Mathematical Foundations of Machine Learning. Homework 2

Teresa Morales

October 2019

1. Let

$$X = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

   (a) What is the rank of X? **The rank of this matrix is 3.** rank of a matrix is the number of linearly independent columns, which has to be equal to the number of linearly independent rows. A collection of vectors v_1, $v_2 ... v_p \epsilon R^n$ are linearly independent if $\Sigma_{i=1}^{p} a_i v_i = 0$ if and only if $a_i = 0$ for all i.
   In this case, we have that

$$a_1 + a_2 + a_3 = 0$$

$$a_1 + a_2 = 0$$

$$a_1 = 0$$

   This system of equations has only one solution in which $a_1 = 0$, $a_2 = 0$ and $a_3 = 0$, which means that the three vectors are linearly independent. Thus, the rank of X is 3.

   (b) Suppose that $y = Xw$. Derive an explicit formula for w in terms of y.

$$w = X^{-1}y$$

   We have to calculate the inverse of matrix X, using the following steps:
   First, we estimate the matrix of minors:

$$MM = \begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & -1 \\ -1 & -1 & 0 \end{bmatrix}.$$

   Second, we estimate the matrix of co-factors which coincides with the adjugate because it is symetrical (equal to its transpose):

$$Adjugate = \begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & 1 \\ -1 & 1 & 0 \end{bmatrix}.$$

Finally, we divide by the determinant of X which is equal to -1, so we have that:

$$w = X^{-1}y = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} y_3 \\ y_1 - y_3 \\ y_1 - y_2 \end{bmatrix}$$

2. Consider trying to estimate how many calories there are per gram of carbohydrates, fat, and protein in breakfast cereals. The data matrix for this problem is X. Each row contains the grams/serving of carbohydrates, fat, and protein, and each row corresponds to a different cereal (Frosted Flakes, Grape-Nuts, Teenage Mutant Ninja Turtles). The total calories for each cereal are y.

$$X = \begin{bmatrix} 25 & 0 & 1 \\ 20 & 1 & 2 \\ 40 & 1 & 6 \end{bmatrix} \quad y = \begin{bmatrix} 110 \\ 110 \\ 210 \end{bmatrix}$$

(a) Write a small program (e.g., in Matlab or Python) that solves the system of equations $Xw = y$. Recall the solution w gives the calories/gram of carbohydrate, fat, or protein. What is the solution?

The formula to calculate the estimated weigths for these features would be the following:

$$\hat{w} = (X^T X)^{-1} X^T y$$

The code is the following:

```
import numpy as np
def calc_weights(X, y):
    Xt = np.transpose(X)
    XtX = np.matmul(Xt, X)
    Inverse_XtX = np.linalg.inv(XtX)
    w = np.matmul(np.matmul(Inverse_XtX, Xt), y)
    print('Weights are', w)
    return w

X0 = [[25, 0, 1],
      [20, 1, 2],
      [40, 1, 6]]

y0 = [110, 110, 210]

calc_weights(X0, y0)
```

The estimated weights are 4.25 calories per gram of carbohydrates, 17.5 calories per gram of fat and 3.75 calories per gram of protein, as showin in our results:

```
Weights are [ 4.25 17.5   3.75]
```

(b) The solution may not agree with the known calories/gram, which are 4 for carbs, 9 for fat and 4 for protein. We suspect this may be due to rounding the grams to integers, especially the fat grams. Assuming the true value for calories/gram is:

$$w = \begin{bmatrix} 4 \\ 9 \\ 4 \end{bmatrix}$$

and that the total calories, grams of carbs, and grams of protein are correctly reported above, determine the "correct" grams of fat in each cereal.

We know that,

$$y = Xw$$

$$\begin{bmatrix} 110 \\ 110 \\ 210 \end{bmatrix} = \begin{bmatrix} 25 & x & 1 \\ 20 & y & 2 \\ 40 & z & 6 \end{bmatrix} . \begin{bmatrix} 4 \\ 9 \\ 4 \end{bmatrix}$$

Solving this system of equations we have that $x = 2/3 = 0.\overline{66}$**,** $y = 22/9 == 2.\overline{44}$ **and** $z = 26/9 = 2.\overline{88}$**.**

(c) Now suppose that we predict total calories using a more refined breakdown of carbohydrates, into total carbohydrates, complex carbohydrates and sugars (simple carbs). So now we will have 5 features to predict calories (the three carb features + fat and protein). So let's suppose we measure the grams of these features in 5 different cereals to obtain this data matrix:

$$X = \begin{bmatrix} 25 & 15 & 10 \\ 0 & 1 & \\ 20 & 12 & 8 \\ 1 & 2 & \\ 40 & 30 & 10 \\ 1 & 6 & \\ 30 & 15 & 15 \\ 0 & 3 & \\ 35 & 20 & 15 \\ 2 & 4 & \end{bmatrix}$$

$$y = \begin{bmatrix} 104 \\ 97 \\ 193 \\ 132 \\ 174 \end{bmatrix}$$

Can you solve $Xw = y$ Carefully examine the situation in this case. Is there a solution that agrees with the true calories/gram?

If we observe matrix X we can see that we cannot solve this problem since there is some information that is redundant. That is to say, **the columns of X are not linearly independent**. In particular we observe that the first column is the sum of the second and third columns. In order to solve this problem we need to eliminate one of these columns. We eliminate the first one since it would make sense to estimate the weights for each type of carbohydrates (complex and sugars).

Using the same program as above, the additional code to estimate this new problem would be:

```
1
2  X1 = [[15, 10, 0, 1],
3        [12, 8, 1, 2],
4        [30, 10, 1, 6],
5        [15, 15, 0, 3],
6        [20, 15, 2, 4]]
7
8  Y1 = [104, 97, 193, 132, 174]
9
10 calc_weights(X1, Y1)
```

As we can see, in this case, **the solution agrees with the true weights and we have 4 calories per gram for each type of carbohydrate, 9 calories per gram for fats and 4 calories per gram for proteins**.

```
1 Weights are [4. 4. 9. 4.]
```

3. Gradients Calculate the gradients of the following functions with respect to w:

   (a) $f(w) = w^T(2x)$

   $$\nabla_w f = 2x$$

   (b) $f(w) = 3w^T x - 0.5x^T w$

   $$\nabla_w f = 3x - 0.5x = 2.5x$$

(c) $f(w) = w^T \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} w$

$$\nabla_w f = Qw + Q^T w$$

$$= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

$$= \begin{bmatrix} 2w_1 + 5w_2 \\ 5w_1 + 8w_2 \end{bmatrix}$$

(d) $f(w) = w^T \begin{bmatrix} 1 & 2.5 \\ 2.5 & 4 \end{bmatrix} w$

$$\nabla_w f = Qw + Q^T w = 2Qw = \begin{bmatrix} 2w_1 + 5w_2 \\ 5w_1 + 8w_2 \end{bmatrix}$$

4. **Design classifier to detect if a face image is happy.**

Consider the two faces below. It is easy for a human, like yourself, to decide which is happy 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 happy and mad 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 Matlab file `face_emotion_data.mat`. This file also includes an $n \times 1$ binary vector $\mathbf{y}$; happy faces are labeled $+1$ and mad 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 happy or mad.

(a) Use the training data $\mathbf{X}$ and $\mathbf{y}$ to find an good set of weights.

The python code used to find a set of weights is the following:

```
import scipy.io as sio
import numpy as np

file_path = '/Users/teresamorales/Documents/Harris/MFML/
    Homework 2/face_emotion_data.mat'
face_data = sio.loadmat(file_path)
X = face_data['X']
y = face_data['y']


def calc_weights(X, y):
```

```
    Xt = np.transpose(X)
    XtX = np.matmul(Xt, X)
    Inverse_XtX = np.linalg.inv(XtX)
    w = np.matmul(np.matmul(Inverse_XtX, Xt), y)
    return w

w = calc_weights(X, y)
```

The weights obtained are the following:

$$\hat{w} = \begin{bmatrix} 0.944 \\ 0.214 \\ 0.266 \\ -0.392 \\ -0.005 \\ -0.017 \\ -0.166 \\ -0.082 \\ -0.166 \end{bmatrix}$$

(b) How would you use these weights to classify a new face image as happy or mad?

Let

$$X_{new} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix}$$

be a vector containing the 9 specific distances of a given new face that was not part of the training sample. Then, **our best prediction of whether the face is happy or mad would be determined by the following equations, using the estimated weights**:

$$\tilde{y} = sign(\hat{y})$$
$$\hat{y} = \hat{w} X_{new}$$

The following code shows how we would build our classifier and an example that results in a face classified as being mad:

```
def sign(num):
    return -1 if num < 0 else 1

def classify_face(training_features, labels, new_features)
    :
```

```
    w_training= calc_weights(training_features, labels)
    y_hat = np.matmul(new_features, w_training)
    y_tilda = [sign(i) for i in y_hat]
    if y_tilda == 1:
        print('The face is happy')
    else:
        print('The face is mad')

# Example:
x_new = [0.2, 0.8, 0.3,0.2, 0.8, 0.3,0.2, 0.8, 0.3]
classify_face(X, y, x_new)

```

(c) Which features seem to be most important? Justify your answer. Given the fact that all features are measuring distances in a face we could assume that these features have the same unit of measure. Similarly, we could assume that they have a similar variance.

Given these assumptions we could pick features by just selecting the ones with the highest absolute value, since they seem to be explaining a bigger share of the outcome (being happy or mad). **The features that have the highest absolute value are 1, 4, 3 and 2 (in that order)**.

(d) Can you design a classifier based on just 3 of the 9 features? Which 3 would you choose? How would you build a classifier?

As explained in the previous section we could pick features with the highest weights (absolute values). We would therefore choose features 1, 4 and 3. Given this selection, **we would need to rebuild our classifier to estimate weights when we are only considering those three features.**

Using the same program as in section a, we would now apply our function to our subset of features.

```
X_sel = X[:, [0, 2, 3]]
w_sel = calc_weights(X_sel, y)
```

Our new weights are $w_1 = 0.705$**,** $w_4 = -0.788$**,** $w_3 = 0.874$**.** Note that while the sign of the weights has been kept, the relative importance of each of these weights has changed when we have excluded the other features. This is probably due to the fact that our selected features are correlated to other features that we were taking into account so, when we exclude the other distances, part of that information is captured by our new weights.

The new classifier gives the same result for the example given above (the face is classified as being mad).

```
X_sel = X[:, [0, 2, 3]]
x_new_sel= [0.2, 0.3,0.2]
classify_face(X_sel, y, x_new_sel)
```

(e) A common method for estimating the performance of a classifier is cross-validation (CV). CV works like this. Divide the data set 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.

The python code for cross validation is the following:

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

def calc_weights(X, y):
    Xt = np.transpose(X)
    XtX = np.matmul(Xt, X)
    Inverse_XtX = np.linalg.inv(XtX)
    w = np.matmul(np.matmul(Inverse_XtX, Xt), y)
    return w

def random_partition(sample_size, n_chunks, chunk_size):
    sample_size_index = list(range(sample_size))
    random.Random(222).shuffle(sample_size_index)  # randomizes selection setting seed 222
    return [sample_size_index[round(chunk_size * i):round(chunk_size * (i + 1))] for i in range(n_chunks)]

def sign(num):
    return -1 if num < 0 else 1

def calc_error_rate(reserved_index, n_chunks, chunk_size, features_matrix, labels_vector):
    X_reserved = features_matrix[sliced_index[reserved_index]]
    y_reserved = labels_vector[sliced_index[reserved_index]]
    X_training = features_matrix[sum(sliced_index[0:reserved_index] +
                                     sliced_index[reserved_index+1:n_chunks], [])]
    y_training = labels_vector[sum(sliced_index[0:reserved_index] +
                                   sliced_index[reserved_index+1:n_chunks], [])]
    w_training = calc_weights(X_training, y_training)
    y_hat = np.matmul(X_reserved, w_training)
    y_tilda = [sign(i) for i in y_hat]
    return np.sum([y_tilda[i] != y_reserved[i] for i in range(int(chunk_size))])/chunk_size

# Cross validation with 9 features:

```

```
sample_size = 128
number_of_chunks = 8
chunk_size = sample_size/number_of_chunks

sliced_index = random_partition(sample_size,
    number_of_chunks, chunk_size)

error_rates = [calc_error_rate(i, number_of_chunks,
    chunk_size, X, y)
                for i in range(number_of_chunks)] #loop for
     8 alternative training samples

np.mean(error_rates)

# Cross validation with 3 features (same randomization as
    with 9 features):

X_sel = X[:, [0, 2, 3]]

error_rates_3features = [calc_error_rate(
    i, number_of_chunks, chunk_size, X_sel, y) for i in
    range(number_of_chunks)]

np.mean(error_rates_3features)
```

(f) What is the estimated error rate using all 9 features? What is it using the 3 features you chose in (d) above?

The estimated error rate using all 9 features is 0.03125.

The estimated error rate using only 3 features is 0.0547.

The error rate is therefore smaller when we use the whole set of features to predict whether a face is happy or mad.

5. **Polynomial fitting.** Suppose we observe pairs of scalar points (z_i, y_i), $i = 1, \ldots, n$. Imagine these points are measurements from a scientific experiment. The variables z_i are the experimental conditions 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 $p(z_i) \approx y_i$ for $i = 1, 2, \ldots, n$. We will set this up as a least-squares problem.

(a) Suppose p is a degree d polynomial. Write the general expression for $p(z) = y$.

The general expression would be the following: $y_i = w_0 z_i^0 + w_1 z_i^1 + w_2 z_i^2 + ... + w_d z_i^d$ for $i = 1, \ldots, n$.

(b) Express the $i = 1, \ldots, n$ equations as a system in matrix form $Xw = y$. Specifically, what is the form/structure of X in terms of the given x_i.

X would have the form of the Vandermonde matrix:

$$\begin{bmatrix} z_1^0 & z_1^1 & z_1^2 \\ \dots & z_2^d & \\ z_2^0 & z_2^1 & z_2^2 \\ \dots & z_2^d & \\ \dots & \dots & \dots \\ \dots & \dots & \\ z_n^0 & z_n^1 & z_n^2 \\ \dots & z_n^d & \end{bmatrix} \cdot \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_d \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}$$

$$(n * (d+1)) * ((d+1) * 1) = n * 1$$

(c) Write a Matlab or Python script to find the least-squares fit to the $n = 30$ data points in `polydata.mat`. Plot the points and the polynomial fits for $d = 1, 2, 3$.

The script is the following:

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

file_path = '/Users/teresamorales/Documents/Harris/MFML/Homework 2/polydata.mat'
poly_data = sio.loadmat(file_path)
x = poly_data['x']
y = poly_data['y']


def calc_weights(X, y):
    Xt = np.transpose(X)
    XtX = np.matmul(Xt, X)
    Inverse_XtX = np.linalg.inv(XtX)
    w = np.matmul(np.matmul(Inverse_XtX, Xt), y)
    return w


V_Matrix_b = np.fromfunction(lambda i, j: x[i] ** j, (30, 30), dtype=int)

Weights = {d: calc_weights([V_Matrix_b[i][0][0:d+1] for i in range(30)], y)
           for d in (1, 2, 3)}

for d in (1, 2, 3):
    print('Weights for a polynomial of degree', d, 'are', Weights[d])

#If estimated y's are needed we would calculate them and store them in our data set as follows:

y_est = {d: np.matmul([V_Matrix_b[i][0][0:d+1] for i in range(30)], Weights[d])
```

```
        for d in (1, 2, 3)}
poly_data['y_est_1'], poly_data['y_est_2'], poly_data['
    y_est_3'] = y_est[1], y_est[2], y_est[3]

# Graph data and fitted functions:

x_graph = np.linspace(0, 1, num=100)

fx_2 = []

for i in range(len(x_graph)):
    fx_2.append(Weights[2][0]+Weights[2][1]*x_graph[i]+
    Weights[2][2]*x_graph[i]**2)

fx_3 = []

for i in range(len(x_graph)):
    fx_3.append(Weights[3][0]+Weights[3][1]*x_graph[i]+
                Weights[3][2]*x_graph[i]**2 + Weights
    [3][3]*x_graph[i]**3)

fig, ax = plt.subplots(figsize=(12, 6))
plt.scatter(x, y)
plt.plot([min(x), max(x)], [min(y_est[1]), max(y_est[1])],
     color='red')
plt.plot(x_graph, fx_2, color='green')
plt.plot(x_graph, fx_3, color='pink')
plt.show()
```

Our results are:

```
Weights for a polynomial of degree 1 are [[-0.11914368]
 [ 0.21420653]]
Weights for a polynomial of degree 2 are [[ 0.00876394]
 [-0.53816355]
 [ 0.70458083]]
Weights for a polynomial of degree 3 are [[-0.04418309]
 [ 0.03303464]
 [-0.58563589]
 [ 0.796239  ]]
```

Figure 1 shows graph of our sample (dots) and the three fitted lines. The red line represents a polynomial of degree d=1, the green line represents a polynomial of degree d = 2 and the pink line a polynomial of degree d = 3.

Figure 1: