# 기계학습개론 과제2 리포트

202011047 김승태

본 코드들을 시작하기 전, install 해야만 하는 package들을 설명하도록 하겠다. 과제에서 Scikit-learn library를 이용하라 하였으니, !pip install scikit-learn을 이용하였다. 이번 과제는, 혹여나 노트북으로 돌렸을 때, 성능이 저하되어 오래 걸리는 것을 방지하기 위해서 ipynb 형태로 colab에서 진행되었음을 알리는 바이다. 링크는 다음과 같다. Ipynb 파일도 따로 첨부하였다.

여기에서는, 성능을 위하여서 20%만 사용하였고 ( test_size = 0.2 , random state는 42로 고정하여서, 코드를 다시 돌렸을 때에도 동일하게끔 하도록 하였다. )

https://colab.research.google.com/drive/1MPwml2_qgCgcY28VFVRzlsbOf-Q-zY8Z?usp=sharing

또한, 첫번째 코드 셀에서, mord와 xgboost도 사용하였는데, 이는 나중에 특수 classifier를 이용하기 위하여서 추가로 install해 주었다. 이제부터, 각 문제별 활용 코드들과, 그 결과를 설명하겠다.

## 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.

결과를 분석하기 위해서, 다음과 같은 세 가지 Logistic regression model들을 사용하였다.

사용한 model 종류 : Binary Logistic Regression model, Multinomial Logistic Regression Model, Ordinal Logistic Regression Model, 아래는 Multinomial Logistic Regression Model코드이다.

```
#Multinomial Logistic Regression model
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

digits = load_digits()

scaler = StandardScaler()
X_scaled = scaler.fit_transform(digits.data)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, digits.target, test_size=0.2, random_state=42)

logistic_model = LogisticRegression(C=0.2, penalty='l2', solver='lbfgs', multi_class='multinomial')
logistic_model.fit(X_train, y_train)

y_pred = logistic_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of Multinomial Logistic Regression model on test set: {:.2f}%".format(accuracy * 100))

```

```
Accuracy of Multinomial Logistic Regression model on test set: 97.50%
```

결과 분석 : Multinomial Logistic Regression Model과 Binary Logistic Regression Model은 성능에 차이를 보이지 않았다. 여기에서는 C는 정규화 강도를 조정하는 매개변수로 쓰이면서, C값에 따라서 정규화를 어떻게 적용할 지 결정할 수 있다. C 값이 줄어들수록, 더 많은 정규화를 적용하기 때문에, 더 높은 accuracy를 주로 제공하였다. 그러나, 0.2 미만으로 줄이니, 오히려 accuracy가 떨어지는 경향성이 존재하였고, 이는 정규화의 강도가 너무 커져 모델이 너무 단순해져서일 것이라고 생각하였다. Penalty를 l2로 지정하여, L2 정규화를 사용하여 가중치 벡터 제곱 항을 추가하였으며, lbfgs 라는 최적화 알고리즘을 채택하였다. ( 이는 다른 결과들을 참조하여 선택되었다. ) 그리고, binary는 지정하지 않았지만, Multinomial Logistic Regression Model에서는 Multi_classf를 multinomial로 지정하여 돌아가도록 설정하였다. 각각의 accuracy는 97.5%이었다.

아래는, Ordinal Logistic Regression Model code이다.

```python
#Ordinal Logistic Regression model
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from mord import LogisticAT
from sklearn.metrics import accuracy_score

digits = load_digits()

scaler = StandardScaler()
X_scaled = scaler.fit_transform(digits.data)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, digits.target, test_size=0.2, random_state=42)

ordinal_model = LogisticAT(alpha=1000)
ordinal_model.fit(X_train, y_train)

y_pred = ordinal_model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of Ordinal Logistic Regression model on test set: {:.2f}%".format(accuracy * 100))

```

여기서 Ordinal Logistic Regression Model을 사용하기 위해서, Mord에서 LogisticAT를 가져와 프로그래밍하였다. 위와 달리, 여기에서는 alpha 가 c의 역할을 하고 있기 때문에, 이 값을 조정하면서 최적의 결과를 도출해 내려 하였다. 최적의 값은 1000이었다. (이보다 크거나 작으면 정확도가 낮아지는 결과를 보였다.) 하지만, 이 데이터는 다중 클래스가 순서형으로 배치되어 있는 구조가 아니기 때문에 이 모델은 적합하지 않은 듯 하였다. 결과적으로 25.28%의 가장 저조한 accuracy를 보였다.

**결론 : Binary,Multinomial Logistic Regression Model : 97.5%**

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

결과를 분석하기 위해서, 다음과 같은 세 가지 K-NN classifiers들을 사용하였다.

사용한 classifier 종류 : K-NN classifier, weighted K-NN classifer, Radius-based K-NN classifier
아래는 그 중, weighted K-NN classifier 코드이다.

```python
#Weighted K-NN classifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

digits = load_digits()

scaler = StandardScaler()
X_scaled = scaler.fit_transform(digits.data)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, digits.target, test_size=0.2, random_state=42)

knn_model = KNeighborsClassifier(n_neighbors=5, weights='distance')
knn_model.fit(X_train, y_train)

y_pred = knn_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of weighted K-NN classifier on test set: {:.2f}%".format(accuracy * 100))

```

```
Accuracy of weighted K-NN classifier on test set: 97.50%
```

결과 분석 : K-NN과 weighted K-NN은 성능에 차이를 보이지 않았다. 여기에서는 n_neighbors는 고려하는 이웃의 수이다. 위 코드에서는 5로 지정하였기 때문에 주변 5개의 이웃값을 참조하여 결정한다. n_neighbor의 값이 너무 작으면, 과적합이 일어나는 결과를 보이고 있다. 1로 지정하였을 때는, 97.78%라는 굉장한 수치를 자랑하였다. 거의 과적합된 결과이다. 2, 3, .. 도 실시해 보았으나, 그 중에서는 5가 가장 훌륭한 accuracy 결과값을 보였다. 그리고, K-NN에 weights를 추가한 것이 weighted K-NN인데, weights를 distance로 지정하여, 거리의 역수를 가중치로 사용하였는데, 이는 neighbor의 값이 작아서 그런 것인지, 오차가 거의 없는 것으로 보였다. 즉, 이 둘의 accuracy는 과적합을 고려하지(신경쓰지) 않았을 때 97.78%, 과적합을 배제하였을 때 97.5%의 결과값을 보였다.

그 다음은, Radius-based KNN이다. 이는 radius(반경)을 neighbor 수 대신에 설정하는 것이다.

```python
#Radius-based K-NN classifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import RadiusNeighborsClassifier
from sklearn.metrics import accuracy_score

digits = load_digits()

scaler = StandardScaler()
X_scaled = scaler.fit_transform(digits.data)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, digits.target, test_size=0.2, random_state=42)

radius = 9.0
knn_model = RadiusNeighborsClassifier(radius=radius)
knn_model.fit(X_train, y_train)

y_pred = knn_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of radius-based K-NN classifier on test set: {:.2f}%".format(accuracy * 100))

```

```
Accuracy of radius-based K-NN classifier on test set: 66.67%
```

많은 시도 끝에, 9.0 이라는 radius값이 유효한 accuracy를 낸다는 것을 알 수 있었다. 8.xx대는 거의 error code가 났으며, 이는, 참조할 neighbor가 그 반경 내에 없어서 그렇게 되었음을 짐작할 수 있었다. 반경이 커질수록 accuracy가 급격하게 줄어드는 것을 볼 수 있었고, 이에 따라, 9라는 값으로 지정하였을 때, accuracy는 66.67%가 되었다.

**결론 : KNN, weighted KNN : 97.78% ( 97.5%)**

## 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.

이는, linear, non-linear로 나누어서 두 가지로 진행하였다.

```
[62]  1 #SVM classifier
      2 from sklearn.datasets import load_digits
      3 from sklearn.model_selection import train_test_split
      4 from sklearn.preprocessing import StandardScaler
      5 from sklearn.svm import SVC
      6 from sklearn.metrics import accuracy_score
      7
      8 digits = load_digits()
      9
     10 scaler = StandardScaler()
     11 X_scaled = scaler.fit_transform(digits.data)
     12
     13 X_train, X_test, y_train, y_test = train_test_split(X_scaled, digits.target, test_size=0.2, random_state=42)
     14
     15 svm_model = SVC(kernel='linear', C=0.01)
     16 svm_model.fit(X_train, y_train)
     17
     18 y_pred = svm_model.predict(X_test)
     19 accuracy = accuracy_score(y_test, y_pred)
     20 print("Accuracy of linear SVM classifier on test set: {:.2f}%".format(accuracy * 100))
     21

Accuracy of linear SVM classifier on test set: 97.78%
```

```
 1 #non-linear SVM classifier
 2 from sklearn.datasets import load_digits
 3 from sklearn.model_selection import train_test_split
 4 from sklearn.preprocessing import StandardScaler
 5 from sklearn.svm import SVC
 6 from sklearn.metrics import accuracy_score
 7
 8 digits = load_digits()
 9
10 scaler = StandardScaler()
11 X_scaled = scaler.fit_transform(digits.data)
12
13 X_train, X_test, y_train, y_test = train_test_split(X_scaled, digits.target, test_size=0.2, random_state=42)
14
15 svm_model = SVC(kernel='rbf', C=1.0, gamma='scale')
16 svm_model.fit(X_train, y_train)
17
18 y_pred = svm_model.predict(X_test)
19 accuracy = accuracy_score(y_test, y_pred)
20 print("Accuracy of non-linear SVM classifier (RBF kernel) on test set: {:.2f}%".format(accuracy * 100))
21

Accuracy of non-linear SVM classifier (RBF kernel) on test set: 98.06%
```

위는 그 둘의 코드와 최적의 결과값인데, 이는 1번에서와 같이 c로 정규화 정도를 조절한 것이다. Non-linear에서는 kernel을 rbf, Radial Basis Function을 이용하였으며, 입력 space를 무한한 space로 매핑하여 실행시킨다. 각각 c는 0.01, 1이 최적의 accuracy를 제공하였으며, 이보다 작으면 갑자기 값이 튀거나, 작아지거나 하였고, 크면 또 작아지는 결과를 나타내었기에, 이 것들로 잡았다.

각각 결과는 97.78%, 98.06%이었다. 그리고, non-linear에서 gamma는 rbf커널의 폭을 지정하는데, scale로 지정하면 자동으로 값을 설정해 주는 것이다.

**결론 : non-linear SVM classifier : 98.06%**

## 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.

```python
#Random Forest classifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

digits = load_digits()

X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size=0.2, random_state=42)

rf_model = RandomForestClassifier(n_estimators=1000, random_state=42)
rf_model.fit(X_train, y_train)

y_pred = rf_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of Random Forest classifier on test set: {:.2f}%".format(accuracy * 100))

```

```
Accuracy of Random Forest classifier on test set: 97.78%
```

```python
#Extra Trees classifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.metrics import accuracy_score

digits = load_digits()

X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size=0.2, random_state=42)

extra_trees_model = ExtraTreesClassifier(n_estimators=1000, random_state=42)
extra_trees_model.fit(X_train, y_train)

y_pred = extra_trees_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of Extra Trees classifier on test set: {:.2f}%".format(accuracy * 100))

```

```
Accuracy of Extra Trees classifier on test set: 98.06%
```

사용한 것은 Random forest와, Extra Trees classifier이다. Random forest는 의사결정 트리를 구성할 때, 임의의 특성 집합에서 최적의 분할을 선택하는데, extra trees는 먼저 무작위로 특성을 선택해서 분할을 먼저 수행한 후, 이 중에서 최적의 분할을 선택하는 것이 다르다. 위 코드들에서는 특별한 파라미터가 n_estimators밖에 없는데, 이는 몇 개의 Decision tree를 생성할 것인지를 결정하는 것이다. 즉, 이것이 많을수록 모델이 복잡해지면서 성능은 증가할 것이다. 실제로 이 수치를 증가시킬수록 실행시간이 증가함을 쉽게 볼 수 있었다. 그 중, 1000에서 성능이 둘 다 좋은 것으로 보였으며, 사실 Extra trees는 100, 1000에서 차이가 없음을 보아, 100을 사용하였을 때, 더 효과적일 것이라고 생각하였다. 두 코드의 차이는 y_pred에 다른 모델을 사용하였다는 것이다.

**결론 : Extra Trees Classifier : 98.06%**

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

결과를 분석하기 위해서, 다음과 같은 세 가지 K-NN classifiers들을 사용하였다.

사용한 classifier 종류 : Gradient Boosting classifier, XGBoost classifier, Stochastic Gradient Boosting classifier 아래는 GB code이다.

```python
#GB
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score

digits = load_digits()

X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size=0.2, random_state=42)

gbm_model = GradientBoostingClassifier(n_estimators=500, learning_rate=0.1, random_state=42)
gbm_model.fit(X_train, y_train)

# Evaluate the model
y_pred = gbm_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of Gradient Boosting Machine classifier on test set: {:.2f}%".format(accuracy * 100))

```

```
Accuracy of Gradient Boosting Machine classifier on test set: 97.50%
```

먼저, GB는 n_estimator, learning_rate라는 특수한 파라미터를 가지고 있으며, 이는 각각 n_estimator는 부스팅 단계의 수를 의미하며, 더 많이 사용할수록 모델의 복잡성과 수행 시간은 늘어나지만, 성능이 향상된다. 여기에서는 500을 지정하였는데, 100부터 이전 400까지는 accuracy에 변화가 없다가, 500부터 600 사이에서 가장 좋은 accuracy를 보여주었다. 그러므로, 500으로 가장 효과적인 값을 찾아내었다. learning_rate는 학습률이며, 이는 0과 1사이의 값으로, 작을수록 더 안정적으로 수렴하고, 일반화 성능이 향상된다. 줄였을 때, 성능이 증가한 수치는 0.1이었으며, 0.05, 0.01같은 더욱 더 작은 수로 바꾸었을 때, 수행 시간이 급격하게 늘어났으며, 정확도도 떨어지는 결과값을 볼 수 있었다. Accuracy는 97.5%이다. (아래는 XGBoost 코드이다.)

```python
#XGBoost
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score

digits = load_digits()

X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size=0.2, random_state=42)

xgb_model = XGBClassifier(learning_rate=0.1, n_estimators=1000, random_state=42)
xgb_model.fit(X_train, y_train)

y_pred = xgb_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of XGBoost classifier on test set: {:.2f}%".format(accuracy * 100))

```

```
Accuracy of XGBoost classifier on test set: 97.22%
```

그 다음으로는, XGBoost이다. 이는, 파라미터의 의미는 GBM과 거의 유사하다고 볼 수 있다. 하지만, 이 것의 최적 수치는 달랐다. n_estimators 값이 1000이 되었을 때, accuracy가 올라가는 것을 볼 수 있었다. Accuracy는 97.22%이다.

마지막으로, Stochastic GB이다. 부스팅 과정에서 일부 데이터를 무작위로 선택해서 사용한다는 점이 다른 점이다. 아래가 그 코드이다.

```
#Stochastic GB
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score

digits = load_digits()

X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size=0.2, random_state=42)

sgb_model = GradientBoostingClassifier(n_estimators=1000, subsample=0.8, random_state=42)
sgb_model.fit(X_train, y_train)

y_pred = sgb_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of Stochastic Gradient Boosting classifier on test set: {:.2f}%".format(accuracy * 100))

```

```
Accuracy of Stochastic Gradient Boosting classifier on test set: 98.06%
```

여기에서는, n_estimator를 동일하게 사용했지만, subsample이 다르다는 점이 있다. n_estimator는 1000이었을 때 가장 좋은 결과를 보여주었으며, subsample은 훈련 데이터의 몇 퍼센트를 가져올 것인지를 설정하는 것이다. 이것으로 모델의 다양성을 증가시키고, 과적합을 방지하는 역할을 한다. 여기에서는 각 부스팅 단계에서 80%저도의 훈련 데이터를 사용하여 나타내었다. 70%, 90%에서 더 낮은 accuracy를 보여주었기 때문에 이를 선택하였다. 즉, 너무 적어도, 너무 많아도 SGB accuracy가 좋지 않음을 알 수 있었다. 결과적으로 98.06%라는 좋은 결과를 보여주었다.

**결론 : Stochastic GB: 98.06%**

**All the best results ( 최종적 가장 높은 accuracy % 표시한 표 )**

| | Logistic Regression | K-NN | SVM | Random Forest | Gradient Boosting |
|---|---|---|---|---|---|
| Accuracy | **97.5%** | **97.78%** **( 97.5%)** | **98.06%** | **98.06%** | **98.06%** |

결론이 무언가 이상하다. SVM과 Random Forest, Gradient Boosting이 동일한 결과가 나왔다는 것이다. 물론, 파라미터 값을 accuracy 결과값만 가지고 변경하였던 것이 가장 큰 요인인듯 싶다. 이렇게 되면, 98.06%는 이 모델이 과적합되어 있다는 것을 의미하는 것이 아닐까라는 의심이 든다. 즉, 나올 수 있는 최대의 accuracy이자, 과적합 기준이라는 것이다. 물론, 정확하지는 않다. 결과적으로, 각각의 방법들은 서로 다 비슷한 수치의 정확성을 보여 주었으며, 그 방법들 사이에서도, 이 데이터셋에는 적합하지 않은 형태여서 다른 것들과의 차이가 큰 것도 있었다는 것을 알 수 있

었다. 위의 결과(인위적으로 끌어낸 최적값)이 아닐 때에는 97.5%라는 값이 가장 많이 나왔으며, 이 또한 무언가를 의미하는게 아닐까 싶지만, 아직 이 수준에서는 이해할 수 없다는 점이 아쉬웠다.