Homework 2

University of Central Florida
Electrical and Computer Engineering

Current Topics in Machine Learning (EEL 4815), Fall 2023

Issued: Thu, Oct 12th **Due**: Mon Oct 30th.

(k Nearest Neighbor) In this exercise, we will use a kNN classifier to classify the handwritten digits of the digits dataset in `sklearn.datasets`. The dataset consists of the 1797 images of hand-written digits, where each digit is represented by a 64-dimensional vector of pixel values.

- First, import and read the digits dataset. To this end, you can use

```
from sklearn.datasets import load_digits
digits = load_digits()
X, y = digits.data, digits.target
```

- Next, split the data into training and test data, and print the shape of the train and test data and targets. For visualization, plot few of the digits using the `imshow` method of `matplotlib.pyplot` and reshaping the vector to 8×8.
- Initialize and train a kNN classifier model for $k = 1, 3, 5$. Compute and report the accuracy on the test data.

(Logistic Regression) You will use the social media ads data to determine whether the social media user's demographics can be used to predict the user buying the product advertised.

- Use the model class `Logit` from `statsmodels.api` to build a logistic regression-based classifier. You will find the dataset `Social_Network_Ads.csv` available on webcourses. Report your model's classification accuracy, plot the ROC curve, and report the area under the curve (AUC).
- Repeat but this time using the logistic regression class `logistic_regression_class.py` posted on webcourses. Confirm that the results are consistent by using the same data in both methods for training and testing.

(Linear Discriminant Functions for Classification) In this exercise, we will implement classification algorithms using linear discriminants. This includes the least-squares (LS) approach, Fisher's linear discriminant and the perceptron algorithm.

- **Dataset generation:** First, we will generate our data. We will use the two-dimensional Gaussian blobs from `sklearn.datasets`. To generate and plot 1000 samples from two classes $\mathcal{C}_1$ and $\mathcal{C}_2$ along with their labels, use the following code:

```
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
X, y = make_blobs(n_samples=1000, centers=2)
plt.scatter(X[:,0], X[:,1], c=y)
```

Next, you can obtain the training and test data using

```
y_true = y[:, np.newaxis]
X_train, X_test, y_train, y_test = train_test_split(X, y_true)
```

which splits the data into a training set with 75% of the points and a test set with 25% of the points.

- **Least-squares (LS) solution:** Now, we will use LS for classification. To this end, compute the weights of the model using (Eq. 4.16 in PRML)

$$\widetilde{\mathbf{W}} = (\widetilde{\mathbf{X}}^\top \widetilde{\mathbf{X}})^{-1} \widetilde{\mathbf{X}}^\top \mathbf{T}\,, \tag{1}$$

which minimizes a sum of squares error function. Since the data here is 2-dimensional, $\widetilde{\mathbf{W}}$ is a 3×2 matrix whose k-th column is the 3-dimensional vector $\tilde{\mathbf{w}}_k = (w_{k0}, w_{k1}, w_{k2})^\top, k = 1, 2$, $\widetilde{\mathbf{X}}$ is the matrix of augmented inputs whose n-th row is $\tilde{\mathbf{x}}_n^\top = (1, \mathbf{x}_n^\top)$, where $\mathbf{x}_n$ is the n-th training example, and $\mathbf{T}$ is a matrix whose n-th row is the target vector $\mathbf{t}_n^\top$, with a one-hot encoding for the target vectors. Note that, since the class label of the n-th point is either $y_n = 0$ or $y_n = 1$, its one-hot encoding can be obtained as $\mathbf{t}_n^\top = [1 - y_n \quad y_n]$.

After learning the weights, you can readily obtain the predictions for a new test point $\mathbf{x}$ by computing

$$\mathbf{y}(\mathbf{x}) := [y_1(x), y_2(x)] = \widetilde{\mathbf{W}}^\top \tilde{\mathbf{x}}\,. \tag{2}$$

The input is then assigned to the class for which y_k is largest. Obtain the prediction for the test data points and evaluate the test accuracy by computing the fraction of errors between the predictions and the ground truth.

Next, we would like to visualize the decision boundary of the learned classifier in the feature space. Note that the boundary is the hyperplane $(\tilde{\mathbf{w}}_1 - \tilde{\mathbf{w}}_2)\tilde{\mathbf{x}} = 0$. Hence, The hyperplane corresponds to the line

$$y_{\text{hyperplane}} = \text{slope} \cdot x_{\text{hyperplane}} + \text{intercept} \tag{3}$$

where the slope $= -(w_{11} - w_{21})/(w_{12} - w_{22})$ and the intercept $= -(w_{10} - w_{20})/(w_{12} - w_{22})$. Plot the dataset and the estimated decision hyperplane.

Finally, we would like to study the effect of outliers on the LS solution. To this end, generate an additional blob of 50 points and add them to one of the classes. Retrain your weights and visualize the decision boundary. Comment on the results.

- **Fisher's linear discriminant (FLD):** Next, you will implement the FLD classifier, which projects the data along the direction that maximizes the class separation. In particular, generate the blobs data as before and compute the weights using (PRML Eq. (4.30))

$$\mathbf{w} = \mathbf{S}_W^{-1}(\mathbf{m}_2 - \mathbf{m}_1) \tag{4}$$

where

$$\mathbf{m}_1 = \frac{1}{N_1} \sum_{n \in \mathcal{C}_1} \mathbf{x}_n \qquad\qquad \mathbf{m}_2 = \frac{1}{N_2} \sum_{n \in \mathcal{C}_2} \mathbf{x}_n \tag{5}$$

are the mean vectors of the two classes, and N_1 and N_2 are the number of training examples in $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Note that you can find the indices of the points that belong to the class for which

$y = 0$ using `np.where(y==0)`. The matrix $\mathbf{S}_W$ is the total within-class covariance matrix, given by (PRML Eq. (4.28))

$$\mathbf{S}_W = \sum_{n \in \mathcal{C}_1} (\mathbf{x}_n - \mathbf{m}_1)(\mathbf{x}_n - \mathbf{m}_1)^\top + \sum_{n \in \mathcal{C}_2} (\mathbf{x}_n - \mathbf{m}_2)(\mathbf{x}_n - \mathbf{m}_2)^\top \tag{6}$$

Now, project the test data along the computed direction $\mathbf{w}$; given a test input $\mathbf{x}$, $y(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$. Visualize the projected data using a scatter plot. Threshold the projection $y(\mathbf{x}) \lessgtr \mathrm{thr}$ to decide on the class label. You can eyeball the data to choose the threshold thr to best separate the projected data. Obtain the prediction for the test data points and evaluate the test accuracy by computing the fraction of errors between the predictions and the ground truth. Plot the dataset and the estimated decision boundary. Note that the decision boundary is the hyperplane $\mathbf{w}^T\mathbf{x} - \mathrm{thr} = 0$.

Bonus (Optional): Let's skew the isotropic data so that the covariances of the class distributions are non-diagonal. This can be simply achieved by multiplying the isotropic data by the square root of the desired (non-diagonal) covariance matrix. Choose non-diagonal covariance matrices $\mathbf{\Sigma_1}$ and $\mathbf{\Sigma_2}$ for the data points of each class. Given the data matrix $\mathbf{X}_1$ of class $\mathcal{C}_1$, you can obtain a new (skewed) data matrix by computing $\mathbf{X}_1\mathbf{L}$, where $\mathbf{L}$ is the square root matrix, i.e., $\mathbf{\Sigma_1} = \mathbf{L}^T\mathbf{L}$. Do the same for the data points of class $\mathcal{C}_2$. You can use the following function to compute the square root of some matrix $\mathbf{A}$.

```
def sqrtm(A):
    """
    Compute square root matrix
    """
    evalues, evectors = np.linalg.eig(A)
    # Ensuring square root matrix exists
    assert (evalues >= 0).all()
    sqrt_matrix = evectors @ np.diag(np.sqrt(evalues)) @
        np.linalg.inv(evectors)
    return sqrt_matrix
```

As before, obtain the optimal FLD weights and the predictions of the (skewed) test data. Plot the dataset and the estimated decision boundary. You should be able to see that FLD chooses the direction that maximizes the class separation, which is not necessarily the line joining the class means when the data is non-isotropic.

- **Perceptron algorithm:** Our last linear discriminant model is the perceptron algorithm. The perceptron is trained using gradient descent. The model parameters are first initialized then updated for a number of iterations until convergence. Given input $\mathbf{x}$, it computes $a = \mathbf{w}^T\mathbf{x} + w_o$. Note that this computation can be done in one step for all N training examples using vectorization and broadcasting. Subsequently, we apply the activation function, which returns the binary predictions

$$\hat{y}^{(i)} = 1 \text{ if } a^{(i)} \geq 0, \text{ else } 0, i = 1, \dots N.$$

The weights can be iteratively updated using the perceptron learning rule

$$\Delta\mathbf{w} = \eta\, \mathbf{X}^\top \cdot (\hat{\mathbf{y}} - \mathbf{y}) \tag{7}$$
$$\Delta w_0 = \eta\, (\hat{\mathbf{y}} - \mathbf{y})$$

where η is the learning rate and the entries are stacked in vectors and matrices of proper dimensions with the obvious naming of variables. The weights and bias are updated as

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w} \tag{8}$$

$$w_0 = w_0 + \Delta w_0$$

The following class is a perceptron model class.

```python
class Perceptron():

   def __init__(self):
      pass

   def train(self, X, y, learning_rate=0.05, n_iters=100):
      n_samples, n_features = X.shape

      # Step 0: Initialize the parameters
      self.weights = np.zeros((n_features,1))
      self.bias = 0

      for i in range(n_iters):
         # Step 1: Compute the activation
         a = np.dot(X, self.weights) + self.bias

         # Step 2: Compute the output
         y_predict = self.step_function(a)

         # Step 3: Compute weight updates
         delta_w = learning_rate * np.dot(X.T, (y - y_predict))
         delta_b = learning_rate * np.sum(y - y_predict)

         # Step 4: Update the parameters
         self.weights += delta_w
         self.bias += delta_b

      return self.weights, self.bias

   def step_function(self, x):
      return np.array([1 if elem >= 0 else 0 for elem in x])[:,
         np.newaxis]

   def predict(self, X):
      a = np.dot(X, self.weights) + self.bias
      return self.step_function(a)
```

Create a dataset as in the previous examples using the Gaussian blobs and split it into training and testing sets. Create an instance of the perceptron class (e.g., `p = Perceptron()`), then use its train function `p.train` to train the weights and the bias. You can use a learning rate $\eta = 0.05$ and 500 iterations.

Now use the predict function `p.predict` to obtain the predictions for the training and the test data. Evaluate the train and test accuracies. Plot a scatter plot of the data set and the decision boundary.