Machine Learning Final Project

> 作者: 吳孟維 0816120
> Github Link: https://github.com/wumengwei0213/CS_CS20024_final_project
> Reference:
> https://www.kaggle.com/competitions/tabular-playground-series-aug-2022/discussion/349385
> https://www.kaggle.com/code/johnybhiduri/tabular-playground-series-keras

Brief Intro.

```
After researching some discussion on the Kaggle, I have breif concept of this competition. I decide to design
the NN model by myself in order to treat this project as a self-traing and strengthen my skill in machine
learning region. With looking into the data, I found that feature "loading" has a great relationship with the
goal. And the features that is categorical are not much assosiate with the goal, so I decided to drop these
feature to focus on the numerial features. The crux that I can pass the baseline is adding "model assembling" at
the end of training model. I chose models that got top-5 score in the Kaggle private leaderboard and re-made the
prediction.
```

Model Architecture

```
Model: "model_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_13 (InputLayer)       [(None, 21)]              0

 normalization_12 (Normaliza  (None, 21)               43
 tion)

 dense_60 (Dense)            (None, 256)               5632

 dropout_36 (Dropout)        (None, 256)               0

 batch_normalization_36 (Bat  (None, 256)              1024
 chNormalization)

 dense_61 (Dense)            (None, 256)               65792

 dropout_37 (Dropout)        (None, 256)               0

 batch_normalization_37 (Bat  (None, 256)              1024
 chNormalization)

 dense_62 (Dense)            (None, 256)               65792

 dropout_38 (Dropout)        (None, 256)               0

 batch_normalization_38 (Bat  (None, 256)              1024
 chNormalization)

 dense_63 (Dense)            (None, 64)                16448

 dense_64 (Dense)            (None, 1)                 65

=================================================================
Total params: 156,844
Trainable params: 155,265
Non-trainable params: 1,579
_________________________________________________________________
```

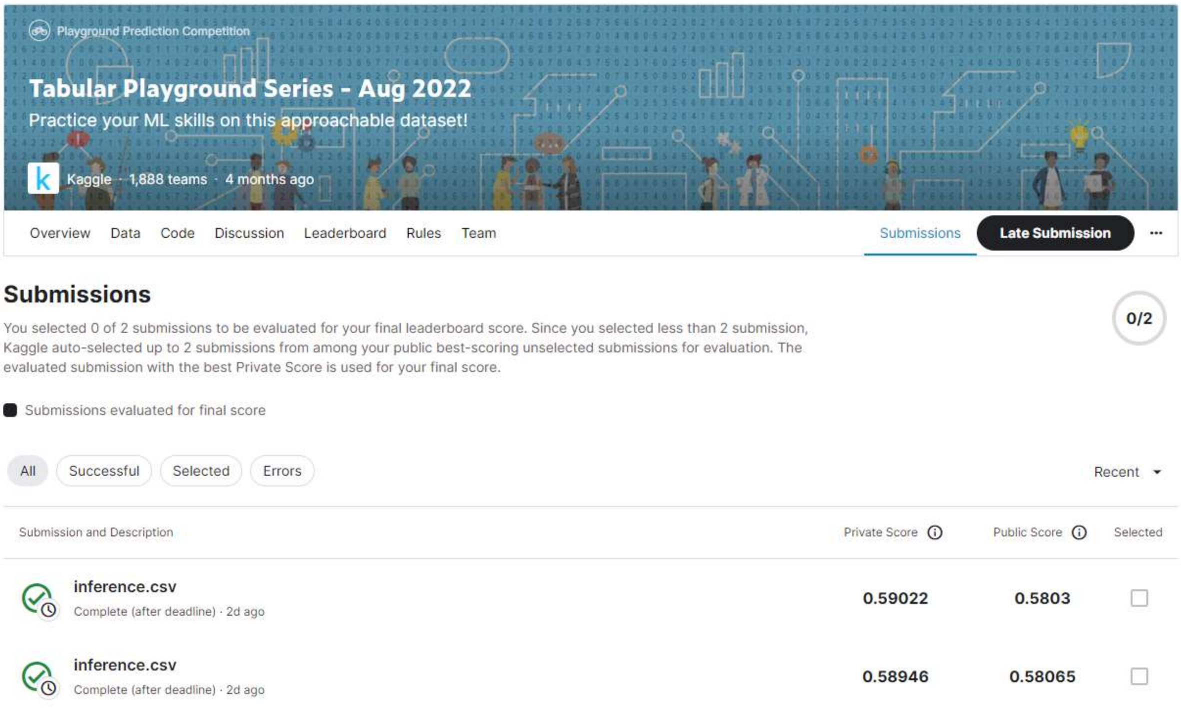
Hyperparameters

> optimizer = Adam with learning_rate=0.001
> loss = 'binary_crossentropy'
> validation_split = 0.3
> batch_size = 128
> epochs = 10

Summary

I got 0.59022 in the end.

Name	Private Score	Public Score
inference.csv	0.59022	0.5803



Methodology

Import Module

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import RobustScaler, StandardScaler
from sklearn.model_selection import KFold
# from imblearn.over_sampling import RandomOverSampler
import tensorflow as tf
from sklearn.metrics import confusion_matrix, classification_report
import warnings
warnings.filterwarnings('ignore')
```

Data Loading

According to the research, adding columns that corresponds to if there is NA value in the features "measurement 3-6" can improve AUC value.

In [2]:
```
train = pd.read_csv('train.csv')
pd.set_option('display.max_columns',None)

train['miss3'] = train['measurement_3'].isna().astype(int)*32
train['miss4'] = train['measurement_4'].isna().astype(int)*32
train['miss5'] = train['measurement_5'].isna().astype(int)*32
train['miss6'] = train['measurement_6'].isna().astype(int)*32
train.head()
```

Out[2]:

	id	product_code	loading	attribute_0	attribute_1	attribute_2	attribute_3	measurement_0	measurement_1	measurement_2	measurement_3	measurem
0	0	A	80.10	material_7	material_8	9	5	7	8	4	18.040	
1	1	A	84.89	material_7	material_8	9	5	14	3	3	18.213	
2	2	A	82.43	material_7	material_8	9	5	12	1	5	18.057	
3	3	A	101.07	material_7	material_8	9	5	13	2	6	17.295	
4	4	A	188.06	material_7	material_8	9	5	9	2	8	19.346	

Data Prepoccessing

- Pick the beneficial features for training
- Replace NA with mean value of the feature

In [3]:
```
checkColumns = ['loading','measurement_0','measurement_1','measurement_2','measurement_3','measurement_4','measurement_5','measureme

train_df = train.copy()
for i in checkColumns :
    m = train_df[i].mean()
    train_df[i] = train_df[i].fillna(m)
```

```
Y = train_df['failure']
X = train_df[checkColumns]
```

Deprecated Method

> Add RandomOverSampler to make the data more balance

Result

> It did not enhance the score, so I remove it.

```
x_train,x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.25, shuffle = True, random_state = 144)
num_samples = int(y_train.value_counts().mean())
majority_ind = y_train[y_train == 0.0].index
samples_to_drop = y_train[majority_ind].sample(num_samples, random_state = 1).index
x_train = x_train.drop(samples_to_drop, axis = 0)
y_train = y_train.drop(samples_to_drop, axis = 0)
over_sampler = RandomOverSampler(random_state = 1)
x_train,y_train = over_sampler.fit_resample(x_train, y_train)
scaler = StandardScaler()
scaler.fit(x_train)
x_train = pd.DataFrame(scaler.transform(x_train), columns = x_train.columns, index = x_train.index)
x_test = pd.DataFrame(scaler.transform(x_test), columns = x_test.columns, index = x_test.index)
```

Train Test Split

In [4]:
```
x_train,x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.25, shuffle = True, random_state = 144)
x_train
```

Out[4]:

	loading	measurement_0	measurement_1	measurement_2	measurement_3	measurement_4	measurement_5	measurement_6	measurement_7	me
22216	109.270000	1	18	1	19.247	11.069000	17.675	17.781	10.855000	
18029	160.020000	3	7	13	18.546	10.638000	16.686	17.536	12.078000	
17046	162.310000	8	13	12	18.851	12.204000	16.031	16.599	12.715000	
3397	118.510000	11	2	2	15.033	11.413000	16.297	17.797	10.999000	
16532	147.620000	4	10	9	18.490	11.731988	20.569	17.462	13.069000	
...	...	...	...	...	...	...	...	...	...	
17830	86.650000	5	11	10	16.651	13.058000	18.441	18.240	11.997000	
7384	109.450000	7	13	5	18.227	11.841000	17.107	18.150	11.910000	
22394	127.826233	6	11	5	18.029	9.568000	17.183	16.258	12.597000	
1468	88.290000	9	5	3	18.092	12.160000	18.499	16.669	10.590000	
25959	123.340000	9	7	5	18.669	10.919000	17.101	17.604	11.716624	

19927 rows × 21 columns

Prediction Data Generating and Model Saving

In [5]:
```
def outp(auc) :
    test = pd.read_csv('test.csv')
    pd_id = test['id']

    test['miss3'] = test['measurement_3'].isna().astype(int)*32
    test['miss4'] = test['measurement_4'].isna().astype(int)*32
    test['miss5'] = test['measurement_5'].isna().astype(int)*32
    test['miss6'] = test['measurement_6'].isna().astype(int)*32

    for i in checkColumns :
        m = test[i].mean()
        test[i] = test[i].fillna(m)
    test = test[checkColumns]

    predictions = model.predict(test)
    predictions = pd.DataFrame(predictions, columns=['failure'])
    out = pd.concat([pd_id, predictions],axis=1)
    out.to_csv(f'save/out{auc}.csv',index=False)
    model.save(f'save/model_{auc}.h5')
```

Building Model and Training

In [6]:
```
while 1 :

    inputs = tf.keras.Input(shape = (x_train.shape[1],))
    x = tf.keras.layers.Normalization(axis=-1)(inputs)
    x = tf.keras.layers.Dense(256, activation = 'relu')(x)
    x = tf.keras.layers.Dropout(0.1)(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Dense(256, activation = 'relu')(x)
    x = tf.keras.layers.Dropout(0.2)(x)
```

```python
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Dense(256, activation = 'relu')(x)
    x = tf.keras.layers.Dropout(0.2)(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Dense(64, activation = 'relu')(x)
    outputs = tf.keras.layers.Dense(1, activation = 'sigmoid')(x)

    model = tf.keras.Model(inputs = inputs, outputs = outputs)
    model.compile(
        optimizer = tf.keras.optimizers.Adam(learning_rate=0.001),
        loss = 'binary_crossentropy',
        metrics = [
            'accuracy',
            tf.keras.metrics.AUC(name = 'auc')
        ]
    )
    history = model.fit(
        x_train,
        y_train,
        validation_split = 0.3,
        batch_size = 128,
        epochs = 10,
    )


    print(history.history['val_auc'])

    if(history.history['val_auc'][-1] > 0.58) :
        outp(str(round(history.history['val_auc'][-1],4)))
        break
```

```
Epoch 1/10
109/109 [==============================] - 8s 11ms/step - loss: 0.5679 - accuracy: 0.7529 - auc: 0.5240 - val_loss: 0.5555 - val_ac
curacy: 0.7869 - val_auc: 0.5746
Epoch 2/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5383 - accuracy: 0.7772 - auc: 0.5376 - val_loss: 0.5141 - val_acc
uracy: 0.7919 - val_auc: 0.5639
Epoch 3/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5282 - accuracy: 0.7828 - auc: 0.5483 - val_loss: 0.5145 - val_acc
uracy: 0.7919 - val_auc: 0.5683
Epoch 4/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5276 - accuracy: 0.7841 - auc: 0.5490 - val_loss: 0.5063 - val_acc
uracy: 0.7918 - val_auc: 0.5727
Epoch 5/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5244 - accuracy: 0.7833 - auc: 0.5558 - val_loss: 0.5064 - val_acc
uracy: 0.7918 - val_auc: 0.5813
Epoch 6/10
109/109 [==============================] - 1s 8ms/step - loss: 0.5235 - accuracy: 0.7843 - auc: 0.5522 - val_loss: 0.5064 - val_acc
uracy: 0.7911 - val_auc: 0.5712
Epoch 7/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5204 - accuracy: 0.7842 - auc: 0.5590 - val_loss: 0.5053 - val_acc
uracy: 0.7903 - val_auc: 0.5713
Epoch 8/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5201 - accuracy: 0.7843 - auc: 0.5637 - val_loss: 0.5051 - val_acc
uracy: 0.7916 - val_auc: 0.5789
Epoch 9/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5203 - accuracy: 0.7848 - auc: 0.5556 - val_loss: 0.5045 - val_acc
uracy: 0.7919 - val_auc: 0.5778
Epoch 10/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5177 - accuracy: 0.7856 - auc: 0.5662 - val_loss: 0.5076 - val_acc
uracy: 0.7916 - val_auc: 0.5774
[0.5746009349822998, 0.5639437437057495, 0.5682618021965027, 0.572697103023529, 0.581304132938385, 0.5712159872055054, 0.5713181495
666504, 0.5789045095443726, 0.5777824521064758, 0.57735478878O2124]
Epoch 1/10
109/109 [==============================] - 2s 8ms/step - loss: 0.5611 - accuracy: 0.7604 - auc: 0.5366 - val_loss: 0.5690 - val_acc
uracy: 0.7918 - val_auc: 0.4577
Epoch 2/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5403 - accuracy: 0.7793 - auc: 0.5245 - val_loss: 0.5094 - val_acc
uracy: 0.7918 - val_auc: 0.5814
Epoch 3/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5278 - accuracy: 0.7823 - auc: 0.5501 - val_loss: 0.5049 - val_acc
uracy: 0.7916 - val_auc: 0.5759
Epoch 4/10
109/109 [==============================] - 1s 6ms/step - loss: 0.5260 - accuracy: 0.7832 - auc: 0.5535 - val_loss: 0.5065 - val_acc
uracy: 0.7918 - val_auc: 0.5638
Epoch 5/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5241 - accuracy: 0.7829 - auc: 0.5514 - val_loss: 0.5068 - val_acc
uracy: 0.7894 - val_auc: 0.5804
Epoch 6/10
109/109 [==============================] - 1s 6ms/step - loss: 0.5237 - accuracy: 0.7839 - auc: 0.5519 - val_loss: 0.5052 - val_acc
uracy: 0.7919 - val_auc: 0.5716
Epoch 7/10
109/109 [==============================] - 1s 6ms/step - loss: 0.5217 - accuracy: 0.7851 - auc: 0.5562 - val_loss: 0.5132 - val_acc
uracy: 0.7918 - val_auc: 0.5432
Epoch 8/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5198 - accuracy: 0.7851 - auc: 0.5575 - val_loss: 0.5076 - val_acc
uracy: 0.7919 - val_auc: 0.5736
Epoch 9/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5199 - accuracy: 0.7851 - auc: 0.5703 - val_loss: 0.5063 - val_acc
uracy: 0.7918 - val_auc: 0.5724
Epoch 10/10
109/109 [==============================] - 1s 7ms/step - loss: 0.5192 - accuracy: 0.7851 - auc: 0.5655 - val_loss: 0.5058 - val_acc
uracy: 0.7918 - val_auc: 0.5809
[0.45770102739334106, 0.5813931822776794, 0.5758665800094604, 0.5637527108192444, 0.5804040431976318, 0.5715550780296326, 0.5431562
066078186, 0.5736348032951355, 0.5723981857299805, 0.5808916687965393]
650/650 [==============================] - 1s 1ms/step
```

In [7]:
```
model.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_2 (InputLayer)        [(None, 21)]              0         
                                                                 
 normalization_1 (Normalizat  (None, 21)               43        
 ion)                                                            
                                                                 
 dense_5 (Dense)             (None, 256)               5632      
                                                                 
 dropout_3 (Dropout)         (None, 256)               0         
                                                                 
 batch_normalization_3 (Batc  (None, 256)              1024      
 hNormalization)                                                 
                                                                 
 dense_6 (Dense)             (None, 256)               65792     
                                                                 
 dropout_4 (Dropout)         (None, 256)               0         
                                                                 
 batch_normalization_4 (Batc  (None, 256)              1024      
 hNormalization)                                                 
                                                                 
 dense_7 (Dense)             (None, 256)               65792     
                                                                 
 dropout_5 (Dropout)         (None, 256)               0         
                                                                 
 batch_normalization_5 (Batc  (None, 256)              1024      
 hNormalization)                                                 
                                                                 
 dense_8 (Dense)             (None, 64)                16448     
                                                                 
 dense_9 (Dense)             (None, 1)                 65        
                                                                 
=================================================================
Total params: 156,844
Trainable params: 155,265
Non-trainable params: 1,579
_________________________________________________________________
```

Model Evaluating

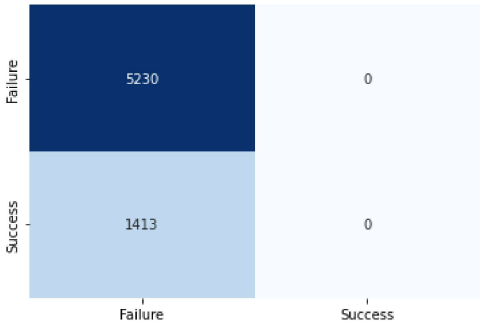
In [8]:
```
results = model.evaluate(x_test, y_test, verbose = 0)
print('Test Loss : {:.4f}%'.format(results[0]*100))
print('Test Accuracy : {:.3f}%'.format(results[1]*100))
print('Test AUC : {:.4f}%'.format(results[2]*100))
```

```
Test Loss : 50.8950%
Test Accuracy : 78.729%
Test AUC : 59.6999%
```

In [9]:
```
y_pred = np.array(model.predict(x_test) >= 0.5, dtype = np.int)
```

```
208/208 [==============================] - 0s 1ms/step
```

In [10]:
```
cm = confusion_matrix(y_test, y_pred)
clr = classification_report(y_test, y_pred, target_names = ['Failure', 'Success'])
sns.heatmap(cm, annot = True, fmt = 'g', cbar = False, cmap = 'Blues')
plt.xticks(ticks = (0.5, 1.5), labels = ['Failure', 'Success'])
plt.yticks(ticks = (0.5, 1.5),labels = ['Failure', 'Success'])
plt.show()
print(f'Classification Report : \n{clr}')
```




```
Classification Report : 
              precision    recall  f1-score   support

     Failure       0.79      1.00      0.88      5230
     Success       0.00      0.00      0.00      1413

    accuracy                           0.79      6643
   macro avg       0.39      0.50      0.44      6643
weighted avg       0.62      0.79      0.69      6643
```

Model Assembling

- Choose models that got top-5 score in the Kaggle private leaderboard and re-made the prediction.

```python
pre = []
test = pd.read_csv('test.csv')
pd_id = test['id']
dirlist = os.listdir('infer')
for doc in dirlist:
    df = pd.read_csv('infer/'+doc)
    # Get the value after ranking with the failure rate
    rank = df['failure'].rank(axis=0).rename(doc)
    pd_id = pd.concat([pd_id, rank],axis=1)

pd_id['min'] = pd_id.drop('id', axis=1).min(axis=1)
pd_id['max'] = pd_id.drop('id', axis=1).max(axis=1)
pd_id['mean'] = pd_id.drop('id', axis=1).mean(axis=1)
pd_id['median'] = pd_id.drop('id', axis=1).median(axis=1)

df = pd_id['max'].copy()
# Normalization
normalized_df=(df-df.min())/(df.max()-df.min())
out = pd.concat([test['id'], normalized_df.rename('failure')],axis=1)

# Generate prediction data
out.to_csv('inference.csv',index=False)
```

- In the end, I chose the max ranking value to do the prediction in those prediction data after several experiments.

	id	model_5847.csv	model_5856.csv	model_5859.csv	model_5869.csv	model_587.csv	model_5872.csv	model_5875.csv	min	max	mean	median
0	26570	9771.0	10200.0	9965.0	10108.0	9888.0	10051.0	10080.0	9771.0	10200.0	10003.777778	10027.388889
1	26571	6448.0	7112.0	6346.0	6425.0	7290.0	6438.0	6714.0	6346.0	7290.0	6712.111111	6580.055556
2	26572	6928.0	6926.0	6444.0	7063.0	6750.0	6765.0	6051.0	6051.0	7063.0	6671.222222	6757.500000
3	26573	9539.0	9117.0	9494.0	9082.0	8977.0	9632.0	9979.0	8977.0	9979.0	9419.555556	9456.777778
4	26574	20248.0	19939.0	19983.0	19941.0	20320.5	20005.0	20008.0	19939.0	20320.5	20078.222222	20006.500000
...	...	...	...	...	...	...	...	...	...	...	...	...
20770	47340	14876.0	14977.0	15070.0	15003.0	14360.0	15148.0	13544.0	13544.0	15148.0	14630.000000	14926.500000
20771	47341	1402.0	1239.0	1582.0	1240.0	1292.0	1352.0	1509.0	1239.0	1582.0	1381.888889	1366.944444
20772	47342	3227.0	1674.0	4167.0	3181.0	1298.0	917.0	1436.0	917.0	4167.0	2331.555556	2002.777778
20773	47343	11604.0	11322.0	11379.0	11835.0	11405.0	11882.0	12043.0	11322.0	12043.0	11648.333333	11626.166667
20774	47344	2698.0	2681.0	2912.0	2885.0	2502.0	2819.0	2493.0	2493.0	2912.0	2710.555556	2704.277778

20775 rows × 12 columns