와인 감별사: 와인의 Quality를 분류하는 Classifier 만들기

이름: 유서정

목차:

코드 설명

과제 0. 데이터 처리

1. 데이터 로드

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
white_wine = pd.read_csv('./gdrive/My Drive/datamining/winequality-white.csv')
red_wine = pd.read_csv('./gdrive/My Drive/datamining/winequality-red.csv')
```

2. 데이터 분석

데이터 분석 - white wine

```
[49] cnt=np.zeros(11)

     for i in range(len(white_wine)):
       q=int(white_wine.iloc[i].quality)
       cnt[q]+=1;

     print(cnt)
```

```
[   0.    0.    0.   20.  163. 1457. 2198.  880.  175.    5.    0.]
```

데이터 분석 - Red wine

```
[50] cnt=np.zeros(11)

     for i in range(len(red_wine)):
       q=int(red_wine.iloc[i].quality)
       cnt[q]+=1;

     print(cnt)
```

```
[  0.   0.   0.  10.  53. 681. 638. 199.  18.   0.   0.]
```

- 데이터 분석 결과 두 와인 다 quality 3~8에 몰려 있고, quality 0,1,2,11를 가지는 데이터가 없기 때문에 정확하게 트레이닝하기에 적절하지 않은 데이터 셋이라고 생각됨.

3. 데이터를 (x, y)로 나누고, (train, test)로 나누기.

```
from sklearn.model_selection import train_test_split
def generate_data(df, t_r):
    X= pd.DataFrame(df.drop(columns="quality"))
    Y=pd.DataFrame(df, columns=['quality'])
    X_train, X_test, Y_train, Y_test = train_test_split(X,Y, te
st_size=t_r, random_state=1)

    return X_train.values, Y_train, X_test.values, Y_test
```

Label인 quality 값들만 Y로 두고, 나머지는 X로 둔다.

```
x_train_white, y_train_white, x_test_white, y_test_white = gene
rate_data(white_wine, 0.7)

x_train_red, y_train_red, x_test_red, y_test_red = generate_dat
a(red_wine, 0.7)
```

White wine과 red wine을 각각 분류.

과제 1. 화이트 와인 분류 모델과 레드 와인 분류 모델 설계 및 학습

1. 모델 생성

```
n_in=11 # input dimension
n_hiddens=32 # hidden layer 당 32 개의 node 를 가짐
n_out=11 # output dimension (0~10)

model = Sequential()
```

```
model.add(layers.Dense(units=n_hiddens,input_dim=n_in, activ
ation='relu'))
model.add(layers.Dense(units=n_hiddens,activation='relu'))
model.add(layers.Dense(units=n_hiddens,activation='relu'))
model.add(layers.Dense(units=n_out,activation='softmax'))
```

- Keras로 model 구현.
- feature가 11개이므로 input dimension 11개로 설정.
- Hidden layer 당 노드 32개
- 분류해야 할 Quality는 0~10으로 11개이므로 output dimension 11로 설정

2. 모델 컴파일

```
model.compile(loss='sparse_categorical_crossentropy', optimi
zer='adam', metrics=['accuracy'])
```

- Label을 one-hot encoding하지 않고 정수 형태로 넣기 때문에 loss 함수를 sparse_categorical_crossentropy로 선택.

3. 모델 학습

```
epochs = 100
batch_size = 200

# White Wine Classifier
hist1=model.fit(x_train_white, y_train_white, epochs=epochs,
 batch_size=batch_size)

#Red Wine Classifier
hist2=model.fit(x_train_red, y_train_red, epochs=epochs, bat
ch_size=batch_size)
```

- White wine과 red wine 따로 학습시킴
- Epoch와 batch size는 각각 100, 200으로 두 와인 모두 동일

4. 모델 평가 및 결과 출력

- White wine

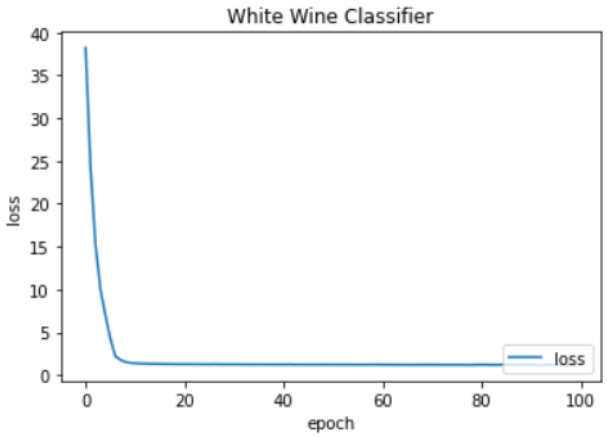
```
test_loss, test_acc = model.evaluate(x_test_white,y_test_whi
te)
```

```
print("White Wine")
print("test loss: ", test_loss)
print("test accuracy: ", test_acc)
```

```
Epoch 96/100
1469/1469 [==============================] - 0s 16us/step - loss: 1.1811 - accuracy: 0.4826
Epoch 97/100
1469/1469 [==============================] - 0s 16us/step - loss: 1.1823 - accuracy: 0.4813
Epoch 98/100
1469/1469 [==============================] - 0s 16us/step - loss: 1.1820 - accuracy: 0.4765
Epoch 99/100
1469/1469 [==============================] - 0s 16us/step - loss: 1.1781 - accuracy: 0.4888
Epoch 100/100
1469/1469 [==============================] - 0s 16us/step - loss: 1.1994 - accuracy: 0.4881
3429/3429 [==============================] - 0s 47us/step
White Wine
test loss:  1.3313665256864582
test accuracy:  0.44327792525291443
```

■ 정확도: 44%~49%

```
plt.title('White Wine Classifier')
plt.plot(hist1.history['loss'], label='loss')
#plt.plot(hist.history['accuracy'], label='acuracy')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(loc='lower right')
plt.show()
```



- Red wine

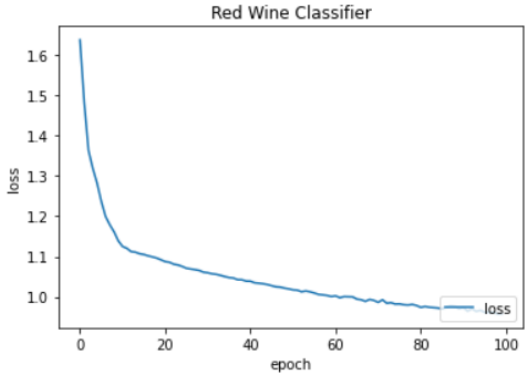
```
test_loss, test_acc = model.evaluate(x_test_red,y_test_red)
print("\nRed Wine")
print("test loss: ", test_loss)
print("test accuracy: ", test_acc)
```

```
Epoch 95/100
479/479 [==============================] - 0s 22us/step - loss: 0.9940 - accuracy: 0.5658
Epoch 96/100
479/479 [==============================] - 0s 22us/step - loss: 0.9939 - accuracy: 0.5678
Epoch 97/100
479/479 [==============================] - 0s 22us/step - loss: 0.9924 - accuracy: 0.5574
Epoch 98/100
479/479 [==============================] - 0s 22us/step - loss: 0.9906 - accuracy: 0.5825
Epoch 99/100
479/479 [==============================] - 0s 20us/step - loss: 0.9889 - accuracy: 0.5846
Epoch 100/100
479/479 [==============================] - 0s 21us/step - loss: 0.9862 - accuracy: 0.5741
1120/1120 [==============================] - 0s 36us/step

Red Wine
test loss:  1.1106757521629333
test accuracy:  0.5303571224212646
```

- 정확도: 49%~55%

```
plt.title('Red Wine Classifier')
plt.plot(hist2.history['loss'], label='loss')
#plt.plot(hist.history['accuracy'], label='acuracy')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(loc='lower right')
plt.show()
```



과제 2. 각 모델의 성능을 향상시킬 수 있는 방법 적용

1. Normalization

```
def normalization(x_train, x_test):
  minval=[]
  maxval=[]
  for i in range(11):
    minval.append(min(x_train[:,i]))
    maxval.append(max(x_train[:,i]))

  for i in range(len(x_train)):
    for j in range(11):
      x_train[i][j]=(x_train[i][j]-minval[j])/(maxval[j]-
minval[j])

  for i in range(len(x_test)):
    for j in range(11):
      x_test[i][j]=(x_test[i][j]-minval[j])/(maxval[j]-
minval[j])

  return x_train, x_test
```

- 각 feature 별로 min 값과 max 값을 구해 [(현재 값- min)/(max-min)] 해 줌으로써 normalization 진행

```
x_train_white, y_train_white, x_test_white, y_test_white
= generate_data(white_wine, 0.7)
x_train_white, x_test_white = normalization(x_train_white
,x_test_white)

x_train_red, y_train_red, x_test_red, y_test_red = genera
te_data(red_wine, 0.7)
x_train_red, x_test_red = normalization(x_train_red,x_tes
t_red)
```

- 다음과 같이, 먼저 초기 데이터를 (x,y), (train, test)로 나눈 후, 각각 normalization 진행

2. One-hot encoding

```
from tensorflow.keras.utils import to_categorical

y_train_white = to_categorical(y_train_white,11);
y_test_white = to_categorical(y_test_white,11);
y_train_red = to_categorical(y_train_red,11);
y_test_red = to_categorical(y_test_red,11);
```

- (label)y값을 one-hot encoding한다.
- categorical_crossentropy를 loss 함수로 사용하기 위함.

3. 모델 생성

```
n_in=11
n_out=11
p_keep=0.5 # 드롭아웃 확률의 비율

model = Sequential()

model.add(layers.Dense(units=512,input_dim=n_in, activa
tion='relu'))
model.add(Dropout(p_keep))
model.add(BatchNormalization())
model.add(layers.Dense(units=256,activation='relu'))
model.add(Dropout(p_keep))
model.add(BatchNormalization())
model.add(layers.Dense(units=128,activation='relu'))
model.add(Dropout(p_keep))
model.add(BatchNormalization())
model.add(layers.Dense(units=n_out,activation='softmax'
))
```

- Internal Covariate Shift 문제를 해결하기 위해 BatchNormalization을 사용.
- BatchNormalization만 사용하는 경우, training 동안 accuracy는 높지만, test 결과 accuracy는 낮은 것으로 보아 overfitting이 되어있음을 알 수 있다. Overfitting을 방지하기 위해 50%의 확률로 노드를 삭제하는 dropout을 같이 사용하여 overfitting을 방지.
- Hidden layer의 개수와 각 layer의 node 수를 전보다 늘렸다.
    - 이 이상 layer를 늘려도 성능의 변화를 보이지 않았기 때문에, 이 값으로 hidden layer를 결정함.
- Activation function의 경우, sigmoid로 사용하면 정확도가 relu를 사용하는 것에 비해 10% 감소함.

4. 모델 컴파일

```
sgd = optimizers.SGD(lr=0.01, decay=1e-
6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimize
r='sgd', metrics=['accuracy'])
```

- Optimizer의 파라미터를 지정.

- Learning late를 0.1~0.001 사이에서 변화하며 결과 관찰하였는데, 정확도의 변화가 거의 없었습니다.

- Loss를 categorical_crossentropy로 변경함으로써 약간의 성능 향상을 도출

5. 모델 학습

```
epochs = 100
batch_size =100

# White Wine Classifier
hist1=model.fit(x_train_white, y_train_white, epochs=ep
ochs, batch_size=batch_size)

#Red Wine Classifier
hist2=model.fit(x_train_red, y_train_red, epochs=epochs
, batch_size=batch_size)
```

- White wine과 red wine 따로 학습시킴

- Epoch와 batch size는 각각 50, 20으로 두 와인 모두 동일

6. 모델 평가 및 결과 출력

- White Wine

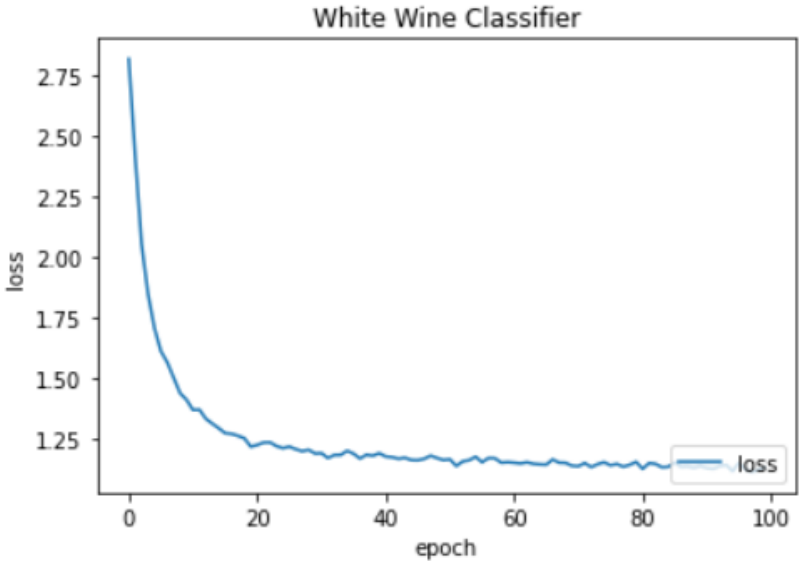
```
test_loss, test_acc = model.evaluate(x_test_white,y_tes
t_white)
print("White Wine")
print("test loss: ", test_loss)
print("test accuracy: ", test_acc)
```

```
Epoch 95/100
1469/1469 [==============================] - 0s 239us/step - loss: 1.1152 - accuracy: 0.5201
Epoch 96/100
1469/1469 [==============================] - 0s 227us/step - loss: 1.1487 - accuracy: 0.4963
Epoch 97/100
1469/1469 [==============================] - 0s 285us/step - loss: 1.1224 - accuracy: 0.5126
Epoch 98/100
1469/1469 [==============================] - 0s 244us/step - loss: 1.1120 - accuracy: 0.5174
Epoch 99/100
1469/1469 [==============================] - 0s 234us/step - loss: 1.1193 - accuracy: 0.5357
Epoch 100/100
1469/1469 [==============================] - 0s 234us/step - loss: 1.1309 - accuracy: 0.5071
3429/3429 [==============================] - 0s 88us/step
White Wine
test loss:  1.083899854275255
test accuracy:  0.5357246994972229
```

- 정확도: 52%~56%

■ BatchNormalization과 dropout을 사용한 결과 white wine의 경우 [과제1]의 결과에 비해 정확도가 3%~12% 정도 향상됨.

```
plt.title('White Wine Classifier')
plt.plot(hist1.history['loss'], label='loss')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(loc='lower right')
plt.show()
```



- Red Wine

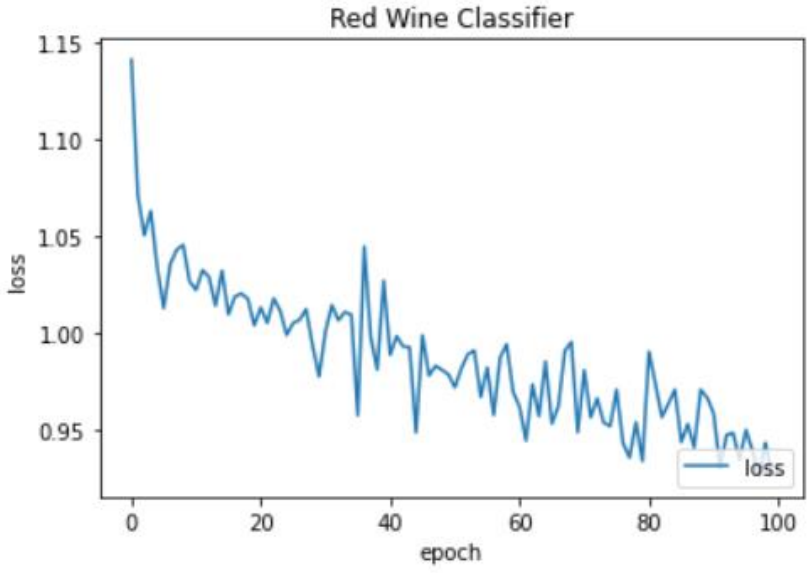
```
test_loss, test_acc = model.evaluate(x_test_red,y_test_
red)
print("\nRed Wine")
print("test loss: ", test_loss)
print("test accuracy: ", test_acc)
```

```
Epoch 95/100
479/479 [==============================] - 0s 288us/step - loss: 0.9346 - accuracy: 0.6117
Epoch 96/100
479/479 [==============================] - 0s 224us/step - loss: 0.9501 - accuracy: 0.5950
Epoch 97/100
479/479 [==============================] - 0s 249us/step - loss: 0.9394 - accuracy: 0.5846
Epoch 98/100
479/479 [==============================] - 0s 217us/step - loss: 0.9264 - accuracy: 0.5846
Epoch 99/100
479/479 [==============================] - 0s 221us/step - loss: 0.9434 - accuracy: 0.5637
Epoch 100/100
479/479 [==============================] - 0s 228us/step - loss: 0.9271 - accuracy: 0.5762
1120/1120 [==============================] - 0s 62us/step

Red Wine
test loss:  0.9693934270313808
test accuracy:  0.5973214507102966
```

- 정확도: 59%~61%
- BatchNormalization과 dropout을 사용한 결과 Red wine의 경우 [과제 1]의 결과에 비해 정확도가 4%~12% 정도 향상됨.

```
plt.title('Red Wine Classifier')
plt.plot(hist2.history['loss'], label='loss')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(loc='lower right')
plt.show()
```



과제 3. 화이트 와인과 레드 와인을 하나의 모델만 사용하여 분류

1. White wine, Red wine 데이터 합치기

```
whole_wine=pd.concat([white_wine,red_wine],ignore_index=True
)
```

2. 모델 생성 & 학습 과정 [과제 2]와 동일

3. 결과 출력

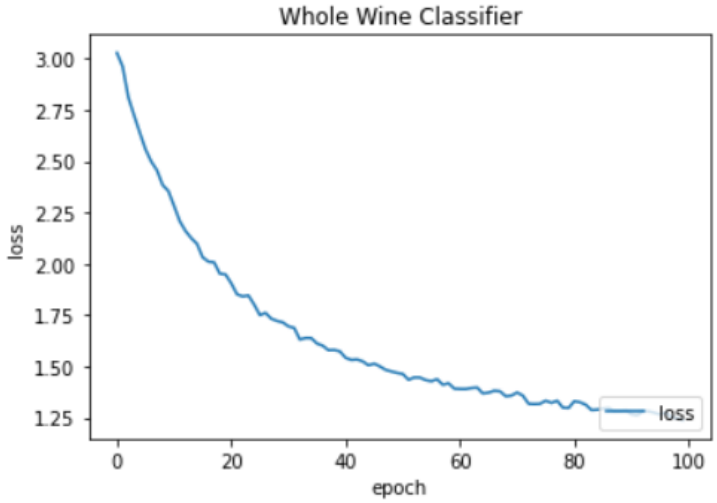
```
hist1=model.fit(x_train, y_train, epochs=epochs, batch_size=
batch_size)
test_loss, test_acc = model.evaluate(x_test,y_test)
print("Whole Wine")
print("test loss: ", test_loss)
print("test accuracy: ", test_acc)
```

```
Epoch 96/100
1949/1949 [==============================] - 0s 30us/step - loss: 1.2103 - accuracy: 0.5033
Epoch 97/100
1949/1949 [==============================] - 0s 33us/step - loss: 1.2124 - accuracy: 0.5115
Epoch 98/100
1949/1949 [==============================] - 0s 30us/step - loss: 1.2134 - accuracy: 0.5244
Epoch 99/100
1949/1949 [==============================] - 0s 30us/step - loss: 1.2382 - accuracy: 0.5085
Epoch 100/100
1949/1949 [==============================] - 0s 30us/step - loss: 1.2186 - accuracy: 0.5213
4548/4548 [==============================] - 0s 88us/step
Whole Wine
test loss:  1.088131062902687
test accuracy:  0.5532101988792419
```

- 정확도: 53%~57%
- [과제3]의 경우 데이터 셋이 그전 과제보다 많기 때문인지, batch 사이즈가 100 이하일 때보다 150이상일 때 정확도가 1~3% 정도 높았다.
  - ■ 하지만, 그 이상은 변화가 없었다.

```
plt.title('Whole Wine Classifier')
plt.plot(hist1.history['loss'], label='loss')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(loc='lower right')
```

```
plt.show()
```



4. 다른 방법

   - [과제 3]의 경우 Random Forest 방식을 이용해서 문제를 해결해 보았습니다.
   - 하지만, 위에서 사용한 방법이 정확도가 더 높게 나왔기 때문에, 위 방법을 선택.
   - 자세한 구현 방법은 하단의 '실험 결과 및 고찰'에 작성함.

실험 결과 및 고찰

1. 모델 생성

   - Layer의 개수와, 각 layer 별 node의 수
     - ■ 실험 전에는 Layer의 개수와 node의 개수가 모델의 성능이 큰 영향을 줄 것으로 예상했지만, 예상보다 큰 변화가 없었다.
       - ◆ 특히, layer의 개수가 더 늘어나면 정확도가 감소했다. 정확도 향상에는 node 수 증가시키는게 더 영향력 있었던 것 같다.
     - ■ 모델과 데이터가 단순한 형태이기 때문이라고 생각.

- Normalization

```
def normalization(x_train, x_test):
  minval=[]
  maxval=[]
  for i in range(11):
    minval.append(min(x_train[:,i]))
    maxval.append(max(x_train[:,i]))

  for i in range(len(x_train)):
    for j in range(11):
      x_train[i][j]=(x_train[i][j]-
minval[j])/(maxval[j]-minval[j])

  for i in range(len(x_test)):
    for j in range(11):
      x_test[i][j]=(x_test[i][j]-
minval[j])/(maxval[j]-minval[j])

  return x_train, x_test
```

■ 다음과 같이 데이터를 normalization할 함수를 구현하여 적용하였다.

■ 정확도가 0.1~0.9 정도로 미세하게 향상된 것 같으나 이 정도 차이는 실험할 때마다 변하는 정도로, 성능을 확실히 향상시켰다고 보기 힘들다.

- Dropout

```
model.add(layers.Dense(units=512,input_dim=n_in, ac
tivation='relu'))
model.add(Dropout(p_keep))
```

■ 다음과 같이 Dropout을 적용할 경우 오히려 정확도가 떨어졌다.

■ 모델이 단순한 형태이기 때문에, 학습을 방해하는 요소로만 작용된 것 같다.

- BatchNormalization

```
model.add(layers.Dense(units=512,input_dim=n_in, ac
tivation='relu'))
model.add(BatchNormalization())
```

■ Internal Covariate Shift 문제를 해결하기 위해 다음과 같이 BatchNormalization을 적용하였다.

■ 하지만 이 경우도 마찬가지로 정확도가 떨어졌다.

■ Echo당 loss와 accuracy를 보면, 학습할 때는 accuracy가 굉장히 높은데, test할 때는 성능이 낮은 것으로 보아 overfitting이 문제인 것으로 판단된다.

- Dropout & BatchNormalization

```
model.add(layers.Dense(units=512,input_dim=n_in, activation='relu'))
model.add(Dropout(p_keep))
model.add(BatchNormalization())
```

■ 각 layer마다 Droput을 해준 다음 BatchNormalization을 해주었다.

■ 둘 다 사용하지 않을 때와 비교해서 정확도가 3~12% 증가 하였다..

2. 모델 컴파일

- Loss function: sparse_categorical_crossentropy

```
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='sparse_categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

■ Loss 함수를 sparse_categorical_crossentropy를 사용하고 label(y)를 정수 형태로 두었고, sgd의 파라미터를 직접 작성하여 학습시켰다.

■ Learning late의 변화에 대한 정확도 차이는 거의 없었다.

- Loss function: categorical_crossentropy

```
from tensorflow.keras.utils import to_categorical

y_train = to_categorical(y_train,11);
y_test = to_categorical(y_test,11);
```

■ (label)y를 one-hot encoding

```
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss=categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

■ Label을 one-hot encoding한 후 categorical_crossentropy loss 함수를 적용하니, [과제 1]과 비교하여 성능이 3%~5% 정도 향상되었다.

3. 모델 학습

- Epoch 수, batch size

■ [과제 1,2]의 경우 두 값의 차이가 결과에 큰 영향을 미치지 않았다. 데이터 셋이 작기 때문인 것 같다.

■ [과제 3]의 경우 batch size가 150이상이 가장 정확도가 높았다.

- Random Forest

```
a=1
def make_model():
  global a
  N = len(white_wine)
  n_in=11
  n_hiddens=32
  n_out=11
  p_keep=0.5 # 드롭아웃 확률의 비율

  model = Sequential()

  model.add(Dense(50*a,input_shape=(n_in,)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  model.add(Dense(50*a))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  model.add(Dense(50*a))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  model.add(Dense(50*a))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  model.add(layers.Dense(units=n_out,activation='so
ftmax'))
  a=a+1
  sgd = optimizers.SGD(lr=0.001, decay=1e-
6, momentum=0.9, nesterov=True)
  model.compile(loss='sparse_categorical_crossentro
py', optimizer=sgd, metrics=['accuracy'])

  return model
```

■ 다음과 같이 모델을 생성할 수 있는 함수를 구현하였다.

■ a라는 전역변수를 두어 model을 생성할 때마다 layer의 노드 수가 달라지도록 하였다.

```
# 서로 다른 모델을 만들어 합치기 (Model Ensemble)
model1 = KerasClassifier(build_fn=make_model,epochs
=epochs, verbose=0)
model1._estimator_type="classifier"
model2 = KerasClassifier(build_fn=make_model,epochs
=epochs, verbose=0)
model2._estimator_type="classifier"
model3 = KerasClassifier(build_fn=make_model,epochs
=epochs, verbose=0)
model3._estimator_type="classifier"

ensemble_model=VotingClassifier(estimators = [('mod
el1', model1), ('model2', model2), ('model3', model
3)], voting = 'soft')
```

- 위에서 언급한 함수를 이용하여 서로 다른 모델 3개를 만들어 random forest를 구현하였다.
- 하지만 결과는 [과제 1]의 결과와 크게 다르지 않았다.
- 정확도: 51%~55%

결론

- Layer 개수, node 개수, epoch 수, batch size, learning late 등의 파라미터 값의 변화는 결과에 큰 영형을 미치지 않았다.
  - [과제 1]에서 위 값들을 변화시켜 얻을 수 있는 최적의 결과는 정확도 60%였다. ([과제 1] – 정확도 50%대)
- Wine을 두 종류로 나눠서 분류할 때와 함께 분류할 때 모델의 성능 변화는 없는 것으로 판단된다.
- Label을 one-hot encoding하고 loss 함수를 categorical_crossentropy로 선택했을 때 정확도가 3%~5% 정도 향상되었다.
- 데이터가 모든 quality를 가지고 있지 않고, 일부 quality 값에만 집중되어 있기 때문에, 모델과 상관없이 정확한 예측이 불가능한 것으로 판단된다.