빅데이터 분석

MNIST 숫자 데이터를 이용한 분류 모델 학습 및 평가

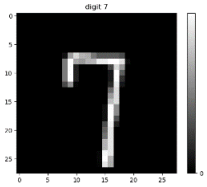
산업정보시스템공학과

20162263 박윤아

0. 데이터 확인 및 전처리

- pd.read_csv() 함수를 통해 데이터를 받아오고, info() 함수를 이용해 데이터의 정보를 확인하였다. 그 결과 42000행, 785열, int64형 변수임을 알았다. (NULL 값은 없다.)

- 해당 데이터는 0~255값을 갖는 데이터이고, pixel 정보는 최소, 최대값이 정해져 있기 때문에 0-1 normalization 수행이 필요함을 알았고, 이는 사용 함수 중 load_mnist() 함수에 포함 되어있다.



(50번째 데이터를 그림으로 나타낸 결과)

1. 사용할 함수 정리

– 'UtilForHw'에 사용할 함수들을 따로 정리해 두었다. 사용함수는 아래와 같다.

- getxfold(data, turn, fold): data를 train데이터와 test데이터로 나눠 반환한다.

- runCVshuffle(clf, data, labels, fold=10, isAcc=True): data와 labels를 shuffle하여 getxfold 함수를 통해 train용 data와 labels, test용 data와 labels로 나눈 후 해당 classifier로cross validation 수행 후 isAcc 값에 따라 accuracy 또는 precision, recall, f1score, support 값을 반환한다.

- load_mnist(path): 해당 path에서 mnist데이터를 불러오는 함수로, data를 255로 나눠주는 정규화 과정이 포함 되어있다.

2. 데이터 분석

2-0 기본적인 데이터 분석과정은 아래와 같고, 각각의 classifier만 수정하여 수행하였다. (2.12 Grid Search 제외)

1) 해당 classifier 생성 2) runCVshuffle 함수를 통해 accuracy 값 받아오고 평균 구하기 3) runCVshuffle 함수를 통해 precision, recall, f1score, support 값을 받아와서 평균 구하기

- 이때, 받아온 precision, recall, f1score, support 값의 평균 구하는 방법은 새롭게 구현한 것으로 사용한 코드는 아래와 같다. (decision tree에서의 사용 예시)

```
prfs_dt = runCVshuffle(clf_dt, d_mn, l_mn, isAcc=False)
prfs_dt_np = np.array(prfs_dt)
m_prfs_dt = []
for j in range(4):
    x = 0
    for i in range(10):
        x += sum(prfs_dt_np[i,j])
    m_prfs_dt.append(x/100)
```

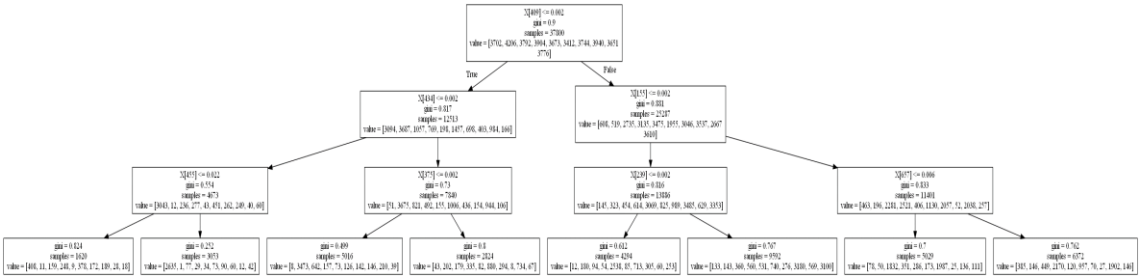
2-1 Prediction: 불가능. 수치 예측이 아니다.

2-2 Decision Tree

-default 값을 이용해 구한 값

```
>>> mean_acc_dt
0.8594285714285714
>>> m_prfs_dt
[0.8564433206484613, 0.8565694195861855, 0.8563042601029396, 420.0]
```

-그래프를 그려본 결과 트리의 깊이가 깊고 분할이 너무 많아서, max_depth=2로 함수 내부를 수정해 그래프를 그려보았다.



- 보다 간략하게 그래프로 나타낼 수 있으나, 이 경우 accuracy는 약 0.45로 낮다.

2-3 Naïve Bayesian Classifier

-default 값을 이용해 구한 결과

```
>>> mean_acc_nb
0.5557619047619048
>>> m_prfs_nb
[0.548101985582886, 0.6691172979180721, 0.5039062756577968, 420.0]
```

2-4 k-Nearest Neighbor Classifier

-default 값을 이용해 구한 결과

```
>>> mean_acc_knn
0.9680238095238096
>>> m_prfs_knn
[0.9675858117844851, 0.9685758626302595, 0.9678523803619474, 420.0]
```

2-5 Logistic Regression

-default 값을 이용해 구한 결과

```
>>> mean_acc_lr
0.919452380952381
>>> m_prfs_lr
[0.9183638879529297, 0.9186032554327679, 0.9183460076890495, 420.0]
```

- 함수 내부의 C값을 2로 변경하여 수행해 본 결과

```
>>> mean_acc_lr2
0.9190000000000002
>>> m_prfs_lr2
[0.9179000380143237, 0.9181482726937983, 0.9178936225869634, 420.0]
```

2-6 Perceptron

-max_iter=500, n_jobs=3을 넣어 구한 결과

```
>>> mean_acc_pc
0.8741666666666668
>>> m_prfs_pc
[0.8723974684078359, 0.8823254284739329, 0.8720300768072674, 420.0]
```

- max_iter=500, n_jobs=3, 학습률인 eta0를 0.1로 변경하여 수행해 본 결과

```
>>> mean_acc_pc2
0.8764285714285714
>>> m_prfs_pc2
[0.8750196807013667, 0.8841941714950354, 0.8745778487748741, 420.0]
```

2-7 Multi-layer Perceptron

-hidden_layer_sizes=20, max_iter=50을 넣은 후의 결과

```
>>> mean_acc_mlp
0.9429285714285716
>>> m_prfs_mlp
[0.9435228798543294, 0.9435886706556735, 0.9434843322685837, 420.0]
```

- hidden_layer_sizes=20, max_iter=50, learning_rate_init=0.1, activation='tanh'을 넣어 구한 결과

```
>>> mean_acc_mlp2
0.9023095238095238
>>> m_prfs_mlp2
[0.902780122911208, 0.9038098157363867, 0.9026394607341621, 420.0]
```

2-8 Random Forest

-default 값을 이용해 구한 결과

```
>>> mean_acc_rf
0.9653571428571428
>>> m_prfs_rf
[0.9648617072002201, 0.9648471124585717, 0.9648103373346157, 420.0]
```

2-9 Linear Discriminant Analysis

- n_component=2로 하여 구한 결과

```
>>> mean_acc_lda
0.8648809523809525
>>> m_prfs_lda
[0.8635916083659528, 0.8661122174671568, 0.8636971753309463, 420.0]
```

- n_component=8로 하여 결과

```
>>> mean_acc_lda2
0.8648809523809525
>>> m_prfs_lda2
[0.8635916083659528, 0.8661122174671568, 0.8636971753309463, 420.0]
```

2-10 Quadratic Discriminant Analysis

-default 값을 이용해 구한 결과

```
>>> mean_acc_qda
0.18904761904761905
>>> m_prfs_qda
[0.19083290464995206, 0.3101436101976841, 0.11163113858926466, 420.0]
```

2-11 Support Vector Machine

-default 값을 이용해 구한 결과

```
>>> mean_acc_lsvc
0.9115238095238096
>>> m_prfs_lsvc
[0.9101058488181977, 0.9102793298290186, 0.9100078599370651, 420.0]
```

2-12 Grid Search

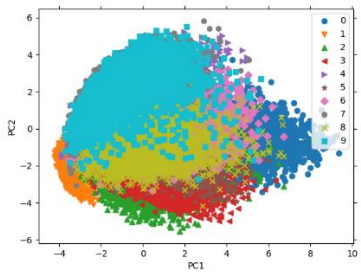
-kernel이 linear, rbf, polynomial, sigmoid인 경우의 코드를 모두 작성하였으나, 수행시간이 너무 많이 소요되어 kernel이 linear인 경우의 결과만 작성하였다.

-kernel이 linear 일 때 0.001, 0.01, 0.1, 1, 10, 100 중에 C값은 0.1이 가장 좋으며, 이때 평균 accuracy는 0.941이다.

```
0.919 for {'kernel': 'linear', 'C': 0.001}
0.937 for {'kernel': 'linear', 'C': 0.01}
0.941 for {'kernel': 'linear', 'C': 0.1}
0.930 for {'kernel': 'linear', 'C': 1.0}
0.920 for {'kernel': 'linear', 'C': 10.0}
0.914 for {'kernel': 'linear', 'C': 100.0}
```

2-13 Principal Components Analysis & Map to New Space using PC

-sklearn 라이브러리를 통해 explained variance ratio를 구한 후 2차원으로 축소하였다.



(차원이 많이 축소되어 의미를 도출하기 어렵다.)

3. 결과표 작성

	acc	precision	recall	f1score	Support
dt	0.859	0.856	0.857	0.856	420
nb	0.556	0.548	0.669	0.504	420
knn	0.968	0.968	0.969	0.968	420
lr	0.919	0.918	0.919	0.918	420
Lr2	0.919	0.918	0.918	0.918	420
Pc	0.874	0.872	0.882	0.872	420
Pc2	0.876	0.875	0.884	0.875	420
mlp	0.943	0.944	0.944	0.943	420
mlp2	0.902	0.903	0.904	0.903	420
rf	0.964	0.965	0.965	0.965	420
lda	0.865	0.864	0.866	0.864	420
lda2	0.865	0.864	0.866	0.864	420
qda	0.189	0.191	0.310	0.112	420
lsvc	0.912	0.910	0.910	0.910	420

- k-Nearest Neighbor Classifier를 사용한 경우의 결과가 모든 수치에 있어서 가장 좋다.