CMSC 25300/35300, STAT 27700 (Fall 2023)
Homework 2

Submission Instructions:

- Please submit your homework in PDF to Gradescope (which can be accessed from the course's website on Canvas);
- Please paste your code in the submitted PDF. In other words, your submission should be a single PDF that contains both your writing solutions and your code.
- Be sure to include the output of your code when relevant (e.g. text and figures). Keep in mind that we are not easily able to run your code.
- Note that you do not need to copy the problem statements in your solution, *as long as you clearly indicates the problem numbers* (e.g., 1.a, 2.c, etc).

1. Let

$$\boldsymbol{X} = \begin{bmatrix} 1 & 0 & 1 & -1 \\ 3 & 2 & 4 & 0 \\ -1 & 1 & 0 & 3 \\ 2 & 4 & 2 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

From Boyd and Vandenberghe, a collection of vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ (with $k \geq 1$) is *linearly independent* if

$$\alpha_1 \boldsymbol{x}_1 + \cdots + \alpha_k \boldsymbol{x}_k = 0$$

only holds for $\alpha_1 = \cdots = \alpha_k = 0$. Let the columns of $\boldsymbol{X}$ be the vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4 \in \mathbb{R}^5$.

a) (5 points) Are the columns of $\boldsymbol{X}$ linearly independent? In other words, is the set of vectors $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4\}$ linearly independent?

b) (5 points) Find a largest set of linearly independent columns in $\boldsymbol{X}$ and prove that the set is linearly independent. In other words, find a set with as many vectors as possible where all of the vectors are linearly independent.

c) (2 points) Is the set you found in part (b) unique? In other words, is it the only linearly independent set with the largest possible size?

d) (3 points) What is the rank of $\boldsymbol{X}$?

2. Answer the following questions. Make sure to explain your reasoning.

a) (5 points) Are the columns of the following matrix linearly independent?

$$\boldsymbol{X} = \begin{bmatrix} +1.4 & -1.4 \\ -1.4 & +1.4 \\ +1.4 & +1.4 \\ -1.4 & +1.4 \end{bmatrix}$$

b) (5 points) Are the columns of the following matrix linearly independent?

$$\boldsymbol{X} = \begin{bmatrix} 4 & 2 & 0 \\ 5 & 2 & 1 \\ 13 & 8 & -3 \end{bmatrix}$$

c) (5 points) Are the columns of the following matrix linearly independent?

$$\boldsymbol{X} = \begin{bmatrix} -1 & +1 & -1 \\ 0 & -1 & -1 \\ +1 & 0 & 0 \end{bmatrix}$$

d) (5 points) What is the rank of the following matrix?

$$\boldsymbol{X} = \begin{bmatrix} +4 & +6 \\ -6 & +14 \\ +6 & +10 \end{bmatrix}$$

3. Gradients Calculate the gradients of the following functions with respect to $\boldsymbol{w}$:

a) (2 points) $f(\boldsymbol{w}) = (3\boldsymbol{x})^\top \boldsymbol{w} - 2\boldsymbol{w}^\top \boldsymbol{x}$

b) (3 points) $f(\boldsymbol{w}) = (\boldsymbol{w} - 2\boldsymbol{x})^\top (\boldsymbol{w} - \boldsymbol{x})$

c) (3 points) $f(\boldsymbol{w}) = \boldsymbol{x}^\top \begin{bmatrix} 2 & 3 \\ 5 & 7 \end{bmatrix} \boldsymbol{w}$

d) (3 points) $f(\boldsymbol{w}) = \boldsymbol{w}^\top \begin{bmatrix} 1 & 4 \\ 4 & 16 \end{bmatrix} \boldsymbol{w}$

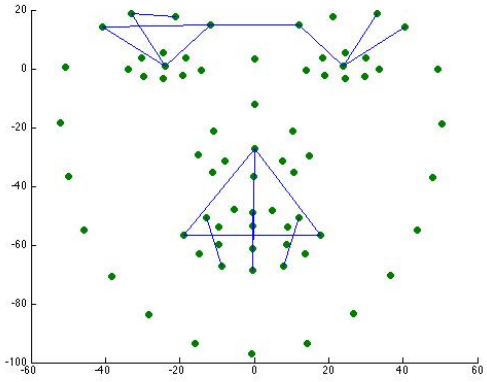
e) (4 points) $f(\boldsymbol{w}) = \boldsymbol{w}^\top \begin{bmatrix} 2 & 3 \\ 4 & -1 \end{bmatrix} \boldsymbol{w}$

4. Design classifier to detect if a face image is smiling.

Consider the two faces below. It is easy for a human, like yourself, to decide which is smiling and which is not. Can we get a machine to do it?


The key to this classification task is to find good features that may help to discriminate between smiling and non-smiling faces. What features do we pay attention to? The eyes, the mouth, maybe the brow?

The image below depicts a set of points or "landmarks" that can be automatically detected in a face image (notice there are points corresponding to the eyes, the brows, the nose, and the mouth). The distances between pairs of these points can indicate the facial expression, such as a smile or a frown. We chose $p = 9$ of these distances as features for a classification algorithm. The features extracted from $n = 128$ face images (like the two shown above) are stored in the $n \times p$ matrix $\mathbf{X}$ in the NumPy file `face_emotion_data.npz` (which you can find on the Canvas assignment post for HW2). This file also includes an $n \times 1$ vector $\mathbf{y}$; smiling faces are labeled $+1$ and non-smiling faces are labeled -1. The goal is to find a set of weights for the features in order to predict whether the "emotion" of a face image is smiling or non-smiling.



a) (6 points) Use the training data $\mathbf{X}$ (as `face_features`) and $\mathbf{y}$ (as `face_labels`) to find a good set of weights. Here is some starter code (make sure to place the file `face_emotion_data.npz` in the same directory as your Python script or notebook):

```
import numpy as np

##### Part a #####
# Load in training data and labels
```

```
# File available on Canvas
face_data_dict = np.load("face_emotion_data.npz")
face_features  = face_data_dict["X"]
face_labels    = face_data_dict["y"]
n, p = face_features.shape

# Solve the least-squares solution. weights is the array of
   weight coefficients
# TODO: find weights

print(f"Part 4a. Found weights:\n{weights}")
```

b) (3 points) How would you use these weights (i.e., the ones learned from the training data) to classify a new face image as smiling or non-smiling? You can assume you have access to the feature-extraction methods used to create the training data.

c) (10 points) A common method for estimating the performance of a classifier is cross-validation (CV). CV works like this. Divide the dataset into 8 equal sized subsets (e.g., examples $1-16$, $17-32$, etc). Use 7 sets of the data to chose your weights, then use the weights to predict the labels of the remaining "hold-out" set. Compute the number of mistakes made on this hold-out set and divide that number by 16 (the size of the set) to estimate the error rate. Repeat this process 8 times (for the 8 different choices of the hold-out set) and average the error rates to obtain a final estimate.

We provide starter code below with the cross-validation error calculated in a separate function for easy reuse in part e).

```
def lstsq_cv_err(features: np.ndarray, labels: np.ndarray,
   subset_count: int=8) -> float:
    """Estimate the error of a least-squares classifier
       using cross-validation. Use subset_count different
       train/test splits with each subset acting as the
       holdout set once.

    Parameters:
        features (np.ndarray): dataset features as a 2D
            array with shape (sample_count, feature_count)
        labels (np.ndarray): dataset class labels (+1/-1)
             as a 1D array with length (sample_count)
        subset_count (int): number of subsets to divide the
            dataset into
            Note: assumes that subset_count divides the
               dataset evenly

    Returns:
        cls_err (float): estaimted classification error
```

```
            rate of least-squares method
    """
    sample_count, feature_count = features.shape
    subset_size = sample_count // subset_count
    # Reshape arrays for easier subset-level manipulation
    features = features.reshape(subset_count, subset_size,
       feature_count)
    labels   = labels.reshape(subset_count, subset_size)

    subset_idcs = np.arange(subset_count)
    train_set_size = (subset_count - 1) * subset_size
    subset_err_counts = np.zeros(subset_count)

    for i in range(subset_count):
        # TODO: select relevant dataset,
        # fit and evaluate a linear model,
        # then store errors in subset_err_counts[i]
        pass

    # Average over the entire dataset to find the
       classification error
    cls_err = np.sum(subset_err_counts) / (subset_count*
       subset_size)
    return cls_err
# Run on the dataset with all features included
full_feat_cv_err = lstsq_cv_err(face_features, face_labels)
print(f"Error␣estimate:␣{full_feat_cv_err*100:.3f}%")
```

d) (3 points) Suppose you are designing an alternate classifier that does not use all 9 features. How could you decide which features to keep or remove, without needing to brute-force the full space of different feature subsets? (Keep in mind, this does not need to be optimal but just a heuristic)

e) (8 points) Using the method you suggested in the previous part, remove as many features as you can while keeping the cross-validation error rate below 6%. Be sure to include any code used, as well as the set of feature indices and its corresponding error.

5. Polynomial fitting. (20 points) (Recall that in HW1, you explored writing a general expression for a degree-d polynomial $p(\boldsymbol{z}) = y$ and expressing this as a system in matrix form $\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$.)

Again, consider observations of the form $(\boldsymbol{z}_i, y_i)$, $i = 1, \ldots, n$. Imagine these points are measurements from a scientific experiment. The variables $\boldsymbol{z}_i$ are the experimental conditions with two dimensions, i.e. $\boldsymbol{z}_i = (x_{i,1}, x_{i,2})$ and the y_i correspond to the measured response in each condition. Suppose we wish to fit a degree $d < n$ polynomial to these data. In other

words, we want to find the coefficients of a degree d polynomial p so that $p(\boldsymbol{z}_i) \approx y_i$ for $i = 1, 2, \ldots, n$. Unlike HW1, we will ignore any cross terms between $x_{i,1}$ and $x_{i,2}$ in order to simplify the problem a bit. We will set this up as a least-squares problem.

Write a Python script to find the least-squares fit to the $n = 100$ data points in `polydata_2D.npz` (which you can find on the Canvas assignment post for HW2). Plot the points and the polynomial fits for $d = 1, 2, 3$. (Hint: Due to ignoring cross-terms, your weight matrix should have 7 values in the $d = 3$ case.) Use the 3D viewer in matplotlib to see the surfaces from multiple angles.

Here is starter code to help get started and to help with plotting in 3D:

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

# File available on Canvas
data = np.load('polydata_2D.npz')
x1 = np.ravel(data['x1'])
x2 = np.ravel(data['x2'])
y = data['y']

N = x1.size
p = np.zeros((3,N))

for d in [1,2,3]:
    # Generate the X matrix for this d

    # Find the least-squares weight matrix w_d

    # Evaluate the best-fit polynomial at each point (x1,x2)
    # and store the result in the corresponding column of p

# Plot the degree 1 surface
Z1 = p[0,:].reshape(data['x1'].shape)
ax = plt.axes(projection='3d')
ax.scatter(data['x1'],data['x2'],y)
ax.plot_surface(data['x1'],data['x2'],Z1,color='orange')
plt.show()

# Plot the degree 2 surface
Z2 = p[1,:].reshape(data['x1'].shape)
ax = plt.axes(projection='3d')
ax.scatter(data['x1'],data['x2'],y)
ax.plot_surface(data['x1'],data['x2'],Z2,color='orange')
plt.show()

# Plot the degree 3 surface
```

```
Z3 = p[2,:].reshape(data['x1'].shape)
ax = plt.axes(projection='3d')
ax.scatter(data['x1'],data['x2'],y)
ax.plot_surface(data['x1'],data['x2'],Z3,color='orange')
plt.show()
```