CMSC 25300: Mathematical Foundations of ML

Problem Set 2

Hung Le Tran

15 Oct 2023

Problem 2.1 (Problem 1)

Solution

(a)

No, they are not linearly independent:

$$(-2)\begin{bmatrix}1\\3\\-1\\2\end{bmatrix}+1\begin{bmatrix}0\\2\\1\\4\end{bmatrix}+1\begin{bmatrix}1\\4\\0\\2\end{bmatrix}-1\begin{bmatrix}-1\\0\\3\\2\end{bmatrix}=0$$

(b) Since $\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}\}$ are not linearly independent, we can have at most 3 linearly independent vectors in the column space.

We want to show that $\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}\}$ are indeed linearly independent.

Suppose there there exists $\alpha_1, \alpha_2, \alpha_3$ such that

$$\alpha_1\begin{bmatrix}1\\3\\-1\\2\end{bmatrix}+\alpha_2\begin{bmatrix}0\\2\\1\\4\end{bmatrix}+\alpha_3\begin{bmatrix}1\\4\\0\\2\end{bmatrix}=0$$

then that implies

$$\begin{cases}\alpha_1+\alpha_3=0\\3\alpha_1+2\alpha_2+4\alpha_3=0\\-\alpha_1+\alpha_2=0\\2\alpha_1+4\alpha_2+2\alpha_3=0\end{cases}$$

The first and third equation implies $\alpha_1 = \alpha_2 = -\alpha_3 \equiv a$. The third equation then implies

$$0 = 3a + 2a - 4a = a \Rightarrow a = 0 \Rightarrow \alpha_1 = \alpha_2 = \alpha_3 = 0$$

$\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}\}$ are indeed linearly independent.

(c) No. $\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}\}$ are **NOT** linearly independent, one can replace any of $\mathbf{x_1}, \mathbf{x_2}$ or $\mathbf{x_3}$ with $\mathbf{x_4}$ and linearly independent set in (c) would still be linearly independent.

In short, any set of 3 column vectors of $\mathbf{X}$ is a linearly independent set.

(d) Since the largest possible set of linearly independent column vectors of $\mathbf{X}$ has 3 elements, $Rank(\mathbf{X}) = 3$ □

Problem 2.2 (Problem 2)

Solution

(a) Yes, they are. If there exists a_1, a_2 such that

$$a_1 \begin{bmatrix} 1.4 \\ -1.4 \\ 1.4 \\ -1.4 \end{bmatrix} + a_2 \begin{bmatrix} -1.4 \\ 1.4 \\ 1.4 \\ 1.4 \end{bmatrix} = 0$$

then

$$1.4a_1 - 1.4a_2 = 0, 1.4a_1 + 1.4a_2 = 0$$

which easily implies $a_1 = a_2 = 0$.

(b) No, they are not.

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

(c) Yes, they are. If there exists a_1, a_2, a_3 such that

$$a_1 \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} + a_2 \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} + a_3 \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix} = 0$$

then

$$\begin{cases} -a_1 + a_2 - a_3 = 0 \\ -a_2 - a_3 = 0 \\ a_1 = 0 \end{cases}$$

Substituting $a_1 = 0$ from the last equation into the first 2, it's easily to see that $a_2 = a_3 = 0$ too.

(d) $Rank(\mathbf{X}) = 2$. The 2 column vectors of $\mathbf{X}$ are not linearly independent, because if they were, the second vector has to be a scaled version of the first column vector, which it clearly isn't ($\frac{6}{4} \neq \frac{14}{-6}$). This is also the maximum number of linearly independent column vectors for $\mathbf{X}$, so its rank must also be 2. □

Problem 2.3 (Problem 3)

Solution

(a)

$$f(\mathbf{w}) = (3\mathbf{x})^T\mathbf{w} - 2\mathbf{w}^T\mathbf{x} = \mathbf{x}^T\mathbf{w}$$

so

$$\nabla_{\mathbf{w}} f = \mathbf{x}$$

(b)

$$f(\mathbf{w}) = (\mathbf{w} - 2\mathbf{x})^T(\mathbf{w} - \mathbf{x}) = \mathbf{w}^T\mathbf{w} - 3\mathbf{x}^T\mathbf{w} + 2\mathbf{x}^T\mathbf{x} = (\mathbf{w} - 3\mathbf{x})^T\mathbf{w} + 2\mathbf{x}^T\mathbf{x}$$

so

$$\nabla_{\mathbf{w}} f = \mathbf{w} - 3\mathbf{x}$$

(c)

$$f(\mathbf{w}) = \mathbf{x}^T \begin{bmatrix} 2 & 3 \\ 5 & 7 \end{bmatrix} \mathbf{w} = \mathbf{x}^T \begin{bmatrix} 2 & 5 \\ 3 & 7 \end{bmatrix}^T \mathbf{w} = \left(\begin{bmatrix} 2 & 5 \\ 3 & 7 \end{bmatrix} \mathbf{x} \right)^T \mathbf{w}$$

so

$$\nabla_{\mathbf{w}} f = \begin{bmatrix} 2 & 5 \\ 3 & 7 \end{bmatrix} \mathbf{x}$$

(d)

$$f(\mathbf{w}) = \mathbf{w}^T \begin{bmatrix} 1 & 4 \\ 4 & 16 \end{bmatrix} \mathbf{w}$$

For f to make sense in the first place, $\mathbf{w}$ must have shape 2×1.

Therefore

$$f(\mathbf{w}) = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 4 & 16 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} w_1 + 4w_2 \\ 4w_1 + 16w_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} w_1^2 + 5w_1w_2 + 4w_2^2 \\ 4w_1^2 + 20w_1w_2 + 16w_2^2 \end{bmatrix}$$

yielding

$$\nabla_{\mathbf{w}} f(\mathbf{w}) = \begin{bmatrix} 2w_1 + 5w_2 \\ 20w_1 + 32w_2 \end{bmatrix} = \begin{bmatrix} 2 & 5 \\ 20 & 32 \end{bmatrix} \mathbf{w}$$

(e)

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

For f to make sense in the first place, $\mathbf{w}$ must have shape 2×1.

Therefore

$$f(\mathbf{w}) = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 4 & -1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 2w_1 + 3w_2 \\ 4w_1 - w_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 2w_1^2 + 5w_1w_2 + 3w_2^2 \\ 4w_1^2 + 3w_1w_2 - w_2^2 \end{bmatrix}$$

yielding

$$\nabla_{\mathbf{w}} f(\mathbf{w}) = \begin{bmatrix} 4w_1 + 5w_2 \\ 3w_1 - 2w_2 \end{bmatrix} = \begin{bmatrix} 4 & 5 \\ 3 & 2 \end{bmatrix} \mathbf{w}$$

□

Problem 2.4 (Problem 4)

Solution

(a)

```
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")
X = face_data_dict ["X"]
y = face_data_dict ["y"]
n , p = X.shape


# Solve the least - squares solution . weights is the array of
# weight coefficients
# TODO : find weights
weights = np.linalg.inv((X.T @ X)) @ (X.T) @ y

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

(b) We've found weight $\mathbf{w}$ from part (a). To classify a new face $\mathbf{x}_0$:

$$\hat{\mathbf{y}}(\mathbf{x}_0) = \begin{cases} +1 & \text{if } \mathbf{x}_0^T \mathbf{w} \geq 0 \\ -1 & \text{if } \mathbf{x}_0^T \mathbf{w} < 0 \end{cases}$$

(c)

```
    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 2 D array
    with shape ( sample_count , feature_count )
        labels ( np . ndarray ) : dataset class labels (+1/ -1) as a 1
    D 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 ) : estimated classification errorrate 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 ]
        X = np.row_stack([features[j] for j in subset_idcs if j != i])
        y = np.concatenate([labels[j] for j in subset_idcs if j != i])
        w = np.linalg.inv(np.dot(X.T, X)) @ (X.T) @ y
        yhat = np.dot(features[i], w) > 0
        ytrue = labels[i] > 0
        subset_err_counts[i] = np.sum(yhat ^ ytrue)
    # 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) I would first find the weight using all 9 features. Then, I would remove the feature that has the smallest corresponding weight. This implies that that feature has the least correlation to the final prediction. Of course, this is given that all features are properly scaled to the same range/order of magnitude. Find the weight using the remaining 8 features. Continue doing this process until the error rate exceeds some acceptable threshold value.

(e)

```
def lstsq_cv_err_least_index (features : np.ndarray , labels : np.
    ndarray , subset_count : int =8) -> float :
    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 )

    sum_w = np.zeros(feature_count)
    for i in range ( subset_count ) :
        X = np.row_stack([features[j] for j in subset_idcs if j != i])
        y = np.concatenate([labels[j] for j in subset_idcs if j != i])
        w = np.linalg.inv(np.dot(X.T, X)) @ (X.T) @ y

        sum_w += np.abs(w)

        yhat = np.dot(features[i], w) > 0
        ytrue = labels[i] > 0
        subset_err_counts[i] = np.sum(yhat ^ ytrue)
    # Average over the entire dataset to find the classification error
    cls_err = np . sum ( subset_err_counts ) / ( subset_count *
    subset_size )
    # returns feature index with least weight
    li = np.argmin(sum_w)
    return (cls_err, li)


err = 0
featuresNotUsed = set()
sample_count , feature_count = face_features.shape
while(err < 0.06 and len(featuresNotUsed) <= feature_count - 1):
    features_used = face_features
    features_used = np.array([[row[j] for j in range(feature_count) if
   j not in featuresNotUsed] for row in face_features])
    cls_err, li = lstsq_cv_err_least_index ( features_used, face_labels
   )
    print (f"Error estimate :{cls_err *100:.3f}%, features not used: {
   featuresNotUsed}, feature to remove (in sub array): {li}")

    # converting feature index in subarray to feature index in original
    array
    actualIndex = -1
    count = -1
    while(count < li):
        actualIndex += 1
        while(actualIndex in featuresNotUsed):
            actualIndex += 1
        count += 1

    featuresNotUsed.add(actualIndex)
    err = cls_err
```

which prints out

```
Error estimate :4.688%, features not used: set(), feature to remove (in
    sub array): 5
```

```
Error estimate :4.688%, features not used: {5}, feature to remove (in
    sub array): 4
Error estimate :4.688%, features not used: {4, 5}, feature to remove (
    in sub array): 5
Error estimate :4.688%, features not used: {4, 5, 7}, feature to remove
     (in sub array): 5
Error estimate :6.250%, features not used: {8, 4, 5, 7}, feature to
    remove (in sub array): 1
```

So my choice of features (to keep CV error rate below 6 %) would be all features except for 5th, 6th and 8th feature (4th, 5th and 7th in 0-index). □

Problem 2.5 (Problem 5)

Solution

```
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
    X = None
    if d == 1:
        X = np.column_stack([x1, x2, np.ones(N)])
    elif d==2:
        X = np.column_stack([x1, x1*x1, x2, x2*x2, np.ones(N)])
    else:
        X = np.column_stack([x1, x1*x1, x1*x1*x1, x2, x2*x2, x2*x2*x2,
   np.ones(N)])

    w_d = np.linalg.inv(np.matmul(X.T, X)) @ (X.T) @ y
    p[d-1] = X @ w_d

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 ()
```

□