Machine Learning HW2

202211167 장지원

Setting

1. DataSet

sklearn라이브러리의 fetch_openml 함수를 가져와서 MNIST dataset을 load해 보았다.

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784')
```

2. Data Preprocessing

먼저 값을 0~1사이로 정규화 하였다.

```
X = mnist.data / 255.0
```

데이터의 라벨을 처리해 주었다.

```
y = mnist.target.astype('int')
```

data들을 train set과 test set으로 나누어 주었다. 이 때 전체 데이터의 20%를 test set에 할당하였다.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Model Training

1. For MNIST data set, train *Logistic regression models* and find the best model that can achieve the highest accuracy on the test data set.

LogisticRegession 함수의 파라미터는 밑과 같다.

class `sklearn.linear_model.`**`LogisticRegression`**(*penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None*)

그 중 기계학습개론 수업 시간에 배운 penalty과 C param을 바꿔보며 best model을 찾아볼 것이다. 그 외 나머지 값들은 default를 사용했다. 이 때 penalty는 Regularization 종류를 결정하는 param이고, C는 Regularization의 강도를 결정하는 param이다.

먼저 C값을 1.0으로 고정했을 때의, Regularization 종류에 따른 정확도를 알아볼 것이다. 우리가 default 값으로 사용한 solver param의 *lbfgs* solver는 L2, none 두 가지의 penalty를 지원한다. 두 가지 penalty에 대한 결과는 다음과 같다. L2 norm을 사용한다면 불필요한 feature 사용을 줄일 수 있어 overfit 문제를 막을 수 있다.

penalty / Accuracy(%)	L2 norm	None
	92.08	91.72

L2 norm에서의 정확도가 더 높은 것을 확인할 수 있다. 이는 test data에 대한 과적합이 None param일 때 보다 덜 되었기 때문이라 결론지을 수 있다. 이후 penalty를 가장 정확도가 높았던 L2 norm으로 설정하고(고정하고), C값을 1.0과 0.1로 설정하여서 학습을 진행시켜 보았다. 결과는 다음과 같다.

C / Accuracy(%)	C=1.0	C=0.1
	92.08	92.11

C=0.1일 때 더 높은 정확도를 얻을 수 있었고, 이는 위 에서와 마찬가지로 test data에 대한 과적합이 C=1.0일 때 보다 덜 되었기 때문이라 결론지을 수 있다.

이를 통해 찾은 best model은 다음과 같다.

penalty를 'l2', C를 0.1로 설정했을 때 가장 좋은 결과(92.11)를 얻을 수 있었다.

```
model = LogisticRegression(penalty='l2', max_iter=1000, C=0.1)
```

2. **For the same data set, train *K-NN classifiers* and find the best model that can achieve the highest accuracy on the test data set.**

KNeighborsClassifier 함수의 파라미터는 밑과 같다.

class sklearn.neighbors.**KNeighborsClassifier**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None*)

그 중 기계학습개론 수업 시간에 배운 n_neighbors param을 바꿔보며 best model을 찾아볼 것이다. weight는 함수의 default인 uniform을 이용하였고, 그 외의 나머지 값들은 default를 사용하였다. 이 때 n_neighbors param은 몇 개의 근처 neighbor을 참고할지에 대한 parameter이다. 이 값이 크면 accuracy가 줄어들게 되고, 이 값이 작으면 noise에 예민해지게 된다.

n_neighbor에 개수에 따른 model의 정확도는 다음과 같다.

n_neighbors / Accuracy(%)	n_neighbors=1	n_neighbors=3	n_neighbors=5	n_neighbors=7	n_neighbors=9
	97.20	97.13	97.01	96.87	96.58

n_neighbors이 낮을수록 Accuracy가 올라가는 것을 확인할 수 있다. 이는 우리가 사용한 MNIST dataset에 noise가 없었기 때문일 것이라는 결론을 내릴 수 있다.

이를 통해 찾은 best model은 다음과 같다.

n_neighbors를 1로 설정했을 때 가장 좋은 결과(97.20)를 얻을 수 있었다.

```
model = KNeighborsClassifier(n_neighbors=1)
```

(추가로 cross validation을 이용하면 model에 적합한 k값을 더 쉽고, 정확하게 찾을 수 있다)

3. **For the same data set, train *SVM classifiers* and find the best model that can achieve the highest accuracy on the test data set.**

SVC 함수의 파라미터는 밑과 같다.

class sklearn.svm.**SVC**(**, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', break_ties=False, random_state=None*)

그 중 기계학습개론 시간에 배운 C와 kernel param을 바꿔보면서 best model을 찾아볼 것이다. 이 때 C는 fitting 강도를 설정해주는 파라미터이고, kernel은 kernel trick을 위해 사용할 함수를 결정해주는 파라미터이다.

먼저 C값을 1.0으로 고정했을 때의, kernel 함수 종류에 따른 정확도를 알아볼 것이다. kernel 함수는 computationally expensive를 해결하기 위해서 사용한다. (precomputed kernel 함수는 input으로 정방행렬을 요하기 때문에 따로 test하지 않았다)

kernel / Accuracy(%)	linear	poly	rbf	sigmoid	precomputed
	93.51	97.39	97.64	77.65	X

이후 kernel을 가장 정확도가 높았던 rbf로 설정하고(고정하고), C값을 1.0과 0.1로 설정하여서 학습을 진행시켜 보

았다. 결과는 다음과 같다.

C / Accuracy(%)	C=1.0	C=0.1
	97.64	95.58

C=1.0일 때 더 높은 정확도를 얻을 수 있었고, 이는 C=0.1일 때 보다 더 학습했기 때문이라 예측할 수 있다. C=1.0일 때는 아직 overfit한 경우가 아닌 것이다.

이를 통해 찾은 best model은 다음과 같다.

kernel를 'rbf', C를 1.0로 설정했을 때 가장 좋은 결과(97.64)를 얻을 수 있었다.

```
model = SVC(C=0.1, kernel='rbf')
```

4. **For the same data set, train Random forest classifiers and find the best model that can achieve the highest accuracy on the test data set.**

RandomForestClassifier 함수의 파라미터는 밑과 같다.

class sklearn.ensemble.**RandomForestClassifier**(*n_estimators=100, *, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='sqrt', max_leaf_nodes=None, min_impurity_decrease=0.0, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None, monotonic_cst=None*)

Random Forest는 Decision Tree의 Bagging 앙상블 메서드이다. Random Forest에서는 여러 개의 Tree의 평균값을 이용해 예측을 하는 방법을 사용해서 과적합을 막는다. 이번에는 n_estimators param을 바꿔보면서 best model을 찾아볼 것이다. 이 때 n_estimators는 사용할 Decision Tree의 개수를 결정해 주는 파라미터이다.

n_estimators를 10, 100, 500, 1000으로 하여 학습을 진행해 보았다. 결과는 다음과 같다.

n_estimators / Accuracy(%)	10	100	500	1000
	94.58	96.74	96.83	96.88

n_estimators를 가장 크게 설정했을 때 가장 좋은 결과를 얻을 수 있었고, 이는 Tree를 많이 참조할수록 train data에 overfit되는 현상이 점차 사라지기 때문이다. 단 학습 시간과 Tree의 개수는 trade off 관계이다. 즉 정확도는 올라가지만 학습 시간은 느려지는 것이다. 위 Table을 보면 Tree가 100, 500, 1000일 때의 정확도가 서로 거의 유사함을 확인할 수 있다. 따라서 Tree 100개일 때의 결과가 trade off를 잘 고려한 이상적 결과라 할 수 있다.

이를 통해 찾은 best model은 다음과 같다.

n_estimators를 1000으로 설정했을 때 가장 좋은 결과(96.88)를 얻을 수 있었다.

```
model = RandomForestClassifier(n_estimators=1000)
```

그러나 이상적인 model은 n_estimators를 100으로 설정했을 때의 model일 것이다.

5. **or the same data set, train Gradient Boosting classifiers and find the best model that can achieve the highest accuracy on the test data set.**

GradientBoostingClassifier 함수의 파라미터는 밑과 같다.

class sklearn.ensemble.**GradientBoostingClassifier**(**, loss='log_loss', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0*)

Grandient Boosting는 Decision Tree의 Boosting 앙상블 메서드이다. GD Boosting 방식에서는 하나의 Tree를 보완하는 방식으로 학습이 이루어진다. 이번에도 마찬가지로 n_estimators param을 바꿔보면서 best model을 찾아볼 것이다.

n_estimators를 10, 100으로 하여 학습을 진행해 보았다. 결과는 다음과 같다. (500, 1000은 학습이 너무 오래 걸려서 따로 학습시키지 않았다)

n_estimators / Accuracy(%)	10	100	500	1000
	84.14	94.54	X	X

n_estimators를 가장 크게 설정했을 때 가장 좋은 결과를 얻을 수 있었고, 이는 Tree를 많이 참조할수록 train data에 overfit되는 현상이 사라지기 때문이다. 단 학습 시간과 Tree의 개수는 trade off 관계이다. 즉 정확도는 올라가지만 학습 시간은 느려지는 것이다.

이를 통해 찾은 best model은 다음과 같다.

n_estimators를 100으로 설정했을 때 가장 좋은 결과(94.54)를 얻을 수 있었다.

```
model = GradientBoostingClassifier(n_estimators=100)
```

Best Result Table

	Logistic Regression	K-NN	SVM	Random Forest	Gradient Boosting
Accuracy	92.11	97.20	97.64	96.88	94.54

Discussion

model을 실행할 때 test set에 대한 정확도도 함께 출력했다면 overfitting 정도를 쉽게 판단할 수 있었을 것이다. 마지막 GD Boosting에서 학습 시간이 너무 오래 걸려 파라미터를 두 번 밖에 바꾸지 않았다. XG Boosting은 GD Boosting을 병렬 연산 처리해서 더 나은 속도를 제공하는 알고리즘이다. 이를 사용하면 더 빠르게 결과를 얻을 수 있었을 것이다.