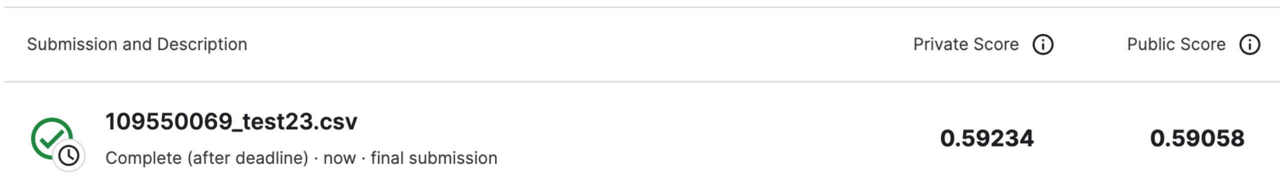
ML Final Project

Introduction

The purpose of the project is to join a real-time Kaggle competition, this particular task Tabular Playground Series - Aug 2022 was to train a machine learning model to predict individual product failures of new codes by the results of a large product testing study. My methodology was to fill the missing values in the dataset with linear models and KNN models in the beginning, then train a neural network model predict the failure and generate a csv file for submission.

Results



Submission and Description	Private Score	Public Score
109550069_test23.csv Complete (after deadline) · now · final submission	0.59234	0.59058

Model Weight: https://drive.google.com/file/d/1gFTlAi2s1r2loJG5coCe7QjdbH3mJLDB/view?usp=share_link
GitHub Link: https://github.com/yrlin411/ML_Final

Environment Details

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

```python
!pip install feature_engine
!pip install tensorflow_addons
```

I finished this assignment in Google Colab, while putting all the data used on my Google Drive, I had to implement the code above to get them, and first of all install the modules that will be imported. Also, I mostly ran it on GPU as my hardware accelerator so the program could be faster.

Methodology - Final_train.ipynb

First I imported all the modules and read the csv files that will be used.

```python
import numpy as np
import pandas as pd
import os
import sys

import joblib # saving sklearn modules
from sklearn.impute import KNNImputer # completing missing values using k-Nearest Neighbors
from feature_engine.encoding import WoEEncoder # encode only categorical variables (type 'object')
from sklearn.linear_model import LogisticRegression, HuberRegressor

from tensorflow.keras import Sequential
from tensorflow.keras.layers import BatchNormalization, Dropout, Dense
import tensorflow as tf
from tensorflow_addons.optimizers import AdamW
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
```

```python
Train = pd.read_csv('/content/drive/MyDrive/大三上作業/ML_final/train.csv')
Test = pd.read_csv('/content/drive/MyDrive/大三上作業/ML_final/test.csv')
```

Next I set up the data by combining the training and testing data, and declare the structures that are going to be used, including codes, feature names, and the columns that didn't contain missing values.

```
[ ]  data = pd.concat([Train, Test])
     # create lists that will be used
     codes = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
     drops = ['id', 'product_code', 'loading', 'attribute_0', 'attribute_1', 'attribute_2', 'attribute_3', 'loading', 'missing3', 'missing5']
     featureM = [f for f in data.columns if f.startswith('measurement') or f=='loading']
     features = ['loading', 'attribute_0', 'measurement_17', 'measurement_0','measurement_1', 'measurement_2', 'attr23', 'missing3', 'missing5']

     # create a dict for measurements 3-17 because they contain value lost (*reference)
     measurements = {}
     # set up measurement 17 manually
     measurements['measurement_17'] = {
             'A': ['measurement_5','measurement_6','measurement_8','measurement_7'],
             'B': ['measurement_4','measurement_5','measurement_7','measurement_9'],
             'C': ['measurement_5','measurement_7','measurement_8','measurement_9'],
             'D': ['measurement_5','measurement_6','measurement_7','measurement_8'],
             'E': ['measurement_4','measurement_5','measurement_6','measurement_8'],
             'F': ['measurement_4','measurement_5','measurement_6','measurement_7'],
             'G': ['measurement_4','measurement_6','measurement_8','measurement_9'],
             'H': ['measurement_4','measurement_5','measurement_7','measurement_8','measurement_9'],
             'I': ['measurement_3','measurement_7','measurement_8','measurement_9']
         }

     # when measurement_3 is missing, the failure rate is 0.160 (much lower than average)
     data['missing3'] = data['measurement_3'].isna().astype(np.int8)
     # when measurement_5 is missing, the failure rate is 0.254 (much higher than average)
     data['missing5'] = data['measurement_5'].isna().astype(np.int8)
     data['attr23'] = data['attribute_2'] * data['attribute_3']
```

In the codes above, I also created a dictionary which contains smaller dictionaries, each representing a measurement which has the code as the keys and the four most correlated measurements other than itself as the values, this part of code is referred to reference [1]. In the website, the author suggested to set up ‘measurement_17’ manually, so I set the exact same values as she did. And since the missing of 'measurement_3’ and ‘measurement_5’ have huge impacts on the failure rate (When ‘measurement_3’ is missing, the failure rate is much lower than the average; When ‘measurement_5’ is missing, the failure rate is much higher than average) as noted in reference[2], I set up the features ‘missing3’ and ‘missing5’ in the model, also, because ‘attribute_2’ has two values (5 and 8) which occur only in the training data, and another value (7) occurs only in the test data while ‘attribute_3’ is similar, like the author mentioned, I simply make up a feature ‘attr23’ representing the multiplication of them instead of using them directly.

```
# get the correlation values of measurements 3-16
colName = []
value = []
for i in range(3,17):
    colName.append(f'measurement_{i}')
    corr = np.absolute(data.drop(drops, axis=1).corr()[f'measurement_{i}']).sort_values(ascending=False)
    value.append(np.sum(corr[1:4]))

# take the measurements in order of sorted correlation
measurementCorr = pd.DataFrame({'column name': colName, 'values': value})
sortedMeasurement = measurementCorr.sort_values(by='values', ascending=False).reset_index(drop=True)
for i in range(10): # only take the top 10
    whichMeasurement = sortedMeasurement.iloc[i, 0]
    best4 = {} # pick 4 measurements with best correlations for each product_code
    for c in codes:
        corr = np.absolute(data[data.product_code == c].drop(drops, axis=1).corr()[whichMeasurement]).sort_values(ascending=False)
        best4[c] = corr[1:5].index.tolist()
    measurements[whichMeasurement] = best4 # measurement_i | product_code : measurement_i1, measurement_i2, measurement_i3, measurement_i4
```

After setting up measurement_17 manually, I set the rest of the dictionaries by simply comparing the correlation values of the other measurements with the product code besides itself.

```
# for each product_code
for code in codes:
    # fill the measurement columns with linear model
    for measure in list(measurements.keys()):
        dataM = data[data.product_code == code]
        best4 = measurements[measure][code] # product_code : measurement_i1, measurement_i2, measurement_i3, measurement_i4
        trainM = dataM[best4+[measure]].dropna(how='any') # 4 related + itself as target (all should not be null)
        testM = dataM[(dataM[best4].isnull().sum(axis=1)==0) & (dataM[measure].isnull())] # 4 related (no null) + itself as target (null)
        modelLinear = HuberRegressor(epsilon=1.75)
        modelLinear.fit(trainM[best4], trainM[measure])
        data.loc[(data.product_code==code)&(data[best4].isnull().sum(axis=1)==0)&(data[measure].isnull()), measure] = modelLinear.predict(testM[best4])
    # fill the others N/A columns with KNN (k=3)
    modelKnn = KNNImputer(n_neighbors=3)
    data.loc[data.product_code==code, featureM] = modelKnn.fit_transform(data.loc[data.product_code==code, featureM])
```

After the most correlated measurements were picked for each measurement, I filled the null spaces in these measurement columns for each product code by splitting them into small training dataset and test datasets and predict them with linear models, I chose HuberRegressor for being robust to outliers, and set the epsilon to 1.75, I once set it to higher values and they performed slightly worse. As for the other null spaces, I chose to use KNN so that it is able to determine the value by three nearest neighbors instead of guessing values with low correlations. After the training, the previous part of data pre-processing is done.

```
    train = data.iloc[:len(Train.index)]
    encode = WoEEncoder(variables=['attribute_0'])
    encode.fit(train, train['failure'])
    train = encode.transform(train)
```

After that I one-hot encoded attribute_0 following reference[2] since it is a categorical feature, in the reference, attribute_1 was also encoded but the performance was better without it so I dropped it.

```
# first 3 codes for training, latter 2 for validation (*reference)
folds = {'fold_1': [['C', 'D', 'E'], ['A', 'B']],
         'fold_2': [['B', 'D', 'E'], ['A', 'C']],
         'fold_3': [['B', 'C', 'E'], ['A', 'D']],
         'fold_4': [['B', 'C', 'D'], ['A', 'E']],
         'fold_5': [['A', 'D', 'E'], ['B', 'C']],
         'fold_6': [['A', 'C', 'E'], ['B', 'D']],
         'fold_7': [['A', 'C', 'D'], ['B', 'E']],
         'fold_8': [['A', 'B', 'E'], ['C', 'D']],
         'fold_9': [['A', 'B', 'D'], ['C', 'E']],
         'fold_10': [['A', 'B', 'C'], ['D', 'E']]}
```

Before getting into the training, since I decided to implement the training with a K-Fold-validation-like structure, I referred to one of the methods of splitting the training dataset into a 3:2 portion in every way of combination used in reference[3] in order to avoid overfitting due to the difference in the modalities between the measurements in the train set and those in the test set.

```
tf.random.set_seed(85)
# set up the callbacks
bestModel = ModelCheckpoint('best.h5', verbose=1, save_best_only=True, monitor="auc", mode="max")
earlyStop = EarlyStopping(patience=10, restore_best_weights=True)
reduceLr = ReduceLROnPlateau(monitor="auc", factor=0.9, patience=5, mode="max", min_delta=0.0001)

# run through all the folds
for fold in folds.keys():
    print(f'\n{fold}\n')

    x_train, y_train = train[train['product_code'].isin(folds[fold][0])][features].values, train[train['product_code'].isin(folds[fold][0])]['failure'].values
    x_valid, y_valid = train[train['product_code'].isin(folds[fold][1])][features].values, train[train['product_code'].isin(folds[fold][1])]['failure'].values

    # create a new model for each fold
    model = Sequential()
    model.add(BatchNormalization())
    model.add(Dense(len(x_train), activation="relu"))
    model.add(BatchNormalization())
    model.add(Dense(128, activation="relu"))
    model.add(BatchNormalization())
    model.add(Dense(64, activation="relu"))
    model.add(BatchNormalization())
    model.add(Dense(32, activation="relu"))
    model.add(BatchNormalization())
    model.add(Dense(16, activation="relu"))
    model.add(BatchNormalization())
    model.add(Dense(1, activation="sigmoid"))

    model.compile(optimizer=AdamW(learning_rate=1e-3, weight_decay=1e-3), loss=BinaryCrossentropy(), metrics=["AUC"])
    model.fit(x_train, y_train, batch_size = 64, epochs = 100, callbacks=[earlyStop, reduceLr, bestModel], validation_data=(x_valid, y_valid))
```

For this assignment I went for Keras to build my model, so first of all I set a random seed and the call backs. I used ModelCheckpoint to save the best model for each epoch in all folds, and used EarlyStopping to stop the fold from training if the accuracy isn't improving in the recent 10 epochs, and ReduceLROnPlateau to decrease the learning rate also if the accuracy isn't increasing in the recent 5 epochs (the number is smaller than the early-stopping one so the training of the fold won't be stopped way too early). In each fold, set up the training data and validation data with the codes also referring to reference[3]. As for the model, I simply used 6 Dense layers, each with a Batch-Normalization layer before it in order to normalize the features at first to accelerate the convergence and regularize the model. The unit starts with the length of the training data which is approximately 256 and divided by 2 each time until it reaches 16 and one more for converging to 1. Most of the layers uses the rectified linear activation function 'relu', while the last layer uses sigmoid function to make sure it is simple and continuous. Lastly compile the model and train it.

Methodology – Final_inference.ipynb

The inference code is mostly the same as the training file, except in the pre-processing I also generated the testing data frame.

```
[ ] tf.random.set_seed(85)
    from keras.models import load_model
    Model = load_model('best.h5')
    predictions = Model.predict(test[features].values)

    650/650 [==============================] - 6s 8ms/step
```

```
[ ] from google.colab import files
    submission['failure'] = predictions
    submission.to_csv('output.csv', index=False)
    files.download('output.csv')
```

```
[ ] !pip freeze >requirements.txt
```

Then I loaded the trained model and make predictions with it, and set the values of the column 'failure' to the predictions and make it a csv file for submission. The last line was to get the requirements of the environment so it could be reproduced.

Summary

Using a linear model and a KNN model in order to fill up the missing parts of a real-world data set by the most correlated measurements of each measurement, and a neural network model to make predictions while training and validating the folds in different permutation of the data, the above method surpassed the baseline accuracy of 0.58990 on the private score.

References

[1] https://irvifa.medium.com/tabular-playground-series-august-2022-c0c8d311e123
[2] https://www.kaggle.com/code/ambrosm/tpsaug22-eda-which-makes-sense/notebook
[3] https://www.kaggle.com/competitions/tabular-playground-series-aug-2022/discussion/349810