<Machine Learning HW #5>

21300008 강민수

I. Experimental Environment

1. 사용된 Data와 Hyperparameter

Dataset은 Train data 50,000장, Test data 10,000장으로 구성된 MNIST dataset을 사용하였다. LDA와 PCA를 써서 data를 projection하였고 LDA와 PCA 각각의 경우에 대해 KNN과 Random Forest를 사용해서 학습을 진행한 후 결과를 비교하였다. 실험에 사용된 hyper parameter는 아래와 같고 아래의 hyperparameter의 다양한 조합들로 결과를 분석하였다.

(1) Eigenspace dimension: 2, 3, 5, 9

(2) kNN 수행 시 k: 1, 3, 5, 10

2. 사용된 코드

1) kNN

```
for test_img_idx in np.ndindex(self._test_x.shape[:1]):

        distance = np.linalg.norm(self._test_x[test_img_idx] - self._train_x, axis=1)
        # calculate distances from one test data point to train data samples

        sorted_indices = np.argsort(distance)
        # sort index of distances in increasing order

        pred_class = stats.mode(self._train_y[sorted_indices[:k]])[0][0]
        # get candidate class labels and select the mode value of them

        if self._test_y[test_img_idx] == pred_class:
                accuracy += 1
        # calculate acc
```

kNN을 구현한 코드는 위와 같다. 첫번째 코드는 매 iteration마다 한 개의 test sample이 뽑혀서 전체 train data에 각각의 train sample과의 Euclidean distance를 구한다. 그리고 그 다음 두 코드에서 낮은 순서대로 distance를 갖도록 정렬하는데 그 때의 index를 k개만큼 가져와 (distance가 낮은 순서에 대응되는 index를 가져와서) 그 index에 해당하는 predicted label들을 뽑아낸다. predicted label들 중 가장 많이 나온 값이 kNN model이 예측한 label이 된다. 마지막 코드에서 predicted label과 test sample의 원래 정답과 비교해 맞춘 개수를 count한 후 학습이 종료되면 test에 사용된 sample수로 나누어 정확도를 산출한다.

2) PCA

```
def PCA(self, eigenspace_dim):

        '''
                do PCA analysis
        '''
        cov_matrix = np.cov((self._train_x - np.mean(self._train_x, axis = 0)).T)
        eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
        eigenvectors =eigenvectors.T[:eigenspace_dim].T
        # do eigen-decomposition

        projectioned_train = np.dot(self._train_x, eigenvectors)
        projectioned_test = np.dot(self._test_x, eigenvectors)
        # project data onto n-eigenspace

        return (projectioned_train, projectioned_test)
        # project data on eigenspace
```

PCA를 수행하는 코드는 위와 같이 구현되었다. train에 사용될 image vector들을 그 평균으로 빼고 그것을 대상으로 각 feature 변수의 covariance를 구한다. 그리고 covariance matrix를 eigen decomposition하여 eigenvector를 구한 후 원하는 만큼의 eigenvector를 이용해 eigenspace를 구성한다. 그리고 train과 test를 이 eigenspace에 projection하여 kNN 알고리즘을 적용하기 위한 준비를 마친다.

3) LDA

```
def LDA(self, eigenspace_dim):
        '''
                do LDA analysis
        '''

        class_number = 10
        class_feature_number = np.array([])
        variance_within_class = np.zeros(784*784).reshape(-1, 784)
        variance_between_class = np.zeros(784*784).reshape(-1, 784)

        global_mean = np.mean(self._train_x, axis = 0)

        for i in np.ndindex(class_number):
                class_features = self._train_x[self._train_y == i]
                class_mean = np.mean(class_features, axis = 0)
                # calculate mean of each class

                for j in np.ndindex(class_features.shape[0]):
                        features_minus_class_mean = (class_features[j] - class_mean).reshape(784, 1)
                        variance_within_class += np.dot(features_minus_class_mean, features_minus_class_mean.T)
                        # calculate variance within class to minimize variance between samples in class

                dist_between_class_data = (class_mean - global_mean).reshape((784, 1))
                variance_between_class += (class_features.shape[0] * np.dot(dist_between_class_data, dist_between_class_data.T))
                # calculate variance between classes to maximize variance between classes in sample space

        eigenvalues, eigenvectors = np.linalg.eig(np.dot(np.linalg.pinv(variance_within_class), variance_between_class))
        # eigen-decompose (SW)^(-1) X (SB)
        eigen_pairs = [(np.abs(eigenvalues[i]), eigenvectors[:,i]) for i in range(len(eigenvalues))]
        eigen_pairs = sorted(eigen_pairs, key=lambda k: k[0], reverse=True)
        eigenvectors = np.array([eigen_pairs[i][1] for i in range(eigenspace_dim)]).T
        # sort (eigenvalue, eigenvector) pair

        projected_train = np.dot(self._train_x, eigenvectors)
        projected_test = np.dot(self._test_x, eigenvectors)
        # project train and test samples

        return projected_train, projected_test
```

LDA를 수행하는 코드는 위와 같이 구현되었다. 먼저 Global mean을 구하고 각 class에 속해 있는 data들의 mean들을 구한다. 그리고 그것들을 대상으로 class 내부에 속해 있는 data point들의 분산과 class간의 분산을 구한 후, class 내부에 속해 있는 data point들의 분산에 역행렬을 취한 것과 class간의 분산을 곱하고 이를 eigen decomposition한다. 높은 Eigenvalue를 갖는 순서대로 eigenvector를 sorting하고 거기에서 원하는 dimension만큼 eigenvector를 뽑아내어 전체 train data와 test data를 projection한다.

4) Random Forest

```
def random_forest_run(self, train_number, test_number, eigenspace_dimension, mode):
        '''
                do Random Forest
        '''
        self._data_preprocessor(train_number = train_number, test_number = test_number, eigenspace_dimension = eigenspace_dimension, mode = mode)
        start_time = time.time()
        clf = RandomForestClassifier()
        clf.fit(self._train_x.real, self._train_y)
        acc = np.mean(np.equal(clf.predict(self._test_x.real), self._test_y))
```

Random Forest는 sklearn에 구현되어 있는 API를 이용하였다.

II. Experimental Result and Analysis

1. kNN 알고리즘 수행시

(1) k=1

Eigenspace dimension	2	3	5	9
PCA	0.3765	0.439	0.6885	**0.8909**
LDA	**0.465**	**0.6556**	**0.7873**	0.8587

(2) k=3

Eigenspace dimension	2	3	5	9
PCA	0.3987	0.4559	0.7258	**0.9049**
LDA	**0.5025**	**0.6917**	**0.8147**	0.8711

(3) k=5

Eigenspace dimension	2	3	5	9
PCA	0.4201	0.4788	0.7245	**0.9119**
LDA	**0.5229**	**0.7128**	**0.826**	0.8765

(4) k=10

Eigenspace dimension	2	3	5	9
PCA	0.4346	0.4993	0.7587	**0.9137**
LDA	**0.545**	**0.7271**	**0.8355**	0.8776

2. Random Forest 알고리즘 수행시

Eigenspace dimension	2	3	5	9
PCA	0.4137	0.478	0.7377	0.8803
LDA	**0.5078**	**0.7203**	**0.8327**	**0.9027**

3. Analysis (my opinion of experimental result)

kNN의 경우 k가 1, 3, 5, 10이고 eigenspace의 dimension이 2, 3, 5일 때, LDA를 수행한 것이 훨씬 더 성능이 높았다. 반대로, k가 1, 3, 5, 10이고 eigenspace가 9일 때는 PCA가 성능이 더 높은 것을 알 수 있었다. eigenspace dimension이 2, 3, 5, 9일 때 LDA를 통해 projection한 data들로 random forest를 수행한 것이 PCA보다 성능이 높았다.

PCA Analysis에서 covariance matrix를 eigen decomposition하여 얻은 eigenvalue들과 LDA Analysis에서 $(S_w)^{-1}S_B$ 를 eigen decomposition하여 얻은 eigenvalue들은 전체 dataset의 정보를 얼마나 담고 있는지를 나타낸다. PCA의 경우에는 전체 dataset의 공분산을 그대로 이용하기 때문에 eigenvalue의 값의 편차가 LDA보다 적게 나타날 것이다. 하지만 LDA는 클래스 간의 분산(S_B)을 최대화되고, 클래스 내부에 있는 data sample들 간의 분산(S_W)이 최소화되는 공간에 data들을 projection하는 것이기 때문에 $(S_w)^{-1}S_B$를 eigen decomposition하여 거기서 절대값이 가장 큰 eigenvalue를 갖는 eigenvector들에 projected하게 되면, C-1개만큼의 eigenvalue들에 대응되는 eigenvector들에 정보들이 집중되게 된다. 그 중에서 dimension을 2와 3, 5, 9만큼 사용하였다면, MNIST가 10개의 class를 갖기 때문에 9차원의 dimension을 갖는 eigenspace에서 전체 dataset의 대부분 정보들이 집약되게 될 것이다. 그렇게 되면, eigenspace dimension 9인 경우에, LDA를 수행할 때에는 data를 나타내는데 필요 없는 noise들도 포함되게 되어 PCA보다 정확도가 떨어질 것이다. 그래서 kNN의 경우에는 eigenspace dimension이 9인 경우 PCA가 LDA보다 높은 성능을 기록하였고, eigenspace dimension이 2, 3, 5인 경우에는 LDA가 PCA보다 높은 성능을 기록한 것으로 판단된다.

```
def _check_variance_percentage(self, eigenvalues, eigenspace_dim, mode):
        percentages = eigenvalues / np.sum(eigenvalues)
        total_percentages = np.sum(eigenvalues[:eigenspace_dim]) / np.sum(eigenvalues)
        print("mode: ", mode, "\tvariance percentages:", list(percentages[:eigenspace_dim]), "\t total percentage:", total_percentages)
```

이를 증명하기 위해 eigenvalue를 절댓값이 큰 순서대로 나열한 후 위의 코드를 이용하여 전체 eigenvalue들 중에서 각각의 eigenvalue들이 몇 %를 각각 차지하는지와 eigenvalue 2개, 3개, 5개, 10개의 분산들이 전체 몇 퍼센트를 차지하는지를

계산해 보았다. (계산할 때 eigenvalue는 절댓값을 취해서 계산하였다.) 그 결과는 아래의 표와 같이 나타났다.

Eigenvalue	1	2	3	4	5	6	7	8	9
PCA	9.744	7.059	6.216	0.538	4.858	4.319	3.277	2.886	2.768
LDA	23.97	20.23	17.94	10.63	9.427	6.798	4.941	3.391	2.642

표: 전체 eigenvalue들에서 각각의 eigenvalue가 차지하는 비중 (단위: %)

Eigenspace Dimension	2	3	5	9
PCA	16.8042	23.0203	33.2579	46.5103
LDA	44.2155	62.1635	82.2222	99.9961

표: 전체 eigenvalue들 중에서 eigenspace dimension만큼의 eigenvalue들이 차지하는 비중 (단위: %)

Random Forest는 entropy를 통해 decision을 결정하는 여러 개의 decision tree model을 구성하여 다양한 파라미터 조합을 고려하고 각 tree에 대해 독립적으로 학습을 수행해 voting 또는 weighted sum의 형태로 model을 산출한다. 하지만 충분한 수의 feature들이 있어야 tree의 depth가 깊어질 수 있고, 그에 따라 정교하게 data를 나누는 것이 가능해진다. 하지만 LDA와 PCA를 수행한 후에 Random Forest를 수행하게 되면 eigenspace dimension이 낮을수록 전체 data들을 구성하는 feature들의 수가 줄어들게 되고 따라서 kNN에 비해 성능이 감소하는 폭이 클 수 밖에 없다는 결론을 얻게 된다. 실험 결과에서도 알 수 있듯이, k=5, 10이고 eigenspace dimension이 2, 3일 때의 kNN의 성능이, (같은 eigenspace dimension을 갖는 data에 대해) random forest를 수행한 것보다 훨씬 더 좋다는 것을 알 수 있다.