Sanaz Salari

NON-SEPARABLE CLASSIFICATION – PATTERN RECOGNITION IN PROCESS CONTROL CHARTS:

This problem is a multi-input and multi-output classification issue that we would like to solve by MLP algorithm. This is a classification problem which involves 7 classes as outputs and 10 inputs. The task here is to design and train an MLP to minimize the error between actual and predicted output values. We have 2500 data points as training dataset and 250 datapoints as testing dataset. The steps I did from beginning to end will be explained in the following.

1. **install and import the required libraries** such as TensorFlow and Keras and fix random seed for reproducibility:

```
import sklearn
import skopt
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.model_selection import RepeatedKFold
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.models import Sequential
from keras.layers import Dense, Dropout
np.random.seed(5)
```

2. **Import the data** from the local drive and do preprocess. Split train and test datasets as two .csv files on a local drive and import them as follows:

```
#Load data from local drive
from google.colab import files
uploaded = files.upload()
import pandas as pd
import io
train0 = pd.read_csv(io.BytesIO(uploaded['Pattern_train.csv']))
test0 = pd.read_csv(io.BytesIO(uploaded['Pattern_test.csv']))
```

```
train0.head()
```

	input1	input2	input3	input4	input5	input6	input7	input8	input9	input10	in_control	fault1	fault2	fault3	fault4	fault5	fault6
0	-1.691	-2.426	1.440	-2.688	2.240	-2.577	1.541	-0.045	-1.093	-0.498	1	0	0	0	0	0	0
1	1.337	-0.014	-1.134	1.714	1.821	-1.989	-1.056	-1.823	2.408	0.906	1	0	0	0	0	0	0
2	-0.278	0.169	-2.756	-0.819	2.074	2.179	-0.616	1.405	-2.847	0.325	1	0	0	0	0	0	0
3	1.156	1.827	1.084	-0.675	-2.409	2.970	1.037	-1.972	-0.191	1.514	1	0	0	0	0	0	0
4	-2.849	-2.715	0.909	-0.211	2.050	-2.632	1.746	-0.534	1.426	-1.512	1	0	0	0	0	0	0

3. **Check the types of input and output values**

   Types of data in both test and train datasets are checked to see whether we have any categorical data and if there are such values, convert them to numerical ones. All data types are float64 and int64. So, we don't need to convert data types.

```
#To see what data types we have(if we have categorical datatypes, encode them)
train0.dtypes
test0.dtypes
```

```
input1         float64
input2         float64
input3         float64
input4         float64
input5         float64
input6         float64
input7         float64
input8         float64
input9         float64
input10        float64
in_control       int64
fault1           int64
fault2           int64
fault3           int64
fault4           int64
fault5           int64
fault6           int64
dtype: object
```

4. **Check any NA values**

   We don't have any NA values in this dataset.

```python
#Clean the data
train0.isna().sum()
test0.isna().sum()
```

```
input1        0
input2        0
input3        0
input4        0
input5        0
input6        0
input7        0
input8        0
input9        0
input10       0
in_control    0
fault1        0
fault2        0
fault3        0
fault4        0
fault5        0
fault6        0
dtype: int64
```

5. **Find duplicated values in the features if we have**

   In the training dataset, we don't have any duplicated values.

```python
#Find Duplicated Data in training
train0[train0.duplicated(['input1','input2','input3','input4','input5','input6','input7','input8','input9','input10'])]
print(train0.duplicated().sum())
```

```
0
```

   In the test dataset, we don't have any duplicated values.

```python
#Find Duplicated Data in test dataset
test0[test0.duplicated(['input1','input2','input3','input4','input5','input6','input7','input8','input9','input10'])]
print(test0.duplicated().sum())
```

```
0
```

6. **Inspect the data to see whether it needs to do normalization or not**

```python
#Inspect the data to see whether it needs to do normalization or not
train1_stats = train0.describe()
train1_stats = train1_stats.transpose()
train1_stats
```

	count	mean	std	min	25%	50%	75%	max
input1	2500.0	-0.005568	1.658949	-2.997	-1.35325	-0.0185	1.39200	2.998
input2	2500.0	0.034232	1.661364	-2.995	-1.32050	0.0655	1.38675	3.000
input3	2500.0	-0.019945	1.616737	-2.997	-1.30225	0.0225	1.27000	2.995
input4	2500.0	0.008083	1.673683	-2.996	-1.38725	0.0540	1.34925	2.997
input5	2500.0	-0.021666	1.585194	-2.996	-1.44275	0.0215	1.35950	2.999
input6	2500.0	-0.022674	1.534029	-2.994	-1.22925	-0.0535	1.19150	2.994
input7	2500.0	0.023896	1.483411	-2.999	-1.00325	0.0350	1.07425	2.997
input8	2500.0	-0.001836	1.501475	-2.991	-1.06725	-0.0110	1.04250	3.000
input9	2500.0	-0.010894	1.536349	-3.000	-1.14200	0.0435	1.10625	2.996
input10	2500.0	-0.027626	1.998715	-4.476	-1.58650	-0.0085	1.49325	4.491
in_control	2500.0	0.400000	0.489996	0.000	0.00000	0.0000	1.00000	1.000
fault1	2500.0	0.100000	0.300060	0.000	0.00000	0.0000	0.00000	1.000
fault2	2500.0	0.100000	0.300060	0.000	0.00000	0.0000	0.00000	1.000
fault3	2500.0	0.100000	0.300060	0.000	0.00000	0.0000	0.00000	1.000
fault4	2500.0	0.100000	0.300060	0.000	0.00000	0.0000	0.00000	1.000
fault5	2500.0	0.100000	0.300060	0.000	0.00000	0.0000	0.00000	1.000
fault6	2500.0	0.100000	0.300060	0.000	0.00000	0.0000	0.00000	1.000

7. Normalization

To decrease the training model sensitivity to features scale, we need to normalize all training features (not outputs) which have different ranges to change their values to a common scale. Normalization is important because the input variables will be multiplied by the model weights and the scale of outputs will be affected by the scale of features. The goal is to predict output with the best accuracy, but we don't want to change the output.

Training data normalization:

```python
#Standardization
train2=(train0 - train0.min())/(train0.max() - train0.min())
train2.head()
```

	input1	input2	input3	input4	input5	input6	input7	input8	input9	input10	in_control	fault1	fault2	fault3	fault4	fault5	fault6
0	0.217848	0.094912	0.740487	0.051393	0.873394	0.069639	0.757171	0.491738	0.318045	0.443627	1.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.722936	0.497248	0.310915	0.785917	0.803503	0.167836	0.324049	0.194959	0.901935	0.600201	1.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.453545	0.527773	0.040220	0.363257	0.845705	0.863894	0.397432	0.733767	0.025517	0.535408	1.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.692744	0.804337	0.681075	0.387285	0.097915	0.995992	0.673115	0.170088	0.468479	0.668005	1.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.024687	0.046706	0.651869	0.464709	0.841701	0.060454	0.791361	0.410115	0.738159	0.330545	1.0	0.0	0.0	0.0	0.0	0.0	0.0

Test data normalization:

```python
#Normalization
test2=(test0 - train0.min())/(train0.max() - train0.min())
test2.head()
```

	input1	input2	input3	input4	input5	input6	input7	input8	input9	input10	in_control	fault1	fault2	fault3	fault4	fault5	fault6
0	0.217848	0.094912	0.740487	0.051393	0.873394	0.069639	0.757171	0.491738	0.318045	0.443627	1.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.722936	0.497248	0.310915	0.785917	0.803503	0.167836	0.324049	0.194959	0.901935	0.600201	1.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.453545	0.527773	0.040220	0.363257	0.845705	0.863894	0.397432	0.733767	0.025517	0.535408	1.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.692744	0.804337	0.681075	0.387285	0.097915	0.995992	0.673115	0.170088	0.468479	0.668005	1.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.024687	0.046706	0.651869	0.464709	0.841701	0.060454	0.791361	0.410115	0.738159	0.330545	1.0	0.0	0.0	0.0	0.0	0.0	0.0

8. Split features from labels for training and testing datasets

Separate the target values from the features. These targets are the values that we will train the model to predict.

```python
#Split features from labels
#Separate the target values from the features. This targets are the values that we will train the model to predict.
train2_features = train2.copy()
test2_features = test2.copy()
train2_labels = train2_features.iloc[:, 10:17]

train2_features.drop(['in_control','fault1','fault2','fault3','fault4','fault5','fault6',], axis = 1, inplace = True)
print(train2_features.shape)
print(train2_labels.shape)
```

```
(2500, 10)
(2500, 7)
```

```python
#Split features from labels
#Separate the target values from the features. This targets are the values that we will train the model to predict.
test2_features = test2.copy()
test2_labels = test2_features.iloc[:, 10:17]

test2_features.drop(['in_control','fault1','fault2','fault3','fault4','fault5','fault6',], axis = 1, inplace = True)
print(test2_features.shape)
print(test2_labels.shape)
```

```
(250, 10)
(250, 7)
```

9. Change train and test datasets from matrix to array:

```
# Change train and test datasets from matrix to array:
X_train = train2_features.to_numpy()
y_train = train2_labels.to_numpy()
X_test = test2_features.to_numpy()
y_test = test2_labels.to_numpy()
X_train = X_train.reshape([X_train.shape[0], -1])
X_test = X_test.reshape([X_test.shape[0], -1])

y_train= train2_labels
y_test= test2_labels
# Printing dimensions
print(X_train.shape, y_train.shape)
```

```
(2500, 10) (2500, 7)
```

10. Define a model with a single layer

The model is defined with sequential keras syntax. As we told earlier, we have 7 outputs and 10 features. Since the problem is classification, the activation function is chosen as “sigmoid” and for the compile part, the optimizer is “adam”, the loss function is “categorical crossentropy” and the metrics is “accuracy”.

```
model = keras.Sequential([
                    keras.layers.Dense(7, input_shape=(10,), activation='sigmoid')
])

model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

model.fit(X_train, y_train, epochs=5)
```

```
Epoch 1/5
79/79 [==============================] - 0s 1ms/step - loss: 2.0184 - accuracy: 0.1492
Epoch 2/5
79/79 [==============================] - 0s 2ms/step - loss: 1.8709 - accuracy: 0.2716
Epoch 3/5
79/79 [==============================] - 0s 2ms/step - loss: 1.8058 - accuracy: 0.3648
Epoch 4/5
79/79 [==============================] - 0s 2ms/step - loss: 1.7847 - accuracy: 0.3940
Epoch 5/5
79/79 [==============================] - 0s 2ms/step - loss: 1.7796 - accuracy: 0.3976
<keras.callbacks.History at 0x7f21a98ea710>
```

The result of running the model will be as follows:

```
Epoch 1/5
79/79 [==============================] - 1s 2ms/step - loss: 1.8398 - accuracy: 0.3568
Epoch 2/5
79/79 [==============================] - 0s 4ms/step - loss: 1.7869 - accuracy: 0.3964
Epoch 3/5
79/79 [==============================] - 0s 3ms/step - loss: 1.7752 - accuracy: 0.3996
Epoch 4/5
79/79 [==============================] - 0s 2ms/step - loss: 1.7725 - accuracy: 0.4000
Epoch 5/5
79/79 [==============================] - 0s 3ms/step - loss: 1.7710 - accuracy: 0.4000
<keras.callbacks.History at 0x7f21ab0acc10>
```

As we see in the result window, the loss value is about 1.7 and accuracy is about 0.4. The loss value is large, and the accuracy is low. So, we need to improve the model by adding some layers or nodes.

11. Evaluate the model on the test dataset

```
#Evaluate model on the test dataset
model.evaluate(X_test, y_test)
```

```
8/8 [==============================] - 0s 2ms/step - loss: 1.7662 - accuracy: 0.4000
[1.7662326097488403, 0.4000000059604645]
```

12. sample prediction

```
#sample prediction
y_predicted = model.predict(X_test)
y_predicted[2]
```

```
array([0.749457  , 0.3646825 , 0.40812185, 0.40990818, 0.39175245,
       0.46627772, 0.4184214 ], dtype=float32)
```

As we see, since the model accuracy is low, it cannot predict the target value accurately.

13.To improve the model, we will add hidden layers

Let's create a helper function that builds the model with various parameters. Builds a Sequential MLP model using Keras.

The model has two hidden layers and one output layer.

Activation function for each hidden layer is “Relu” and for the output layer is “sigmoid” since we have a classification problem.

As we want to solve a regularization problem, the loss function is “cross entropy”.

Regularization method is “He Uniform” for hidden layers as we use “Relu” function and “GlorotUniform” for the output layer as we use “sigmoid” function in the output layer.

```
#To improve model, we will add hidden layers
from tensorflow.keras import regularizers
from tensorflow.keras import layers
from tensorflow.keras import initializers
model = keras.Sequential([
                    keras.layers.Dense(30, input_shape=(10,), activation='relu',
                    kernel_regularizer=regularizers.l1_l2(l1=1e-4, l2=1e-4),
                    bias_regularizer=regularizers.l2(1e-4),
                    activity_regularizer=regularizers.l2(1e-4),
                    kernel_initializer = tf.keras.initializers.HeUniform(),
                    name="dense_1"),
                    keras.layers.Dense(12, input_shape=(10,), activation='relu',
                    kernel_regularizer=regularizers.l1_l2(l1=1e-4, l2=1e-4),
                    bias_regularizer=regularizers.l2(1e-4),
                    activity_regularizer=regularizers.l2(1e-4),
                    kernel_initializer = tf.keras.initializers.HeUniform(),
                    name="dense_2"),
                    keras.layers.Dense(7, activation='sigmoid',kernel_initializer = tf.keras.initializers.GlorotUniform())
])

model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

model.fit(X_train, y_train, epochs=100)
```

The result would be as follows:

```
79/79 [==============================] - 0s 2ms/step - loss: 0.3443 - accuracy: 0.9168
Epoch 84/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3412 - accuracy: 0.9116
Epoch 85/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3390 - accuracy: 0.9160
Epoch 86/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3398 - accuracy: 0.9136
Epoch 87/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3376 - accuracy: 0.9148
Epoch 88/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3325 - accuracy: 0.9156
Epoch 89/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3333 - accuracy: 0.9104
Epoch 90/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3341 - accuracy: 0.9120
Epoch 91/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3294 - accuracy: 0.9144
Epoch 92/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3261 - accuracy: 0.9204
Epoch 93/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3243 - accuracy: 0.9172
Epoch 94/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3250 - accuracy: 0.9132
Epoch 95/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3239 - accuracy: 0.9180
Epoch 96/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3248 - accuracy: 0.9148
Epoch 97/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3205 - accuracy: 0.9172
Epoch 98/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3234 - accuracy: 0.9148
Epoch 99/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3163 - accuracy: 0.9176
Epoch 100/100
79/79 [==============================] - 0s 2ms/step - loss: 0.3152 - accuracy: 0.9160
<keras.callbacks.History at 0x7f21aaab9b10>
```

Loss value decreases and accuracy increases by each epoch which shows no overfitting or underfitting. The last loss is 0.31 and the last accuracy is 0.91.

14. Evaluate model on the test dataset

Although test prediction shows a higher loss and lower accuracy for the testing dataset, there is no overfitting or underfitting since the loss and accuracy between training and test results are small.

```python
#Evaluate model on the test dataset
model.evaluate(X_test, y_test)
```

```
8/8 [==============================] - 0s 2ms/step - loss: 0.3560 - accuracy: 0.9080
[0.35598263144493103, 0.9079999923706055]
```