자살 고위험군 예측

1. 데이터 전처리

1) 결측값 처리

- 범주형 변수(최빈값으로 처리)와 수치형 변수(중앙값, 평균값으로 처리)가 섞여있기 때문에 범주형 변수와 수치형 변수를 나누어 따로 처리한 다음 합친다.
- 결측 비율이 10%가 넘는 LW_mt, LW_oc 변수를 버린다. -> 남은 변수 86개
- labels(타겟)이 되는 mh_suicide의 결측은 행(데이터)를 지운다. -> 남은 데이터 5935개

2) 전처리

- 어떤 변수가 수치형 변수이고 범주형 변수인지 나누기 위해 각 nums, cats 리스트를 만들어주어야 한다.
- 결측치를 처리한 수치형 변수에는 StandardScaler(Z-score) 정규화를 적용하여 결측치 처리한 범주형 변수와 합친다.

3) 데이터 나누기

위의 과정을 통해 정리된 데이터를 train data: test data = 7:3으로 나눈다.

```
print('Train data shape: {0}'.format(X_train.shape))
print('Test data shape: {0}'.format(X_test.shape))

print('Train data label => %s' %Counter(y_train))
print('Test data label => %s' %Counter(y_test))
```

```
Train data shape: (4154, 84)
Test data shape: (1781, 84)
Train data label => Counter({0.0: 3927, 1.0: 227})
Test data label => Counter({0.0: 1684, 1.0: 97})
```

→ 나누어진 것을 확인해보니 label이 0인 값이 1인 값보다 훨씬 많다. 데이터가 불균형하다.

• 데이터 불균형 해결

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=0)
X_train_over,y_train_over = smote.fit_sample(X_train,y_train)
```

```
print('SMOTE 적용 전 학습용 피처/레이블 데이터 세트: ', X_train.shape, y_train.shape)
print('SMOTE 적용 후 학습용 피처/레이블 데이터 세트: ', X_train_over.shape, y_train_over.shape)
print('SMOTE 적용 후 레이블 값 분포: ₩n', pd.Series(y_train_over).value_counts())
```

```
SMOTE 적용 전 학습용 피처/레이블 데이터 세트:  (4154, 84) (4154,)
SMOTE 적용 후 학습용 피처/레이블 데이터 세트:  (7854, 84) (7854,)
SMOTE 적용 후 레이블 값 분포:
 1.0    3927
0.0    3927
```

→ SMOTE 함수를 이용해 기존의 label이 1인 데이터로 임의의 새로운 데이터를 만들어주어 label이 0인 데이터 개수에 맞추었다.

2. DNN 모델 구조 선정

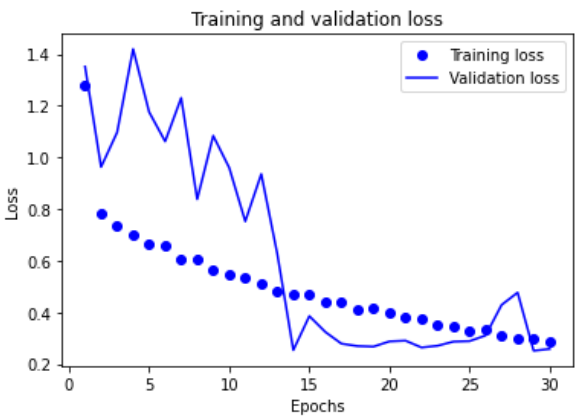
• Original model

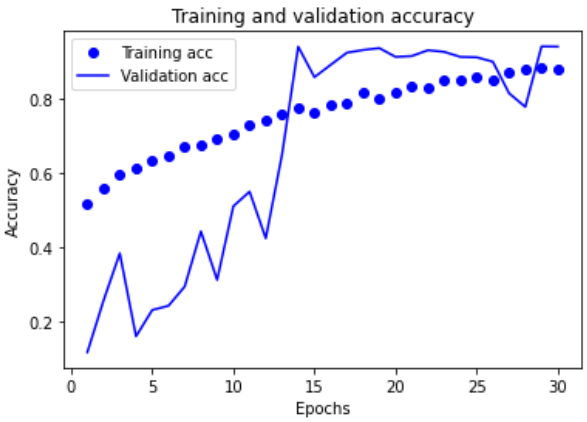
```
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.models import load_model

model = models.Sequential()
model.add(Dense(128, input_dim = 84, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='RMSprop',loss='binary_crossentropy',metrics=['accuracy'])
```

```
history = model.fit(X_train_over,y_train_over,epochs=30,batch_size=512,validation_data=(X_test, y_test))
```

→ 일단 노드의 수를 정하기 위해 은닉층은 하나만 사용하였다. 노드의 수는 128로 지정하였다. 입력층과 은닉층의 활성함수는 'relu', 출력층은 'sigmoid'를 사용하였다. optimizer로는 'RMSprop'로 설정하였고 이진분류이므로 'binary_crossentropy'를 사용하였다.
데이터의 사이즈가 크므로 Epochs=30, batch_size=512를 사용하여 512개의 데이터를 임의로 뽑아 30번 학습시켰다. 검증데이터로는 test data를 사용하였다.

```
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```





→ training data와 validation data(test data를 사용함)의 loss값과 accuracy값을 그래프로 그려보았다. 급격하게 loss가 떨어지고 accuracy가 증가하는 모습을 볼 수 있다. 끝에는 loss값이 오르고 accuracy값은 떨어지는 모양이다. 대체로 불안정해보인다.

• Smaller model

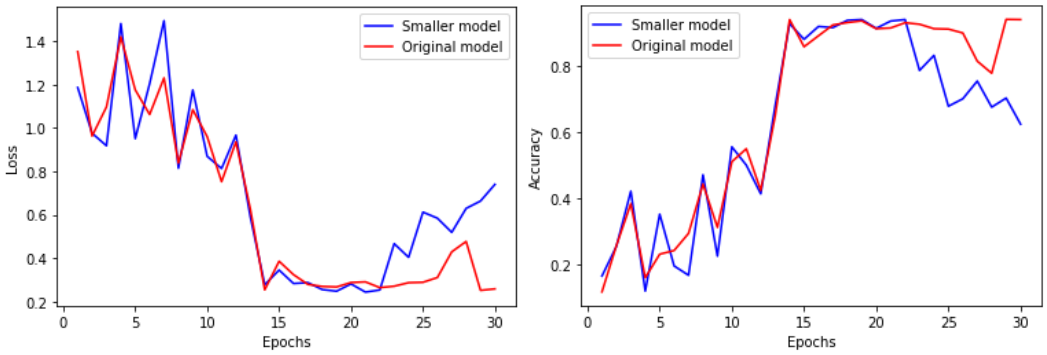
```
# 노드수 줄인 모델
smaller = models.Sequential()
smaller.add(Dense(100, input_dim = 84, activation='relu'))
smaller.add(Dense(100, activation='relu'))
smaller.add(Dense(1, activation='sigmoid'))

smaller.compile(optimizer='rmsprop',
                loss='binary_crossentropy',
                metrics=['accuracy'])
```

```
smaller_hist = smaller.fit(X_train_over,y_train_over,epochs=30,batch_size=512,validation_data=(X_test, y_test))
```

→ original model과 똑같지만 노드의 수를 100으로 줄인 모델이다.
100보다 작은 노드의 수의 모델을 만들어 비교해보았지만 전부 성능이 너무 좋지 않았다.

→ 그래프를 확인해보면 original model과 마찬가지로 불안정하고 급격한 증감이 있다.

```
hist_dict = history.history
smaller_dict = smaller_hist.history

val_acc = hist_dict['val_accuracy']
val_loss = hist_dict['val_loss']
smaller_val_acc = smaller_dict['val_accuracy']
smaller_val_loss = smaller_dict['val_loss']

epochs = range(1, len(val_acc)+1)

#  'b' 는 파란색 실선, 'r'은 빨간색 실선을 의미합니다
plt.plot(epochs, smaller_val_acc, 'b', label='Smaller model')
plt.plot(epochs, val_acc, 'r', label='Original model')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

#  'b' 는 파란색 실선, 'r'은 빨간색 실선을 의미합니다
plt.plot(epochs, smaller_val_loss, 'b', label='Smaller model')
plt.plot(epochs, val_loss, 'r', label='Original model')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

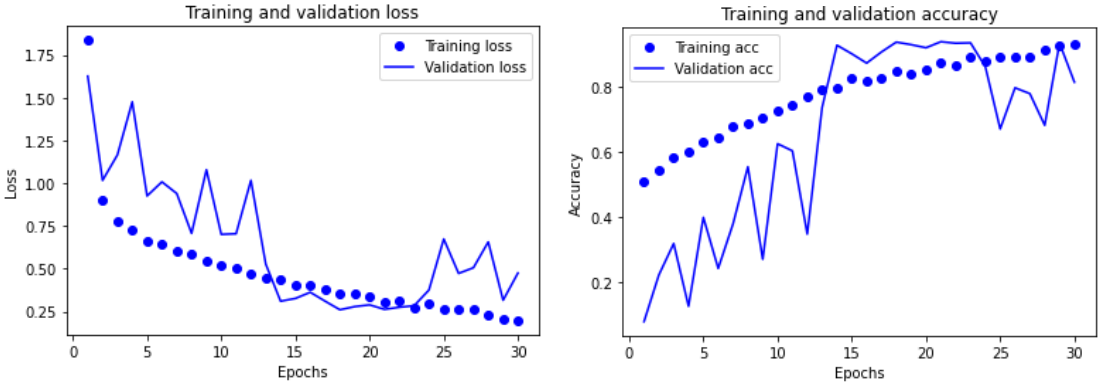


→ original model과 smaller model의 val_loss와 val_accuracy를 비교하는 그래프를 그려보았다. Original model이 조금 더 안정적이고 끝에 loss값이 올라가고 acc가 내려가는 과적합의 정도도 더 나은 것으로 보인다.=> original model (노드 128개)

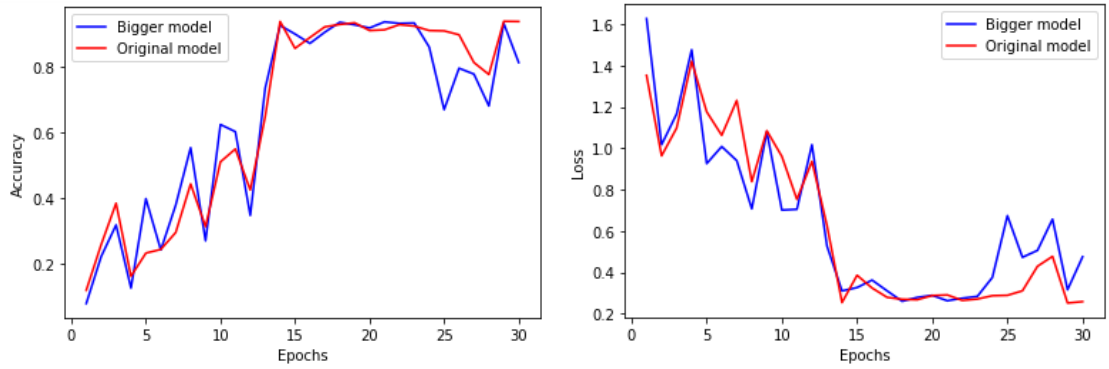
• Bigger model

```
# Bigger

bigger = models.Sequential()
bigger.add(Dense(250, input_dim = 84, activation='relu'))
bigger.add(Dense(250, activation='relu'))
bigger.add(Dense(1, activation='sigmoid'))

bigger.compile(optimizer='RMSprop',loss='binary_crossentropy',metrics=['accuracy'])
```

→ 위의 모델들과 같은 조건으로 노드의 수를 250으로 크게 설정하였다. 그 이상의 값은 심하게 과적합이 발생하여 250으로 지정하였다.



→ training data와 validation data의 loss와 accuracy 그래프이다. 급격한 증감과 끝의 과적합은 위의 두 모델과 비슷한 특징으로 보인다.



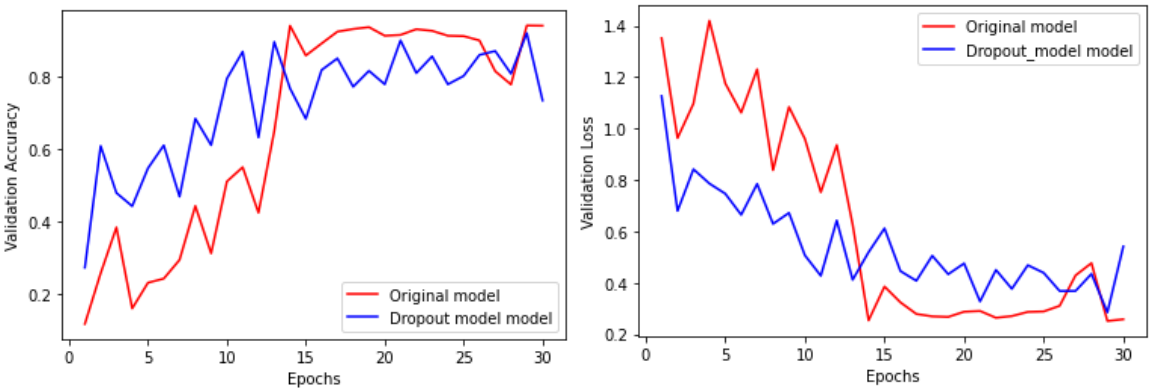
→ original model과 bigger modeld의 loss와 accuracy를 비교했을 때 original model이 더 안정적이고 끝에 과적합도 덜 심한 것을 확인할 수 있다. => Original model

3. 규제 적용

• Dropout layer 추가

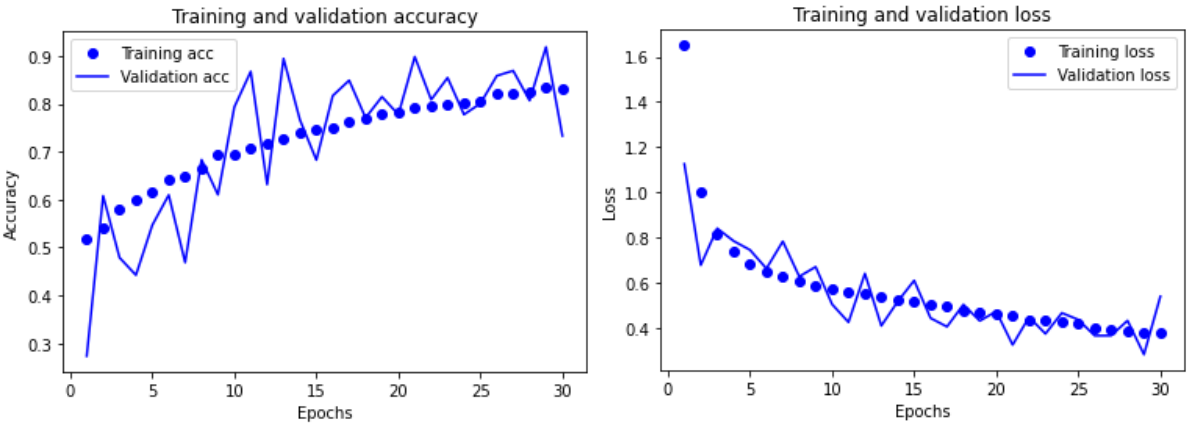
```
dpt_model = models.Sequential()
dpt_model.add(Dense(128, activation='relu', input_dim=84))
dpt_model.add(Dropout(0.5))
dpt_model.add(Dense(128, activation='relu'))
dpt_model.add(Dropout(0.2))
dpt_model.add(Dense(1, activation='sigmoid'))

dpt_model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
```

→ 선정한 DNN 모델(노드 128개)에 dropout layer를 추가하였다. 입력층과 은닉층 사이는 0.5로, 은닉층과 출력화 사이는 0.2로 dropout해주었다.



→ dropout layer를 추가하였더니 급격한 모양의 original model이 보다 안정적으로 변하였다.
=> Dropout model이 더 나은 모델.

+) dropout model와 original model에 층을 더 쌓아보았지만 둘 다 오히려 불안정해졌다.

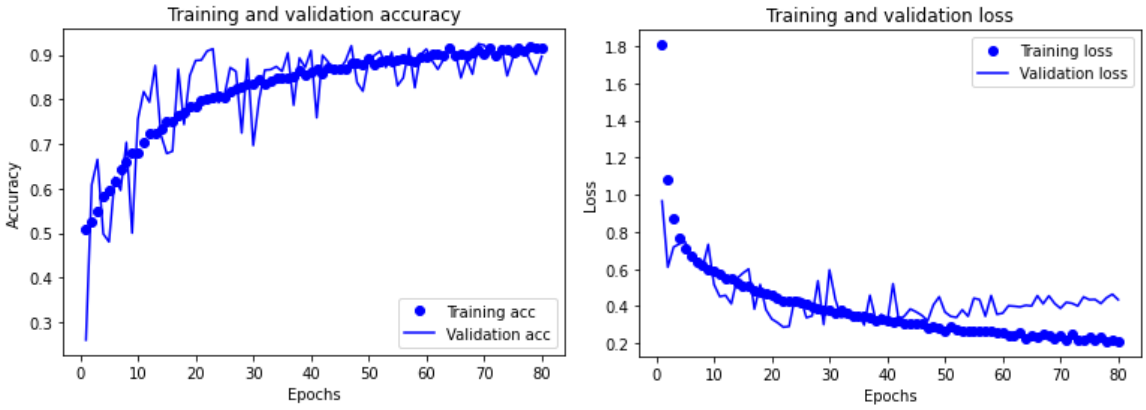


→ training data와 validation data의 accuracy와 loss값 그래프를 그려보았다. Validation data의 그래프가 대체로 불안정하지만 epochs가 커질수록 accuracy가 증가하고 loss가 감소하는 것을 확인할 수 있다.

```
more_model = models.Sequential()
more_model.add(Dense(128, activation='relu', input_dim=84))
more_model.add(Dropout(0.5))
more_model.add(Dense(128, activation='relu'))
more_model.add(Dropout(0.2))
more_model.add(Dense(1, activation='sigmoid'))

more_model.compile(optimizer='rmsprop',
                   loss='binary_crossentropy',
                   metrics=['accuracy'])
```

```
more_hist = more_model.fit(X_train_over,y_train_over,epochs=80,batch_size=512,validation_data=(X_test, y_test))
```

→ L1, L2 규제를 해주기 전에 모델의 용량을 더 키우기 위해 epochs를 80으로 올려 실행하였다.

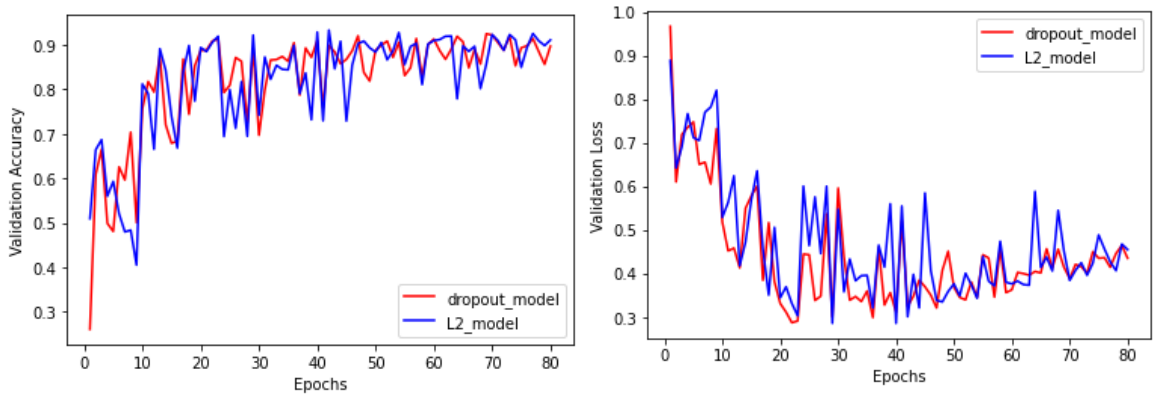


→ training data와 validation accuracy의 accuracy와 loss를 그려보았습니다. Epochs 40이후로 점점 loss값이 증가하는 것을 확인할 수 있다. Loss값이 계속 감소하는 모델이 좋은 모델이다.

• L2 규제

```
l2_model = models.Sequential()
l2_model.add(Dense(128, input_dim = 84, activation='relu', kernel_regularizer=regularizers.l2(0.0001)))
l2_model.add(Dropout(0.5))
l2_model.add(Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.0001)))
l2_model.add(Dropout(0.2))
l2_model.add(Dense(1, activation='sigmoid'))

l2_model.compile(optimizer=RMSprop(),loss='binary_crossentropy',metrics=['accuracy'])
```

→ 위의 모델에 L2 규제를 적용하였다. (0.0001)

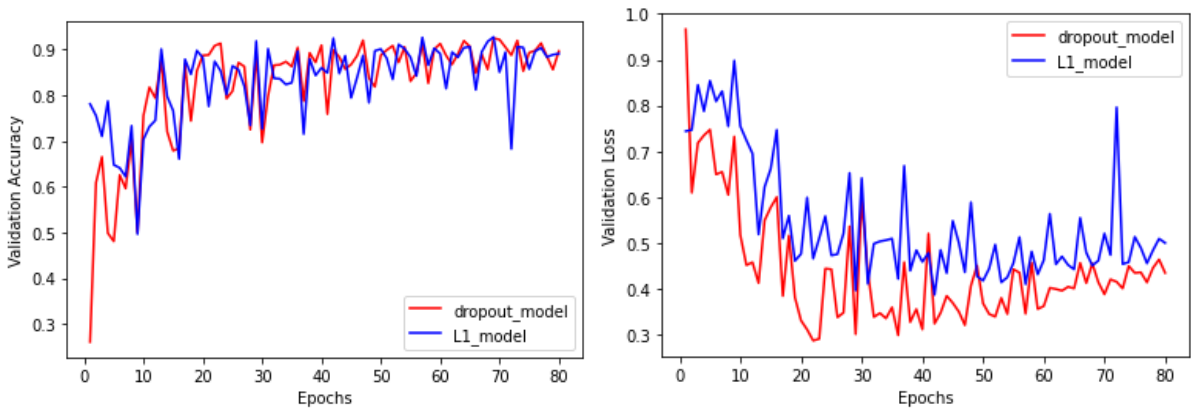


→ 기존의 모델보다 L2 규제를 적용한 모델이 더 불안정한 것으로 보인다.

• L1 규제

```
l1_model = models.Sequential()
l1_model.add(Dense(128, input_dim = 84, activation='relu', kernel_regularizer=regularizers.l1(0.0001)))
l1_model.add(Dropout(0.5))
l1_model.add(Dense(128, activation='relu', kernel_regularizer=regularizers.l1(0.0001)))
l1_model.add(Dropout(0.2))
l1_model.add(Dense(1, activation='sigmoid'))

l1_model.compile(optimizer=RMSprop(),loss='binary_crossentropy',metrics=['accuracy'])
```

```
l1_hist = l1_model.fit(X_train_over, y_train_over,
                       epochs=80,
                       batch_size=512,
                       validation_data=(X_test, y_test))
```

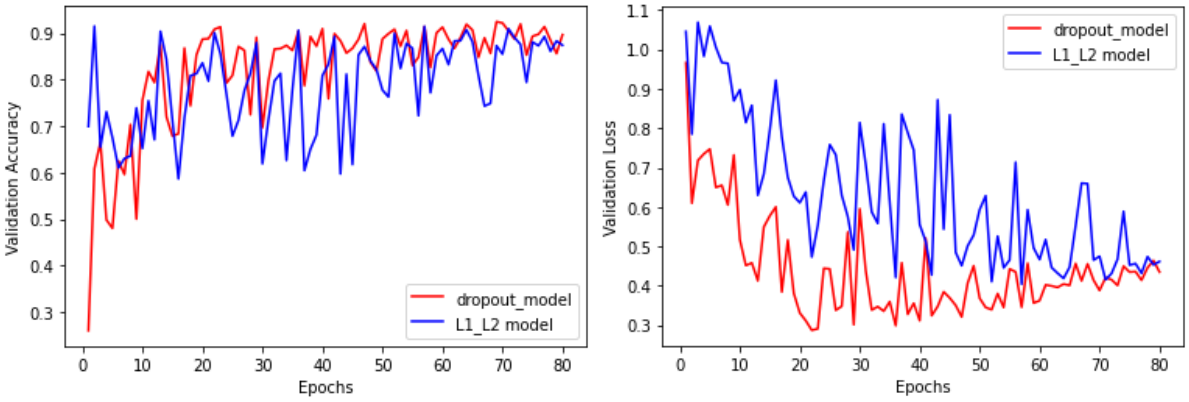


→ 기존의 모델보다 L1 규제를 적용한 모델이 더 불안정해 보이고 val_loss은 더 높다. 기존의 모델이 더 적절한 모델이다.

• L1_L2 규제

```
l1_l2_model = models.Sequential()
l1_l2_model.add(Dense(128, kernel_regularizer=regularizers.l1_l2(l1=0.0001, l2=0.001),
                      activation='relu', input_dim=84))
l1_l2_model.add(Dropout(0.5))
l1_l2_model.add(Dense(128, kernel_regularizer=regularizers.l1_l2(l1=0.0001, l2=0.001),
                      activation='relu'))
l1_l2_model.add(Dropout(0.2))
l1_l2_model.add(Dense(1, activation='sigmoid'))

l1_l2_model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
l1_l2_hist = l1_l2_model.fit(X_train_over, y_train_over,
                             epochs=80,
                             batch_size=512,
                             validation_data=(X_test, y_test))
```



→ L1_L2 모델의 val_loss값이 기존 모델보다 전체적으로 높고 val_accuracy값은 더 낮으며 불안정한 것으로 보인다.

=> 규제를 적용해본 결과 기존의 모델이 가장 적절하였다.

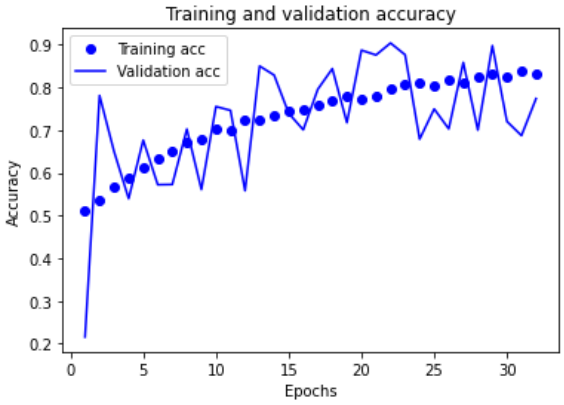
• early Stopping

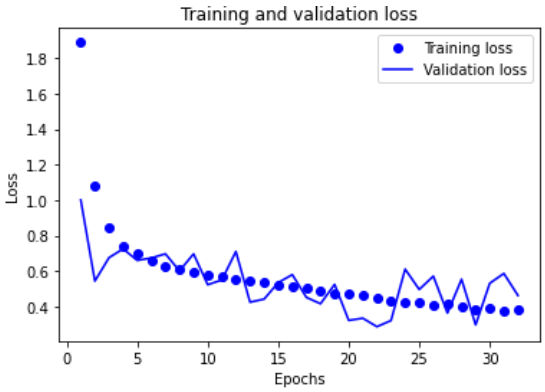
```
final_model = models.Sequential()
final_model.add(Dense(128, activation='relu', input_dim=84))
final_model.add(Dropout(0.5))
final_model.add(Dense(128, activation='relu'))
final_model.add(Dropout(0.2))
final_model.add(Dense(1, activation='sigmoid'))

final_model.compile(optimizer='rmsprop',
                    loss='binary_crossentropy',
                    metrics=['accuracy'])
```

```
import os
from tensorflow.keras.callbacks import ModelCheckpoint,EarlyStopping

MODEL_DIR = './model/'
if not os.path.exists(MODEL_DIR):
    os.mkdir(MODEL_DIR)

modelpath="./model/{epoch:02d}-{val_loss:.4f}.hdf5"
mc = ModelCheckpoint(filepath=modelpath, monitor='val_loss', mode='min', verbose=1, save_best_only=True)

es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=10)
```

```
final_model_hist = final_model.fit(X_train_over,y_train_over,
                                   epochs=80,batch_size=512,validation_data=(X_test, y_test),
                                   callbacks=[es,mc])
```

→ 최종적으로 결정된 모델은 하나의 은닉층과 노드 128개, dropout layer를 추가한 모델이다. 이 모델의 val_loss값이 더 이상 감소하지 않으면 학습을 멈추어 최선의 실행 결과를 얻기 위해 earlyStopping을 사용하였다. 모델이 진동하는 모양으로 불안정하기 때문에 patience를 넉넉히 설정하였다.





→ epochs를 80으로 두고 early stopping을 적용해 실행한 결과, epochs=32 지점에서 멈추었다.

Early stopping을 적용하기 전의 모델인 more_model 그래프는 끝으로 갈수록 loss값이 확실하게 증가하였지만 early stopping을 적용한 final_model은 계속 증가하기 전에 학습이 멈추었다.

4. 성능 평가

• confusion matrix

```
# confusion matrix
from sklearn.metrics import confusion_matrix,classification_report

predict = final_model.predict_classes(X_test)

print(classification_report(y_test,predict))
```

```
              precision    recall  f1-score   support

         0.0       0.97      0.76      0.85      1684
         1.0       0.12      0.59      0.20        97

    accuracy                           0.75      1781
   macro avg       0.55      0.67      0.53      1781
weighted avg       0.92      0.75      0.81      1781
```

→ 다양한 모델들을 실행해보며 accuracy가 0.92이지만 recall은 0.31인 결과 등이 나왔지만 이 주제에서는 자살할 가능성이 낮은 사람에 대한 예측보다 자살 고위험군의 사람을 찾아내는 것이 중요하기 때문에 recall값이 높아야 했다. Final_model의 confusion matrix 결과 recall값은 0.59가 나왔다. Final_model의 임계치를 낮추고 각 class에 가중치를 1:9로 주어 실행하였을 때에는 recall값이 0.8까지 나왔지만 다른 평가지표의 값이 형편없이 나왔다. 따라서 final_model을 최종 모델로 결정하였다.

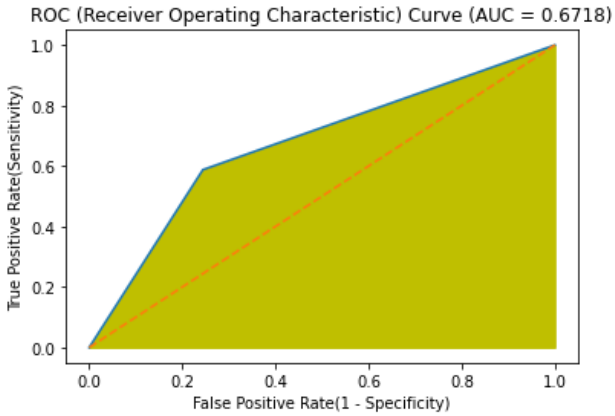
```
# confusion matrix
from sklearn.metrics import confusion_matrix,classification_report

predict = final_model.predict_classes(X_test)

print(classification_report(y_test,predict))
```

```
              precision    recall  f1-score   support

         0.0       0.96      0.96      0.96      1684
         1.0       0.30      0.31      0.31        97

    accuracy                           0.92      1781
   macro avg       0.63      0.63      0.63      1781
weighted avg       0.92      0.92      0.92      1781
```

• **ROC**

```
# ROC 그리기
fpr, tpr, thresholds = metrics.roc_curve(y_test, predict) #곡선 그리기

plt.plot(fpr, tpr)
plt.fill_between(fpr, tpr, color='y')
plt.plot([0, 1], [0, 1], '--')
plt.xlabel('False Positive Rate(1 - Specificity)')
plt.ylabel('True Positive Rate(Sensitivity)')

score = metrics.auc(fpr, tpr) #면적 구하기
print(score) #1.0

plt.title('ROC (Receiver Operating Characteristic) Curve (AUC = {})'.format(round(score,4)))
plt.show()
```

```
0.6717835541298334
```



→ ROC 그래프가 (0,1) 점에 가까워 정사각형 모양에 가까울수록 좋은 성능이다. 즉, AUC가 1에 가까울수록 성능이 좋은 것인데 최종 모델의 AUC는 0.6718정도로 나왔다.

=> 최종 모델의 AUC : 0.6718