지정 *Project*

Fashion MNIST

컴퓨터공학과

20155137

안원영

목차

Project

구성을 크게 4가지로 나누었다.

1. 데이터를 읽어온 뒤, 그 데이터가 어떤지 분석한다.

2. 가져온 데이터를 이용해 3가지의 분류기 (SVM, DecisionTree , RandomForest) 를 통해 각각의 최고성능을 구한다.

3. 가장 좋은 분류기를 사용하여 오차행렬을 구해본다.

4. 많은 오류가 발생했던 부분의 데이터를 plot해보고 이유를 분석한다.

Chapter 1

다중분류기 vs 2진분류기

다중분류기

- 다중분류기
  - 다중분류기는 두 개 이상의 클래스를 구별하는 것이다.
- 다중분류 가능 모델
  - 딥러닝
  - 결정트리
  - 랜덤포리스트
  - 나이브베어즈
  - 로지스틱회기
  - 신경회로망

분류와 회귀

- 분류
  - Class를 예측하는 것
  - 이진분류, 다항분류로 나뉜다.
- 회귀
  - 연속적인 값으로부터 특징을 추출

2진분류기

- 2진분류기
  - 두 개의 클래스를 구분하는 것이다.
- 이진분류 가능 모델
  - 선`형분류기
  - SVM

```
1 from tensorflow import keras
2 fashion_mnist = keras.datasets.fashion_mnist
3 (X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
4 X_train.shape, type(X_train), y_train.shape, type(y_train)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
32768/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
((60000, 28, 28), numpy.ndarray, (60000,), numpy.ndarray)
```

Chapter 2

데이터 읽기

- 먼저 사용할 데이터를 읽어온다. 여기서 사용할 데이터는 **Pashion MNIST** 데이터 이다.
- **Keras**의 장점은 크게 3가지가 있다.
  - 사용자 친화적으로, 오류에 대한 피드백 제공
  - 모듈식 및 구성으로 , 빌딩블록을 연결하는 방식으로 진행
  - 새로운 레이어와 함수를 쉽게 확장
- 데이터는 총 6만개, 28*28 픽셀의 형태로 저장되어있다.

```
1 X = X_train[:1000].reshape(-1, 784)
2 y = y_train[:1000]
```

```
1 X.shape, y.shape
```

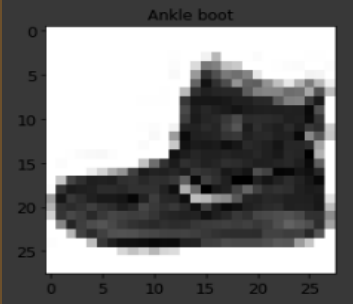
```
((1000, 784), (1000,))
```

```
1 cnt = 0
2
3 for i in class_names:
4   print(cnt,"번 인덱스 이름 : ",i)
5   cnt += 1
6
```

```
0 번 인덱스 이름 :  T-shirt/top
1 번 인덱스 이름 :  Trouser
2 번 인덱스 이름 :  Pullover
3 번 인덱스 이름 :  Dress
4 번 인덱스 이름 :  Coat
5 번 인덱스 이름 :  Sandal
6 번 인덱스 이름 :  Shirt
7 번 인덱스 이름 :  Sneaker
8 번 인덱스 이름 :  Bag
9 번 인덱스 이름 :  Ankle boot
```

```
1 %matplotlib inline
2 import matplotlib as mpl
3 import matplotlib.pyplot as plt
4
5 #data가 섞여서 들어가있다. 0번째에는 class_names 9번째의 ankle boot가 있다.
6 idx = 0
7 some_digit = X[idx]
8 some_digit_image = some_digit.reshape(28,28)
9 some_digit_label = y[idx]
10 plt.imshow(some_digit_image, cmap=mpl.cm.binary)
11 plt.title(class_names[some_digit_label])
```

```
Text(0.5, 1.0, 'Ankle boot')
```



Chapter 3

데 이 터 분 석

- 데이터를 다루기 쉽게 6만개의 데이터중 1000개의 데이터만 가지고 온다.
- X에는 1000개의 28*28의 그림이 들어있는데,계산하기 쉽도록 형태를 일렬인 784로 바꿔준다.
- y에는 데이터의 라벨들이 저장되어 있다.
- 어떤 데이터가 들어가 있는지 구체적으로 보기 위해 x의 첫번째 값을 plot해봤다.
- X의 첫번째에는 Ankle boot의 그림이 들어가 있는것을 확인할 수 있다.

Chapter 4

SVM시 작

SVM

- SVM은 '서포트 벡터 머신'의 줄임말로,마진을 최대화 하는 결정경계(특징을 나누는 경계)를 설정해야 한다.
- 마진이 작다면 조금의 변화에도 민감하게 반응하해서 일반화에 좋지 않음.
- 특성스케일에 영향을 많이 받기 때문에 StandardScaler 이용해 정규화 해주는것이 좋음
  - 정규화 : x0와 x1의 값들이 일정해지면서 샘플들이 재대로 구분된다.
- SVM은 SVC(분류)와 SVR(회귀)로 나눌 수 있다.
  - 여기서는 SVC가 성능 더 좋음
- SVM의 단점
  - 선형으로 분리가 안되는 경우 사용하지 못함
- SVM의 장점
  - 선형,비선형 문제에 모두 적용가능
  - 분류, 회귀 문제에 모두 적용 가능
  - 복잡도를 조정할 수 있다.
  - 수학적으로 정의가 잘 되어있다.
- 따라서 마진을 가장 크게 가지는 서포트 벡터를 구하여 일반화 좋게 만드는 것이 목표이다.

```python
def pipe_nomal(kernel_name):
  if kernel_name == "default":#default 면실행
    model = Pipeline([
        ("svm_clf", SVC(degree=3, coef0=1 ,C=10, gamma='scale'))# 모델 만들때 커널 설정
    ])
  elif kernel_name == "rbf_scaler":#StanderdScaler 적용시킬 rbf일경우 실행
    model = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", degree=3, coef0=1 ,C=10, gamma='scale'))
    ])
  else:#linear, poly, sigmoid 일 경우 실행
    model = Pipeline([
        ("svm_clf", SVC(kernel=kernel_name, degree=3, coef0=1 ,C=10, gamma='scale'))
    ])

  model.fit(X, y)
  cross_nomal = cross_val_score(model,X,y,cv=3,scoring="accuracy")
  print(kernel_name,"에 교차검증 적용: ", cross_nomal)
  model_accuracy = model.score(X_test,y_test)
  print( "test_model_accuracy : ", model_accuracy)
  cross_nomal_mean = cross_nomal.mean()

  return cross_nomal_mean
```

SVM-kernel

- 커널이란, 실제 다항 특징을 추가하지 않고 비슷한 효과를 만드는 수학적 트리이다.
- kernel 사용하면 자동으로 다차항으로 바꿔서 분류를 진행한다.
- 다차항으로 바꿔주는 이유는 복잡도를 높여 특징을 잘 분류해내기 위해서 이다.
- Kernel의 종류
  - Linear
  - Poly
  - Sigmoid
  - RBF (가우시안) -디폴트

SVM-kernel

```
default_accuracy_mean = pipe_nomal("default")#kernel이 rbf로 설정됨
print("kernel=default: ",default_accuracy_mean)
print("############################################################")
print()

linear_accuracy_mean = pipe_nomal("linear")
print("kernel=linear: ",linear_accuracy_mean)
print("############################################################")
print()

poly_accuracy_mean = pipe_nomal("poly")
print("kernel=poly: ",poly_accuracy_mean)
print("############################################################")
print()

rbf_accuracy_mean = pipe_nomal("rbf")
print("kernel=rbf: ",rbf_accuracy_mean)
print("############################################################")
print()

sigmoid_accuracy_mean = pipe_nomal("sigmoid")
print("kernel=sigmoid: ",sigmoid_accuracy_mean)
print("############################################################")
print()

rbf_scaler_accuracy_mean = pipe_nomal("rbf_scaler")
print("StandardScaler 적용 후 kernel=rbf_scaler: ",rbf_scaler_accuracy_mean)
print("############################################################")
print()
```

```
default 에 교차검증 적용:  [0.82335329 0.84084084 0.83483483]
test_model_accuracy :  0.831
kernel=default:  0.8330096563629498
############################################################

linear 에 교차검증 적용:  [0.79341317 0.8018018  0.8018018 ]
test_model_accuracy :  0.785
kernel=linear:  0.7990055924187661
############################################################

poly 에 교차검증 적용:  [0.80538922 0.83183183 0.81981982]
test_model_accuracy :  0.798
kernel=poly:  0.8190136244028446
############################################################

rbf 에 교차검증 적용:  [0.82335329 0.84084084 0.83483483]
test_model_accuracy :  0.831
kernel=rbf:  0.8330096563629498
############################################################

sigmoid 에 교차검증 적용:  [0.4011976  0.37237237 0.31231231]
test_model_accuracy :  0.36
kernel=sigmoid:  0.36196076315836795
############################################################

rbf_scaler 에 교차검증 적용:  [0.80538922 0.8018018  0.81381381]
test_model_accuracy :  0.813
StandardScaler 적용 후 kernel=rbf_scaler:  0.8070016123908344
############################################################
```

```
model1 = Pipeline([
        ("svm_clf", SVC(kernel="rbf",degree=3, coef0=1 ,C=10, gamma='scale'))
    ])
model1.fit(X,y)
print("중축모델 : ",model1.score(X_test,y_test))


model2 = Pipeline([
        ("svm_clf", SVC(kernel="rbf",degree=3, coef0=1 ,C=10000, gamma='scale'))
    ])
model2.fit(X,y)
print("C=10000으로 한 모델(복잡도 증가) : ",model2.score(X_test,y_test))


model3 = Pipeline([
        ("svm_clf", SVC(kernel="rbf",degree=100, coef0=1 ,C=10, gamma='scale'))
    ])
model3.fit(X,y)
print("dgree=100으로 한 모델(복잡도 증가) : ",model3.score(X_test,y_test))

model4 = Pipeline([
        ("svm_clf", SVC(kernel="rbf",degree=3, coef0=1 ,C=10, gamma=0.01))
    ])
model4.fit(X,y)
print("간마=0.01 로 한 모델(분산 커짐) : ",model4.score(X_test,y_test))
```

```
중축모델 :  0.831
C=10000으로 한 모델(복잡도 증가) :  0.824
dgree=100으로 한 모델(복잡도 증가) :  0.831
간마=0.01 로 한 모델(분산 커짐) :  0.095
```

SVM-성능 올리기

- 성능올리는 방법 **3**가지
  - **C**
  - **Dgree**
  - **Gamma**
- **C**
  - 규제와 연관
  - **C**값 증가 **->** 규제 감소 **->** 복잡도 증가
- **Dgree**
  - 다차항 차수와 연관
  - 증가할수록 복잡도 증가
- **Gamma**
  - 분산과 연관
  - 간마가 커질수록 분산 작아짐 **->** 복잡도 증가
- 따라서 간마와 **C, Dgree**를 적절히 조절해 사용

Chapter 5

결정트리 시작

Decision Tree

- 결정트리란 화이트박스 형태로, 한가지 특징을 이용해 **2**개의 노드로 분리해가며 불순도와 **MSE**를 줄이는 것이 목표이다.
- 결정트리
  - **DecisionTreeRegressor (**회귀**)**
  - **DecisionTreeClassifier(**분류**)**
  - 결정트리는 회귀로 푸는것이 분류보다 성능이 더 좋았다.
- **MSE**와 불순도
  - **MSE : Mean Squared Error** 의 줄임말로, 정답에 대한 오류를 숫자로 나타낸다.
  - 불순도 : 노드의 샘플 클래스가 얼마나 분산되어 있는지를 측정한다.
- 결정트리에서의 분류와 회귀 차이
  - 회귀 : 평균과 실제값의 오차**(MSE)**를 줄이는 것을 목표로 한다.
  - 분류 : 불순도를 줄이는 것을 목표로 한다.
- 장점
  - 규칙을 표현하는데 가장 적합한 모델이다.
  - 굉장히 빠르다.
- 단점
  - 훈련세트의 변화에 많이 민감하다.
  - 과대적합이 잘 일어나 복잡도를 줄여줘야 한다.
- 따라서 성능을 높이기 위해 적절한 규제를 사용하여 복잡도를 줄여야 한다.

결정트리 성능높이기

규제

- **Max_leaf_node**
  - 리프 노드의 최대수
  - 증가할수록 더 세부적으로 특징이 셋팅되도록 함

```
def learning(n):
  tree_reg = DecisionTreeRegressor(random_state=42,max_leaf_nodes=n)
  tree_reg.fit(X, y)
  train_accuracy = 0
  test_accuracy = 0
  train_accuracy = tree_reg.score(X,y)
  test_accuracy = tree_reg.score(X_test,y_
  a = (train_accuracy-test_accuracy)#정확도
  print("max_leaf_nodes=",n, ", 학습데이터정확도=",train_accuracy,", 시험데이터정확도=",test_accuracy,", 차이=",a)


leaf_nodes_10 = learning(10)
leaf_nodes_100 = learning(100)
leaf_nodes_1000 = learning(1000)
```

```
max_leaf_nodes= 10 , 학습데이터정확도= 0.7848451625865887 , 시험데이터정확도= 0.6874649498014843 , 차이= 0.09738021278510434
max_leaf_nodes= 100 , 학습데이터정확도= 0.9930648639442261 , 시험데이터정확도= 0.6116655813174816 , 차이= 0.3813992826267445
max_leaf_nodes= 1000 , 학습데이터정확도= 1.0 , 시험데이터정확도= 0.600469686436881 , 차이= 0.39953031356311897
```

결정트리 성능높이기

규제

- **Min_sample_split**
  - 분할되기 위해 노드가 가져야할 최소 샘플 개수

- 과적합이 일어나지 않았다. -> 규제할 필요 없음
- 데이터셋이 더 컸다면 규제를 통해 성능을 높일 수 있었을 것이다.

```
def learning(n):
  tree_reg = DecisionTreeRegressor(random_state=42,max_leaf_nodes=1000,min_samples_split=n)
  tree_reg.fit(X, y)
  train_accuracy = 0
  test_accuracy = 0
  train_accuracy = tree_reg.sco
  test_accuracy = tree_reg.scor
  a = (train_accuracy-test_accu
  print("min_samples_split=",n,
        ", 시험데이터정확도=",t

samples_split_10 = learning(10)
samples_split_126 = learning(100)
samples_split_1000 = learning(1000)#데이터 1000개밖에 없기 때문에
print("###############최적###############")
samples_split_126 = learning(126)
```

```
min_samples_split= 10 , 학습데이터정확도= 0.9871747287870107 , 시험데이터정확도= 0.6164916586000239 , 차이= 0.3706830701869869
min_samples_split= 100 , 학습데이터정확도= 0.8353583595234649 , 시험데이터정확도= 0.7131313863706805 , 차이= 0.12222697315278441
min_samples_split= 1000 , 학습데이터정확도= 0.5068109906313804 , 시험데이터정확도= 0.516961645228159 , 차이= -0.010150654596778663
###############최적###############
min_samples_split= 126 , 학습데이터정확도= 0.8125931093727936 , 시험데이터정확도= 0.7140761639242653 , 차이= 0.0985169454485282
```

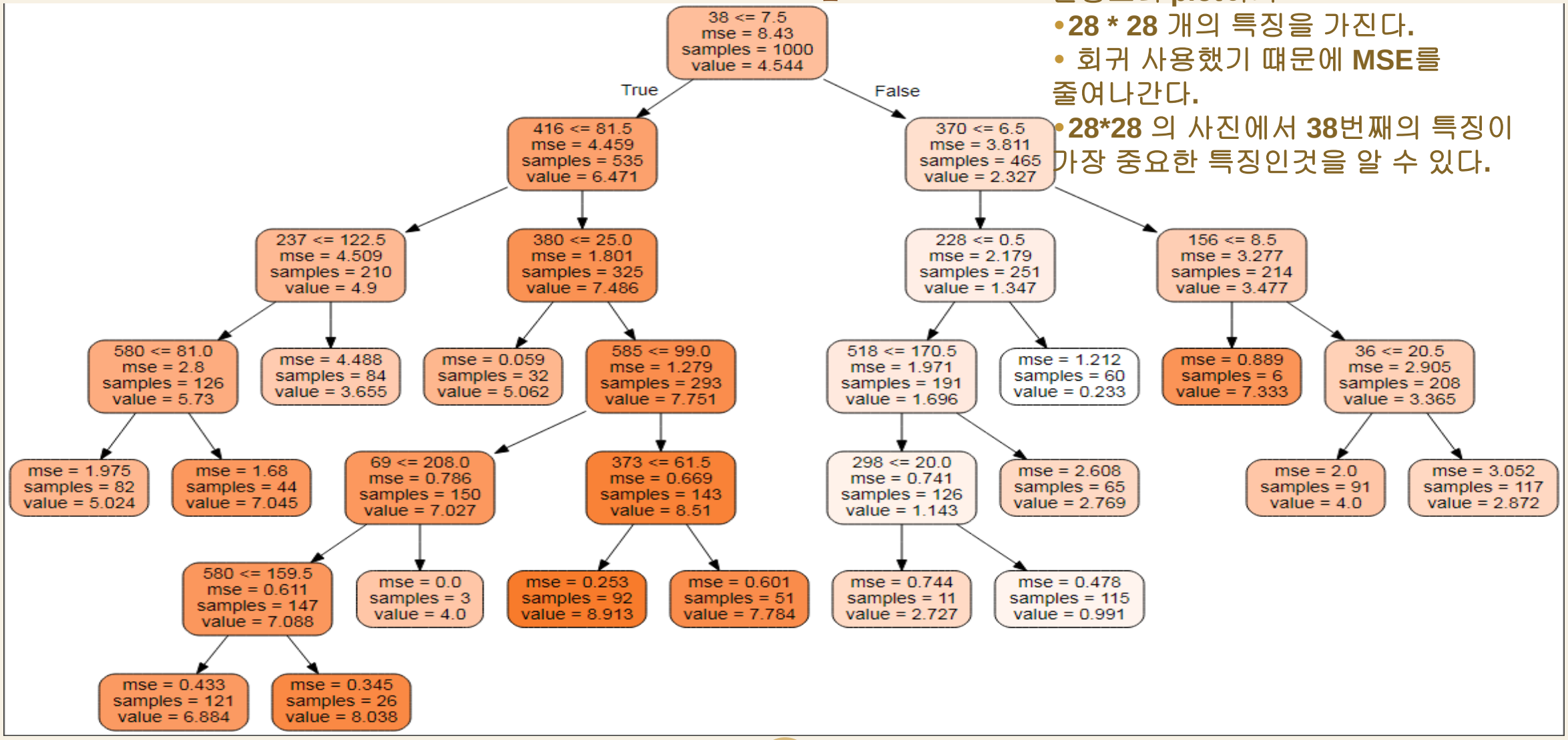
```
from sklearn.tree import export_graphviz# 결정함수 그리는 함수
from graphviz import Source

tree_reg = DecisionTreeRegressor(random_state=42,max_leaf_nodes=1000,min_samples_split=126)
tree_reg.fit(X, y)

a = []
for i in range(784):
  a.append(i)

export_graphviz(
        tree_reg,
        out_file=os.path.join(IMAGES_PATH, "classifi_tree.dot"),
        feature_names=a,
        rounded=True,
        filled=True
    )
Source.from_file(os.path.join(IMAGES_PATH, "classifi_tree.dot"))
```

결정트리 plot

- **Export_graphyiz** 패키지 이용해서 결정트리 **plot**하기

결정트리 plot

- **Export_graphyiz** 패키지 이용해서 결정트리 **plot**하기
- **28 * 28** 개의 특징을 가진다.
- 회귀 사용했기 때문에 **MSE**를 줄여나간다.
- **28*28** 의 사진에서 **38**번째의 특징이 가장 중요한 특징인것을 알 수 있다.



Chapter 6

Random Forest

- 랜덤포레스트는 결정트리의 앙상블로, 성능이 아주 뛰어난 분류기이다.
  - 랜덤 : 무작위로 뽑는다.
  - 포레스트 : 숲(결정트리)의 모임
  - 따라서 랜덤포레스트는 결정트리의 앙상블이다.

- 앙상블 : 여러 개의 모델을 하나의 모델로 합한 것이다.
  - 다양한 분류기를 학습한 뒤, 각 분류기 예측 결과를 합하여 최종 결과를 결정한다.

- 장점
  - 특징의 중요도를 알 수 있다.
  - 특징으로 분할할때 지니 불순도를 계산하는데, 그때 그 특징이 불순도를 얼마나 낮추는지 계산할 수 있다.

```
###### 분류기 5개 만들기 ######
model_Decision = DecisionTreeClassifier(random_state=42)
model_Randomf = RandomForestClassifier(random_state=42)
model_SVC = SVC(probability=True,random_state=42)
modee_voting_hard = VotingClassifier(
    estimators=[('dc', model_Decision), ('rf', model_Randomf), ('svc', model_SVC)],
    voting='hard')
modee_voting_soft = VotingClassifier(
    estimators=[('dc', model_Decision), ('rf', model_Randomf), ('svc', model_SVC)],
    voting='soft')
###### 분류기 만들기 끝 ######
```

```
def learning(model,model_name):
  model.fit(X,y)
  y_pred = model.predict(X_test)
  print(model_name,accuracy_score(y_test, y_pred))

learning(model_Decision,"
learning(model_Randomf,"R
learning(model_SVC,"SVC")
learning(modee_voting_har
learning(modee_voting_sof
```

```
Decision 0.678
RandomForest 0.805
SVC 0.786
Voting_hard 0.801
Voting_soft 0.741
```

앙상블과 RandomForest

- 분류기 모으는 방법 2가지
  - **Voting = hard ->** 결과의 클래스 이용
  - **Voting = soft ->** 결과의 확률이용
  - 원래는 **soft**가 더 잘 나오는데, 분류라 **hard**가 더 잘나옴

- 랜덤포레스트가 **Voting** 방법보다 성능 좋은 이유
  - 랜덤포레스트는 자체로 앙상블이다.
  - **Voting**은 낮은 성능을 내는 분류기와 섞여있다.

RandomForest 성능 높이기-1

```
def learning(n):
  model_RandomF = RandomForestClassifier(random_state=42,max_leaf_nodes=n)
  model_RandomF.fit(X, y)
  train_accuracy = 0
  test_accuracy = 0
  train_accuracy = model_RandomF.score(X,y)
  test_accuracy = model_RandomF.score(X_test,y_test)
  a = (train_accuracy-test_accuracy)#정확도 차이
  print("min_samples_split=",n, ", 학습데이터정확도=",train_accuracy,
        ", 시험데이터정확도=",test_accuracy,", 차이=",a)


learning(10)
learning(100)
learning(1000)
```

```
min_samples_split= 10 , 학습데이터정확도= 0.771 , 시험데이터정확도= 0.729 , 차이= 0.04200000000000004
min_samples_split= 100 , 학습데이터정확도= 0.999 , 시험데이터정확도= 0.804 , 차이= 0.1949999999999995
min_samples_split= 1000 , 학습데이터정확도= 1.0 , 시험데이터정확도= 0.804 , 차이= 0.1959999999999995
```

- 과적합이 일어난것을 알 수 있다.
  - 규제 통해서 성능 높일 수 있다.
- 규제 이용해서 성능 높이기
  - **Min_sample_split :** 최소 샘플 갯수
  - **Max_features :** 최대 특징
  - **Max_depth :** 최대 깊이
- 규제 사용해도 결과 비슷했다.
- 영향을 많이주는 **n_estimator** 를 사용해서 성능 높이기로 함

```
# model_RandomF = RandomForestClassifier(n_estimators = 1000, random_state=42, max_leaf_nodes=1000,
#                                        min_samples_split=61, max_features=100, max_depth=10)#estimator =1000 ->75.9%
 # model_RandomF = RandomForestClassifier(n_estimators = 500, random_state=42, max_leaf_nodes=1000,
 #                                       min_samples_split=61, max_features=100, max_depth=10)#sample_split갯수 증가 ->75%
 model_Rand# model_RandomF = RandomForestClassifier(n_estimators = 500, random_state=42, max_leaf_nodes=1000,
 train_accu#                                      min_samples_split=2, max_features=700, max_depth=10)#특징갯수 변화-> 78%
 test_accuracy # model_RandomF = RandomForestClassifier(n_estimators = 500, random_state=42, max_leaf_nodes=1000,
 train_accuracy = model_Rar# model_RandomF = RandomForestClassifier(n_estimators = 500, random_state=42, max_leaf_nodes=1000,
 test_accuracy = model_Ranc#                                      min_samples_split=2, max_features=10, max_depth=10)#depth=10 -> 80.6%
 a = (train_accuracy-test_accuracy)#정 # model_RandomF = RandomForestClassifier(n_estimators = 500,random_state=42,max_leaf_nodes=1000,
 print('RandomForest oob_score_ :',mode#                                      min_samples_split=2,max_features=10,max_depth=100)#81%
       train_accuracy,", test 데이터정확# model_RandomF = RandomForestClassifier(n_estimators = 500, random_state=42, max_leaf_nodes=1000)# 81.2%
                                        # model_RandomF = RandomForestClassifier(n_estimators = 1000, random_state=42)# 81.3%
learning()                              # model_RandomF = RandomForestClassifier(n_estimators = 3000, random_state=42)# 81.3%
                                        # model_RandomF = RandomForestClassifier(n_estimators = 2000, random_state=42)# 81.4%
```

t 성능높이기 -2

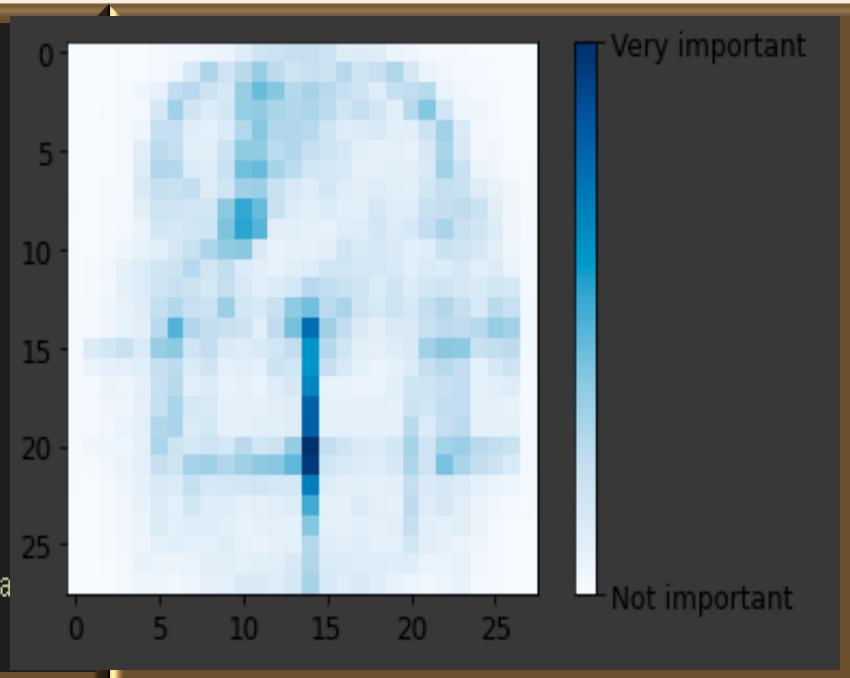
수이다.

때문에 사용되지 않은 샘플이 존재

없음

```
RandomForest oob_score_ : 0.831 , train 데이터정확도= 1.0 , test 데이터정확도= 0.814 , 차이= 0.18600000000000005
```

```
import matplotlib as mpl

def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = mpl.cm.Blues, interpolation="nearest")
    plt.axis("on")

model_RandomF = RandomForestClassifier(n_estimators=2000, oob_score=True, random_state=42)
model_RandomF.fit(X,y)

plot_digit(model_RandomF.feature_importances_)# 중요도를 출력

cbar = plt.colorbar(ticks=[model_RandomF.feature_importances_.min(), model_RandomF.feature_importances_.ma
cbar.ax.set_yticklabels(['Not important', 'Very important'])
```



RandomForest로 특징 중요도 분석

- Feature_importances_ 으로 중요도 파악하기
- 20 * 14 정도의 특징이 가장 중요하다는 것을 알 수 있다.

Chapter 7

오차 행렬

오차 행렬 구현

<Confusion Matrix error>

- 데이터셋은 정확도 만으로 평가하기 어렵기 때문에 오차 행렬을 이용하여 정밀도와 재현율을 봐야한다.
- 오차행렬이란, 분류 모델의 성능을 평가하기 위해 사용하는 것으로, 모델의 예측이 얼마나 맞고 틀렸는지 구분시켜 준다.
  - 오차행렬 보는 방법 => 대각선을 기준으로, 대각선이 맞춘 개수다.
- 재현율
  - 더 많이 포함시켜 확률 내는 것
- 정밀도
  - 검출된 것 중에서 진짜인 것
- 재현율과 정밀도 관계
  - 재현율 높이면 정밀도 낮아짐

0과 6에서 오류가 가장 많음

```
11
12 # confusion(오차행렬) 구현
13 from sklearn.model_selection import cross_val_predict
14 from sklearn.metrics import confusion_matrix
15
16 def confu_matrix(X,Y,high_fit,name):
17   scaler = StandardScaler()
18   high_fit_train_scaler = scaler.fit_transform(X.astype(np.float64))#forest를 Scaler해줌
19   y_train_pred = cross_val_predict(high_fit, high_fit_train_scaler, Y, cv=3)#예측한 값을 저장
20   cm = confusion_matrix(Y, y_train_pred)#오차행렬로 표시
21   return cm#오차행렬을 리턴
22
23 cm = confu_matrix(X,Y,high_fit,name)#오차행렬을 리턴받음
24 print(cm)
```

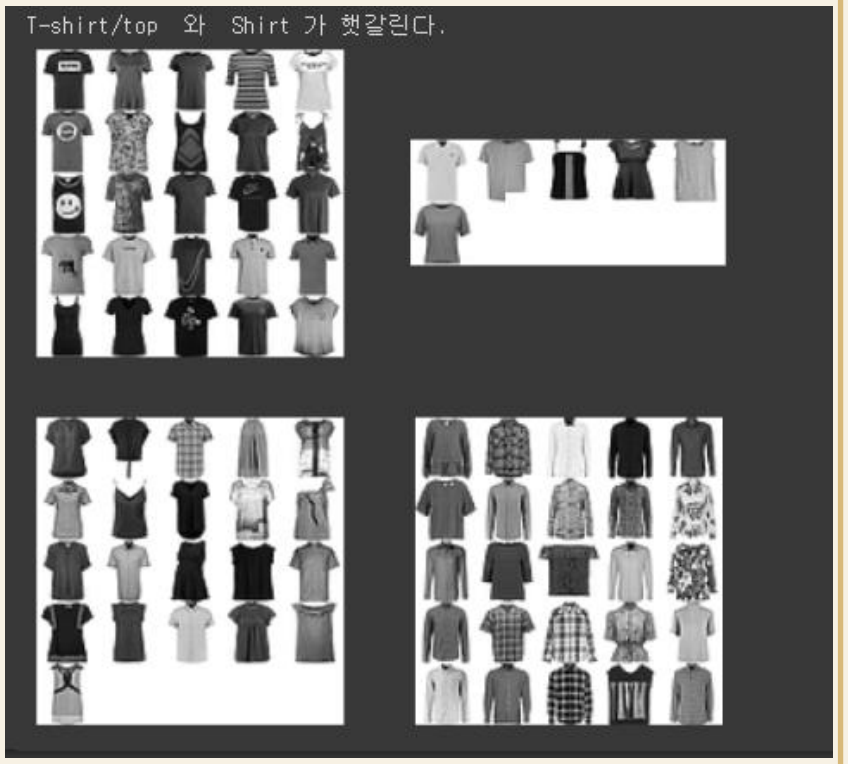
```
[[ 89   0   3   7   0   0   6   0   2   0]
 [  0  94   2   6   0   0   2   0   0   0]
 [  0   0  52   1  17   0  16   0   0   0]
 [  4   1   1  79   2   0   5   0   0   0]
 [  1   0  13   6  70   0   5   0   0   0]
 [  0   0   0   0   0  92   0   3   1   4]
 [ 21   2  11   1  13   2  46   0   4   0]
 [  0   0   0   0   0   3   0 105   0   7]
 [  0   0   2   4   0   2   1   1  92   0]
 [  0   0   0   0   0   1   0   5   1  92]]
```

- 오차행렬 plot에서 가장 많이 오류가 발생한 인덱스(0,6)이 어떤 데이터인지 분석하고 이유를 찾아본다.
- 인덱스 0 = T-Shirt/top
- 인덱스 6 = Shirt

오차 행렬 분석
<Error Analysis>

```
# 왼쪽위 = T-shirt/top 인데 T-shirt/top 이라고 한것
# 오른쪽위 = T-shirt/top 인데 Shirt 라고 한것
# 왼쪽밑 = Shirt 인데 T-shirt/top 이라고 한것
# 오른쪽밑 = Shirt 인데 Shirt 라고 한것
```

```
# Error Analysis 시작
# 에러를 분석하는 과정으로, plot한 결과중 가장많이 틀린(가장밝은) 2개의 그림을 분석한다.

def predict(X,Y,high_fit):
  scaler = StandardScaler()
  high_fit_train_scaler = scaler.fit_transform(X.astype(np.float64))
  y_train_pred = cross_val_predict(high_fit, high_fit_train_scaler, Y, cv=3)
  return y_train_pred

y_pred = predict(X,Y,high_fit)
# print(y_pred)


print(class_names[0]," 와 ", class_names[6], "가 햇갈린다.")

cl_a, cl_b = 0, 6

X_aa = X[(Y == cl_a) & (y_pred == cl_a)]
X_ab = X[(Y == cl_a) & (y_pred == cl_b)]
X_ba = X[(Y == cl_b) & (y_pred == cl_a)]
X_bb = X[(Y == cl_b) & (y_pred == cl_b)]

plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
```



Project Result

프로젝트 결과

- Fashion MNIST 라는 데이터셋에는 6만개의 데이터가 (28*28)의 형태로 10가지 종류의 클래스로 나뉘어 있다.
- 가장 좋은 성능의 분류기는 RandomForest 이다.
- 오차행렬을 통해 가장 많이 틀린 데이터를 확인할 수 있다.
- Plot을 분석한 결과, 라벨 0과 6을 가진 T-Shirt/top 와 Shirt 에서 많은 오류를 찾을 수 있었다.

Thank You!

감사합니다.