Introduction to Machine Learning
CSCE 478/878

Programming Assignment 4

Fall 2020

Linear Support Vector Machine & Principle Component Analysis

Basic Info

You will work in teams of maximum three students from the previous assignment.

The programming code will be graded on **both implementation and correctness**.

This assignment **doesn't require a written report**.

Assignment Goals

This assignment is intended to build the following skill:
- Implement the Gradient Descent algorithm for the Linear Support Vector Machine classifier model.
- Perform Principle Component Analysis (PCA) based dimensionality reduction by using the eigendecomposition technique.

Assignment Instructions

Note: you are not allowed to use any Scikit-Learn or python library for building the Linear SVM model and performing PCA.

i. The code should be written in a Jupyter notebook. Use the following naming convention.
   <lastname1>_ <lastname2>_<lastname3>_assignment4.ipynb
ii. The Jupyter notebook should be submitted via webhandin.

Score Distribution

Part A: 478 (70 pts) & 878 (80 pts)
Part B: 478 (45 pts) & 878 (45 pts)

Total: 478 (115 pts) & 878 (125 pts)

Part A: Linear Support Vector Machine (SVM)

Dataset: You will use the Iris dataset for **binary classification**. Use petal length and petal width features of the Iris dataset, and determine whether a sample is Iris Virginica or not.

URL: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html

1. Implement a **Linear_SVC** model class for performing **binary classification**. The model should implement the batch Gradient Descent (GD) algorithm.

   [**40 pts**]

   a) __init__(self, C=1, max_iter=100, tol=None, learning_rate='constant', learning_rate_init=0.001, t_0=1, t_1=1000, early_stopping=False, validation_fraction=0.1, **kwargs)

      This method is used to initialize the data members of the class when an object of class is created. For example, self.C = C

      Arguments:

      C : float
      It provides the regularization/penalty coefficient.

      max_iter : int
      Maximum number of iterations. The GD algorithm iterates until convergence (determined by 'tol') or this number of iterations.

      tol : float
      Tolerance for the optimization.

      learning_rate : string (default 'constant')
      It allows to specify the technique to set the learning rate: constant learning for all iterations, or varying learning rate given by a learning rate schedule function.

- 'constant': a constant learning rate given by 'learning_rate_init'.
- 'adaptive': gradually decreases the learning rate based on a schedule. Write a function that would be used if learning_rate is set to 'adaptive'. When 'adaptive' is used, the 'learning_rate_init' parameter has no effect as the learning rate varies by a learning rate schedule function. It uses the 't_0' and 't_1' parameters (see below).
-

Pseudocode for the "adaptive" learning_rate function:

Write a function that decreases learning rate gradually during each iteration:

$$\text{Learning rate} = \frac{t\_0}{iteration + t\_1}$$

where t_0 and t_1 are two constants that you need to determine empirically.

Choose constant t_0 and t_1 such that initially the learning rate is large enough.

learning_rate_init : double

The initial learning rate value if learning_rate is set to 'constant'. It controls the step-size in updating the weights. It has no effect is the 'learning_rate' is 'adaptive'.

early_stopping : Boolean, default=False

Whether to use early stopping to terminate training when validation score is not improving. If set to True, it will automatically set aside a fraction of training data as validation and terminate training when validation score is not improving.

validation_fraction : float, default=0.1

The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

b) fit(self, X, Y):

Implement the batch GD algorithm in the fit method. The weight vector and the intercept/bias should be denoted by w and b, respectively. Store the cost values for each iteration so that later you can use it to create a learning curve.

Arguments:

X : ndarray

A numpy array with rows representing data samples and columns representing features.

Y : ndarray

A 1D numpy array with labels corresponding to each row of the feature matrix X.

Note: the "fit" method should update the following parameters:

```
self.intercept_ = np.array([b])
self.coef_ = np.array([w])
self.support_vectors_ =
```

The "fit" method should display the total number of iterations using a print statement.

Returns:
    self

c)
predict(self, X)

Arguments:
    X : ndarray
        A numpy array containing samples to be used for prediction. Its rows represent data samples and columns represent features.

Returns:
    1D array of predicted class labels for each row in X.

Note: the "predict" method uses the **self.coef_[0]** and **self.intercept_[0]** to make predictions.

Binary Classification using Linear_SVC Classifier

2. Read the Iris data using the sklearn.datasets.load_iris method. Create the data matrix X by using two features: petal length and petal width. Recode the binary target such that Iris-Virginica samples are 1, and other samples are 0.

   [**1 pts**]

3. Partition the data into train and test set (80% - 20%). Use the "**Partition**" function from your previous assignment.

   [**2 pts**]

4. **Model selection via Hyper-parameter tuning**: Use the **kFold** function from previous assignment to find the optimal values for the following hyperparameters.

   [**5 pts**]

   C
   learning_rate
   learning_rate_init (when 'constant' learning_rate is used)

max_iter
tol

5. Train the model using optimal values for the hyperparameters and evaluate on the **test data**. Report the test accuracy and test confusion matrix.

   [**5 pts**]

6. Plot the learning curve.

   [**5 pts**]

7. Plot the decision boundary and show the support vectors using the "*decision_boundary_support_vectors*" function given in: https://github.com/rhasanbd/Support-Vector-Machine-Classifier-Beginners-Survival-Kit/blob/master/Support%20Vector%20Machine-1-Linearly%20Separable%20Data.ipynb

   [**12 pts**]

Note that if your test accuracy is less than 95% you will lose 10% of the total obtained points. If your test accuracy is less than 90% you will lose 30% of the total obtained points.

8. [**Extra Credit for 478 and Mandatory for 878**] Implement early stopping in the "fit" method of the Linear_SVC model. You will have to use the following two parameters of the model: early_stopping and validation_fraction. Also note that when training the model using early stopping it should generate an early stopping curve. [**10 pts**]

Part B: Principle Component Analysis

You will perform dimensionality reduction on a grayscale image (posted on Canvas) using PCA. The PCA will be implemented using the ***eigendecomposition*** technique. More specifically, you will find the top k eigenvectors (i.e., principle components) of a pixel matrix (a gray scale image). Then, using the top k eigenvector matrix, you will project the pixel matrix on its principle components. This will reduce the dimension of the pixel matrix without losing much variance.

Note: See the following notebook for understanding the manual implementation of the eigendecomposition based PCA using python.
https://github.com/rhasanbd/Dimensionality-Reduction-Get-More-From-Less-And-See-the-Unseen/blob/master/Dimensionality%20Reduction-PCA-Eigendecomposition-Introduction.ipynb

9. Using the matplotlib.pyplot "*imread*" function read the image as a 2D matrix. Denote it with "X". Show the image using matplotlib.pyplot *imshow* function.

   If the image is RGB, then you need to convert it into a grayscale image, as follows (use matplotlib.pyplot "gray" function).

   ```
   X = imread("image_path")[:,:,0]
   gray()
   ```

   [**3 pts**]

10. Implement the steps of eigendecomposition based PCA on X: (a) mean center the data matrix X, (b) compute the covariance matrix from it, (c) find eigenvalues and eigenvectors of the covariance matrix (you may use the *numpy.linalg.eig* function). [**7 pts**]

11. Then, find the top *k* eigenvectors (sort eigenvalue-eigenvector pairs from high to low, and get the top *k* eigenvectors), and create an eigenvector matrix using top *k* eigenvectors (each eigenvector should be a column vector in the matrix, so there should be *k* columns). [**10 pts**]

12. Finally project the mean centered data on the *k* top eigenvectors (it should be a dot product between mean centered X and the top *k* eigenvector matrix).

    [**5 pts**]

13. Reconstruct the data matrix by taking dot product between the projected data (from last step) and the transpose of the top *k* eigenvector matrix.

    [**5 pts**]

14. Compute the reconstruction error between the mean centered data matrix X and reconstructed data matrix (you may use the *sklearn.metrics.mean_squared_error* function).

    [**5 pts**]

15. Perform steps 11 – 14 for the following values of *k*: 10, 30, 50, 100, 500. For each *k*, show the reconstructed image (use the matplotlib.pyplot *imshow* function with the reconstructed data matrix for each *k*). With each reconstructed image print the value of *k* and the reconstruction error.

    [**10 pts**]