EK381 Probability, Statistics, and Data Science for Engineers — Boston University
David Castañón — Spring 2020

Homework 10

Due: Sunday, April 26, 11:59 PM Eastern US time
(in your HW 10 Blackboard EK 381 folder. Use the Upload Link)

Instructions:

- Unlike previous assignments, this time you will be required to also include your code in addition to a discussion of the results. For each of Problems 10.1–10.6 include in your solutions you code **with comments** so that graders can understand what you are doing. In addition, list the result on both the training set and the test set, and any additional items each problem asks for.
- Upload your solution in Blackboard.
- If you elect to do the **Extra Credit** please upload a single zip file which, when unzipped, should generate a folder in your name that includes your code for this competition. In the folder include a README file with any notes on how to run the code, either MATLAB (preferred) or python. Test it well before you submit. We will not spend time trying to debug your code. A Python shell script will load your code into MATLAB, run it against the hidden data files, report the scores and enter extra credit if appropriate.

In this homework, we will explore some basic *machine learning* concepts through an image classification case study. Specifically, we will write code for a machine learning pipeline that determines whether a 64×64 grayscale image is a picture of a cat or a dog. You should think of this homework as both a handout that explains the machine learning ideas you will need as well as a problem set to solve. Before we get started, make sure that you have downloaded `pet_classification_dataset.zip` and the provided code (MATLAB or Python) in the HW 10 folder. Decompress the zip file to get `catsfolder`, `dogsfolder`. Place the `catsfolder`, `dogsfolder`, and the provided `*.m` files into your working MATLAB (or Python) directory.

Let's start with some basic definitions. In the framework of detection, we were given two distributions $P_{\underline{X}|H_0}(\underline{x})$ and $P_{\underline{X}|H_1}(\underline{x})$ and used this information to come up with an optimal decision rule to guess the hypothesis if can only observe x. Our goal is to the same thing here, with the important caveat that we *do not know the underlying distributions.* Instead, we have access to a **dataset** $\mathcal{D}$ consisting of n samples,

$$\mathcal{D} = \big\{(\underline{X}_1, Y_1), (\underline{X}_2, Y_2), \ldots, (\underline{X}_n, Y_n)\big\}.$$

The i^{th} sample $(\underline{X}_i, Y_i)$ consists of an **observation vector** $\underline{X}_i$ (a column vector) as well as a **label** Y_i, which we assume[1] is either -1, representing H_0 or $+1$, representing H_1. In other words, we get some examples for each hypothesis, and we have to decide what to do when faced with an observation $\underline{X}$ that is not in our dataset.

[1] We use -1 instead of 0 as a label to simplify some of the linear algebra that follows.

In our context, we should think of $\underline{X}_i$ as the i^{th} image and Y_i denoting whether it is a cat (say -1) or a dog (say $+1$). In total, we have $n = 2000$ images (half of which are dogs, the other half cats). Of course, a grayscale image is a matrix of pixel values, not a vector. So our first task is to transform the provided 64×64 images into length-4096 vectors. If you run `[X, y] = read_data();` (without the semicolon in Python) you will obtain 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} observation 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,

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

This will be a convenient data format for building our classifier, but it is hard to visualize high-dimensional vectors. Therefore, we have provided you with the function `show_image(X,i)` that will extract the i^{th} row of $\mathbf{X}$ and display it as an image. Try it out.

Problem 10.1 Your first task is to fill in the code for the function `average_pet` that takes in a data matrix $\mathbf{X}$ and its labels $\underline{Y}$ and outputs the average cat image (as a length-4096 row vector) and the average dog image (as a length-4096 row vector). Include in your solution your printed code as well as the average cat image and average dog image, which you can display with `show_image`.

Our overall goal is to build a **binary classification function** $h(\underline{X})$ that takes as input an observation vector $\underline{X}$ and produces as output a guess for the label. Ideally, we would like to minimize the probability of error, as we did before for optimal detection. However, we need to be a bit careful as to how we estimate the probability of error. The key idea is that we need to split our dataset into non-overlapping **training data** and **test data.** The training data can be used to design our classifier $h(\underline{X})$ whereas the test data can *only* be used to determine its performance. A reasonable split is to use 80% of our data for training and 20% for testing. We will use these values throughout the homework. Overall, we end up with a training dataset and a test dataset,

$$\begin{aligned} \mathcal{D}_{\text{train}} &= \big\{(\underline{X}_{\text{train},1}, Y_{\text{train},1}), (\underline{X}_{\text{train},2}, Y_{\text{train},2}), \ldots, (\underline{X}_{\text{train},n_{\text{train}}}, Y_{\text{train},n_{\text{train}}})\big\}, \\ \mathcal{D}_{\text{test}} &= \big\{(\underline{X}_{\text{test},1}, Y_{\text{test},1}), (\underline{X}_{\text{test},2}, Y_{\text{test},2}), \ldots, (\underline{X}_{\text{test},n_{\text{test}}}, Y_{\text{test},n_{\text{test}}})\big\}. \end{aligned}$$

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 $\underline{Y}_{\text{test}}$. We have provided you with the function `split_data` that randomly partitions the data matrix and label vector into training and test sets, based on the specified percentage of test data.

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

$$\text{Training Error} = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} g_{\text{not\_equal}}(h(\underline{X}_{\text{train},i}), Y_{\text{train},i}),$$

$$\text{Test Error} = \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} g_{\text{not\_equal}}(h(\underline{X}_{\text{test},i}), Y_{\text{test},i})$$

where

$$g_{\text{not\_equal}}(y_{\text{guess}}, y_{\text{true}}) = \begin{cases} 1, & \text{if } y_{\text{guess}} \neq y_{\text{true}}, \\ 0, & \text{if } y_{\text{guess}} = y_{\text{true}}. \end{cases}$$

It is often more convenient to express these values in terms of the **accuracy**, which is just the fraction of correct guesses expressed as a percentage,

$$\text{Accuracy} = (1 - \text{Error}) \times 100\%.$$

We have provided you the function `calculate_accuracy` that takes in a vector of true labels $\underline{Y}_{\text{true}}$ and a vector of guessed labels $\underline{Y}_{\text{guess}}$ and returns the accuracy.

Note: Since there is an equal number of cats and dogs, we can easily get 50% *accuracy by always guessing cat. If your classifier performs worse than this, something is wrong.*

Recall from linear algebra that the **distance** between vectors $\underline{X}_i$ and $\underline{X}_j$ is $\|\underline{X}_i - \underline{X}_j\|$, where $\|\underline{v}\|$ is the Euclidean (or ℓ_2) norm of some vector $\underline{v} = (v_1, \dots, v_m)$ defined as $\|\underline{v}\| = \sqrt{\sum_{i=1}^{m} (v_i)^2}$. This will be useful for our first classifier, which simply checks whether a vector is closer to the average cat or average dog. Specifically, let $\underline{X}_{\text{avgcat}}$ denote the vector version of the average cat image and $\underline{X}_{\text{avgdog}}$ denote the vector version of the average dog image. The classifier is

$$h_{\text{closest\_average}}(\underline{X}) = \begin{cases} -1, & \text{if } \|\underline{X} - \underline{X}_{\text{avgcat}}\| \leq \|\underline{X} - \underline{X}_{\text{avgdog}}\|, \\ +1, & \text{if } \|\underline{X} - \underline{X}_{\text{avgcat}}\| > \|\underline{X} - \underline{X}_{\text{avgdog}}\|. \end{cases}$$

Problem 10.2 Fill in the code for the `closest_average` function that takes in a training observation matrix $\mathbf{X}_{\text{train}}$, training label vector $\underline{Y}_{\text{train}}$, and test observation matrix $\mathbf{X}_{\text{test}}$, and produces a vector of guesses $\underline{Y}_{\text{guess}}$ for the true labels of $\mathbf{X}_{\text{test}}$. Your average cat and dog images should be obtained only from the training data, and your classifier should apply $h_{\text{closest\_average}}(\underline{X})$ to each row of $\mathbf{X}_{\text{test}}$ to produce its output. Include in your solution your code as well as the training and test accuracy, which you can obtain using the `calculate_accuracy` function.

Another approach is to use a **nearest neighbor** classifier. Given a new observation $\underline{X}$ it searches through the training data to find the index of the closest training vector

$$i_{\text{closest}} = \underset{i=1,\dots,n_{\text{train}}}{\arg\min} \ \|\underline{X} - \underline{X}_{\text{train},i}\|$$

and then returns as its guess the label of this vector, $h_{\text{nearest\_neighbor}}(\underline{X}) = Y_{\text{train},i_{\text{closest}}}$. We can expect this classifier to have 100% training accuracy, since it acts like a lookup table if given the training data. However, this might lead to **overfitting** the data. Additionally, your code may run slowly since we loop through the training matrix for each test vector.

Problem 10.3 Fill in the code for the `nearest_neighbor` function that takes in a training observation matrix $\mathbf{X}_{\text{train}}$, training label vector $\underline{Y}_{\text{train}}$, and test observation matrix $\mathbf{X}_{\text{test}}$, and produces a vector of guesses $\underline{Y}_{\text{guess}}$ for the true labels of $\mathbf{X}_{\text{test}}$. For each row of $\mathbf{X}_{\text{test}}$, it searches through $\mathbf{X}_{\text{train}}$ to find the row with the smallest distance and then outputs its label as the guess. Include in your solution your code as well as the training and test accuracy, which you can obtain using the `calculate_accuracy` function.

A different approach is to use a **linear classifier**. That is, we first take a weighed sum of our observations, and then guess +1 if the outcome is positive and −1 otherwise,

$$h_{\text{linear}}(\underline{X}) = \text{sign}\left(\sum_{\ell=1}^{d} b_\ell X_\ell\right),$$

$$\text{sign}(w) = \begin{cases} -1, & w \leq 0, \\ +1, & w > 0, \end{cases}$$

where we have used X_ℓ to denote the ℓ^{th} entry of the length-d vector $\underline{X}$. This classifier can be expressed in vector notation, which will be more convenient for our implementation,

$$h_{\text{linear}}(\underline{X}) = \text{sign}\left(\underline{X}^\mathsf{T}\underline{b}\right),$$

where $\underline{b}$ are the b_ℓ coefficients stacked into a column vector.

There are many ways to select a good $\underline{b}$. We will pick the *Ordinary-Least-Squares (OLS)* solution that is connected to our prior study of linear estimation. Specifically, we will pick the $\underline{b}$ that minimizes the squared error on the training data,

$$\underline{b}_{\text{ols}} = \arg\min_{\underline{b}} \left\|\underline{Y}_{\text{train}} - \mathbf{X}_{\text{train}}\underline{b}\right\|^2 = \sum_{i=1}^{n_{\text{train}}} \left(Y_{\text{train},i} - \underline{X}_{\text{train},i}^\mathsf{T}\underline{b}\right)^2 .$$

It turns out that this minimization yields the solution

$$\underline{b}_{\text{ols}} = \left(\mathbf{X}_{\text{train}}^\mathsf{T}\mathbf{X}_{\text{train}}\right)^{-1}\mathbf{X}_{\text{train}}^\mathsf{T}\underline{Y}_{\text{train}}.$$

Problem 10.4 Fill in the code for the `linear_regression` function that takes in a training observation matrix $\mathbf{X}_{\text{train}}$, training label vector $\underline{Y}_{\text{train}}$, and test observation matrix $\mathbf{X}_{\text{test}}$, and produces a vector of guesses $\underline{Y}_{\text{guess}}$ for the true labels of $\mathbf{X}_{\text{test}}$. It begins by finding the solution $\underline{b}_{\text{ols}}$ using the training data. (*Note that you need to be careful with the matrix inverse in high dimensions. Specifically, the function* `inv` *will probably declare that the matrix is close to singular, which will lead to numerical issues in subsequent steps. Instead, the function* `pinv` *will handle the inverse more carefully. This function computes what is known as the Moore-Penrose pseudo-inverse, which essentially discards small eigenvalues before computing the inverse. Try both in your implementation to see the difference. In Python, this is* `np.linalg.pinv`.) Afterwards, it uses $\underline{b}_{\text{ols}}$ combined with the `sign` function to make a guess for each label. Include in your solution your code as well as the training and test accuracy, which you can obtain using the `calculate_accuracy` function.

Finally, we will explore one method for reducing the dimensionality of our observations. Although each of these 4096 dimensions may contain useful information, we only have $n = 2000$ samples, and our linear classifier is certainly overfitting the data. Our linear classifier would likely perform better if it had fewer dimensions to deal with. However, simply throwing out some of the pixels is not a great solution either. We need to find a way to capture the most informative "directions" in this 4096-dimensional space.

Principal component analysis (PCA) is a powerful method for discovering informative directions in a dataset. Let's say we had access to the true covariance matrix $\mathbf{K}$ of a random vector $\underline{X}$,

$$\mathbf{K} = \mathsf{E}\Big[\big(\underline{X} - \mathsf{E}[\underline{X}]\big)\big(\underline{X} - \mathsf{E}[\underline{X}]\big)^{\mathsf{T}}\Big].$$

Since this is a symmetric matrix, it has real eigenvalues and it is guaranteed to have an eigendecomposition of the form $\mathbf{K} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{\mathsf{T}}$ 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.

Now, we don't actually have the true covariance matrix $\mathbf{K}$, but we can estimate it from our training data. The built-in function `pca` does exactly this. It takes in a data matrix, estimates the covariance matrix, and then outputs an estimate of the matrix $\mathbf{V}$ whose columns are the eigenvectors as described above.

Problem 10.5 Run the `pca` function on the training data matrix $\mathbf{X}_{\text{train}}$. Using the provided `show_image` function, take a look at the first 10 eigenvectors. Include these images in your solution as well as a description of what you see. *Notice that the output of the `pca` function has the eigenvectors as columns. So to use `show_image` which operates on rows, you will have to take a transpose to extract out an image in the desired format.*

With an estimate of $\mathbf{V}$, we are able to reduce the dimensionality of our data. To keep the top k dimensions, we just select out the first k columns of $\mathbf{V}$ to get a new matrix $\mathbf{V}_k$. As we discussed in class, we can now project the original observation vectors to the subspace spanned by the columns of $\mathbf{V}_k$. All we have to do is to compute inner products of the centered observation vectors (i.e., after subtracting the mean) with the columns of $\mathbf{V}_k$. Letting $\underline{e}_n$ a column vector of n 1's, we can compute the mean observation vector (average pet) directly from the data matrix $\mathbf{X}$ as

$$\underline{\bar{x}} = \frac{1}{N}\mathbf{X}^{\mathsf{T}}\underline{e}_n,$$

and the centered data matrix as

$$\mathbf{X}_c = \mathbf{X} - \underline{e}_n\underline{\bar{x}}^{\mathsf{T}}.$$

Now, we can obtain a reduced data matrix whose i row contains the coordinates of the vector $\underline{X}_i$ along the k columns of $\mathbf{V}_k$. Specifically,

$$\mathbf{X}_{\text{reduced}} = \mathbf{X}_c\mathbf{V}_k = (\mathbf{X} - \underline{e}_n\underline{\bar{x}}^{\mathsf{T}})\mathbf{V}_k. \tag{1}$$

Note that the (MATLAB) built-in function `pca` *can provide you* $\mathbf{X}_{reduced}$ *directly.*

Problem 10.6 Fill in the code for the `pca_regression` function that takes in a training observation matrix $\mathbf{X}_{\text{train}}$, training label vector $\underline{Y}_{\text{train}}$, test observation matrix $\mathbf{X}_{\text{test}}$, and dimension k, and produces a vector of guesses $\underline{Y}_{\text{guess}}$ for the true labels of $\mathbf{X}_{\text{test}}$. It does this by first calculating the k-dimensional PCA transform $\mathbf{V}_k$ from the training data matrix $\mathbf{X}_{\text{train}}$. It then transforms the training and test data to get a reduced-dimension training and test data matrices $\mathbf{X}_{\text{train,reduced}}$ and $\mathbf{X}_{\text{test,reduced}}$, respectively, using the transformation in Eq. (1). Now, you can directly apply the `linear_regression` function that you wrote above to make predictions using these reduced-dimension matrices. Turn in your code as well as the training and test accuracy for dimensions $k = 10, 20, 50, 100$.

Extra Credit. If you finished the above fast, and want to do more of this, then this is your turn to try out more sophisticated machine learning methods to do as well as you can. Feel free to use any built-in functions or libraries. You will turn in a function `pet_classifier` that takes as input an observation matrix $\mathbf{X}$ (in the same format used in this homework, i.e., where rows correspond to different observations) and produces a vector $\underline{Y}_{guess}$ with the predicted labels. This means that you will need to "pre-train" your classifier on the data provided (or any additional training data you manage to generate). I have additional cat and dog images that I will use to assess your classifier's performance. Include your code along with its training and test accuracy on the dataset for this homework. In addition, submit your code for just this problem on blackboard. *If you enter and your code runs and produces a decent accuracy (93% or higher on the test data), you will get 8 bonus points in this problem set. Good luck!*