to avoid slowing down your code.

Quadratic Discriminant Analysis (QDA). In Homework 8, another of the special cases assumed that both the mean vectors $\underline{\mu}_0$ and $\underline{\mu}_1$ and the covariance matrices $\mathbf{\Sigma}_0$ and $\mathbf{\Sigma}_1$ were different. In the machine learning literature, the resulting decision is known as Quadratic Discriminant Analysis (QDA).

We need one more numerical trick to get this to work, even after dimensionality reduction. Recall from linear algebra, that the determinant of a matrix is equal to the product of its eigenvalues. This fact, coupled with the fact that $\log a \cdot b = \log a + \log b$, allows us to avoid computing the determinant directly, and instead calculate

$$\log(\det(\mathbf{\Sigma}_0)) = \sum_i \log(\lambda_{0,i}) \qquad \log(\det(\mathbf{\Sigma}_1)) = \sum_i \log(\lambda_{1,i})$$

where the $\lambda_{0,i}$ are the eigenvalues of $\mathbf{\Sigma}_0$ and the $\lambda_{1,i}$ are the eigenvalues of $\mathbf{\Sigma}_1$.

Problem 10.4 (**Video 10.1, 10.2**)

Fill in the code for the `QDA` function. This should be essentially the same as the code you wrote in Problem 8.3(g), except that you should also use the eigenvalue trick for evaluating the log determinant of the covariance matrices. (If you are using Python, be sure to use the command `np.linalg.eigh` to obtain the eigenvalues. If you instead use `np.linalg.eig`, you will run into issues with complex numbers.)

Finally, we are ready to generate a plot.

Problem 10.5 (**Video 10.1, 10.2**)

Run the overall script to obtain a plot of the training and test error rates for the LDA and QDA classifiers at dimensions $k = 10, 25, 50, 100, 250, 500$. What happens to training error rate as the dimension increases? What happens to the test error rate as the dimension increases? For LDA, determine the values of k that yields the lowest training error and the lowest test error. Record these values of k as well as the resulting training and test error rates. For QDA, determine the values of k that yields the lowest training error and the lowest test error. Record these values of k as well as the resulting training and test error rates.

MATLAB Uploading Instructions: First, fill in all missing lines and values for dimensionality_reduction.m, labeled_mean_cov.m, LDA.m, QDA.m. Then, create a PDF file with your comments and values from 10.5. Let's call this hw10mylastname.pdf. Finally, go to Gradescope, select Homework 10, and upload the following files: dimensionality_reduction.m, labeled_mean_cov.m, LDA.m, QDA.m, hw10mylastname.pdf.

Python Uploading Instructions: First, fill in all missing lines and values for the jupyter notebook hw10python.ipynb. Then, create a PDF file with your comments and values from 10.5. Let's call this hw10mylastname.pdf. Finally, go to Gradescope, select Homework 10, and upload hw10python.ipynb and hw10mylastname.pdf.

Mac Users: Please *do not* upload a zip file containing all your files to Gradescope. This will lead to extra "_MACOSX" files that will slow down our grading. Please just drag and drop your files.

Plagiarism Warning: You are welcome to discuss the assignment with your classmates, but the code you submit should be your own. Please note that Gradescope will run a code similarity detector, which is much more sophisticated than you may realize, especially if you have limited experience with programming. It is extremely effective at identifying code that was copied from an online source or from your classmates.