INTRODUCTION :

Detecting fraud transactions is of great importance for any credit card company. We are tasked by a well-known company to detect potential frauds so that customers are not charged for items that they did not purchase. So the goal is to build a classifier that tells if a transaction is a fraud or not.

Data processing:

To quickly understand the distribution of each variable, let's try below.

As you notice, the variable *'Amount'* ranges from 0 to 25,691.16. To reduce its wide range, we use **Standardization** to remove the mean and scale to unit variance, so that 68% of the values lie in between (-1, 1). If you want more details on this topic, read this article.

Specifically,

```
scaler = StandardScaler()data['NormalizedAmount'] =
scaler.fit_transform(data['Amount'].values.reshape(-1, 1))
```

Note we have to use *".values.reshape(-1, 1)"* to convert a **Series** to an **array** and reshape to a 2D array. Try to remove this part and see the error you get. This is a common one !!

Now let's split the data into X and y. Specifically,

```
data = data.drop(['Amount', 'Time'], axis = 1)
y = data['Class']
X = data.drop(['Class'], axis = 1)
```

The column *['Amount', 'Time']* are dropped as they are not needed for model building. Note, we set **axis = 1** when dropping the column. This is another common error if you are not familiar with the drop function !

Finally, let's split the data into train and test datasets. Specifically,

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.3, random_state = 0)
```

DNN Model building

Here we will build a 5-layer deep neural networking using the Sequential model in **Keras**. Specifically,

```
model = Sequential([
Dense(input_dim = 29, units = 16, activation = 'relu'),
Dense(units = 24, activation = 'relu'),
Dropout(0.5),
Dense(units = 20, activation = 'relu'),
Dense(units = 24, activation = 'relu'),
Dense(units =1, activation = 'sigmoid'),])
```

For the 1st hidden layer, '*input_dim*' is the number of input variables. '*units*' is the number of nodes or neurons in each layer.

We use Rectified Linear Unit (*ReLU*) as an activation function for the hidden layers. ***ReLU* normally performs better than Sigmoid and Hyperbolic Tangent functions when building deep neural networks. This is because Sigmoid and Tanh tends to saturate when the input value is either too large or too small. In addition, they only show a high gradient around their mid-points, such as 0.5 for sigmoid and 0 for tanh.** If you want more details on *ReLU*,

feel free to read this article.

We use the Sigmoid function in the output layer for a binary classification problem. If you want to learn more about how to create a neural network, read this article.

DNN model evaluation

With the model architecture built, let's compile and train the model. Specifically,

```
model.compile(optimizer = 'adam', loss = 'binary_crossentropy',
metrics = ['accuracy'])
model.fit(X_train, y_train, batch_size = 15, epochs = 5)
```

Note above we use '*binary_crossentropy*' as the loss function and '*Adam*' to update network weights. *'Adam'* is a popular algorithm to achieve good results fast in the deep learning field. If you want more details on *Adam*, feel free to read this article.

The model weights are updated every 15 samples. If you are not clear with the concept of epoch and batch. **An epoch represents**

one time that the entire training set is fed through the network. A batch defines the number of samples to iterate through before updating the internal model parameters. At the end of the batch, the predictions are compared to the expected output, and error is calculated. Using this error, the optimizer improves the model by moving down the error gradient.

Great. Now let's evaluate the model. Specifically,

Fig.2 DNN evaluation

Remember we use '**accuracy'** as metrics. The model is found with 99.94% accuracy !

Now one question for you: does such a high accuracy indicate good performance? If you recall, accuracy is the sum of Ture Negative and True Positive divided by total dataset size. If 95% of the dataset is Negative (non-frauds), the network will cleverly predict all to be Negative, leading to 95% accuracy. **However, for fraud detection, detecting Positive matters more than detecting Negative.** Therefore, we need better metrics.

Fig.3 shows the confusion matrix using the test dataset. DNN shows a precision of 85.3%, a recall of 78.9%, and an F1 score of 82.0%. About 20% of frauds are misclassified as non-frauds, leading to these extra payments for the customers, though the accuracy is 99.94%. So there is enough space to improve the DNN model .

Fig.3 DNN confusion matrix

Decision Tree

As Wikipedia states, a decision tree is a flowchart-like structure in which each internal node represents a test on a feature (e.g. weather if sunny or rainy), and each leaf node is a class label made after all tests. A decision tree aims to learn ways to split datasets based on conditions. So, it is non-parametric.

Now let's build a decision tree model. Specifically,

```
from sklearn.tree import DecisionTreeClassifier
decision_tree_model = DecisionTreeClassifier()
```

```
decision_tree_model.fit(X_train, y_train)
y_pred = decision_tree_model.predict(X_test)
```

Fig.4 shows the confusion matrix of the model. The decision tree gives a precision of 82.7%, a recall of 74.8%, and an F1 score of 78.6%, worse than the DNN model ☹☹!

Fig.3 Decision tree confusion matrix

Random forest

Conceptually, a random forest (RF) is a collection of decision trees. Each tree votes a class and the class received the most votes is the predicted class. A decision tree is built on the whole dataset, while a random forest randomly selects features to build multiple decision trees and average the result. If you want to learn more about how RF works and parameter optimization, read this article.

Specifically,

```
from sklearn.ensemble import RandomForestClassifier
rf_model = RandomForestClassifier(n_estimators = 100)
rf_model.fit(X_train, y_train)
y_pred = rf_model.predict(X_test)
```

It takes a few minutes to train the model. In the end, RF shows a precision of 95%, a recall of 77.5%, and an F1 score of 85.3%, better than the DNN and decision tree model !

Fig.3 Random Forest confusion matrix

Sampling

An issue about the dataset we have here is a class imbalance. Only 0.17% of 284,807 transactions are frauds. Sadly, the model is more sensitive to detect the majority class than the minority class. In general, there are two techniques to tackle class imbalance, under-sampling, and over-sampling.

Under-sampling

The simplest strategy is to randomly select the majority class to balance with the minority class. But the limitation is that data randomly removed from the majority class may be useful to create a robust model.

Let's implement this strategy first. Specifically,

```
fraud_ind = np.array(data[data.Class == 1].index)
normal_ind = data[data.Class == 0].index
random_normal_ind = np.random.choice(normal_ind, num_frauds,
replace = False)
random_normal_ind = np.array(random_normal_ind)
under_sample_data = data.iloc[under_sample_ind, :]X_undersample =
under_sample_data.iloc[:, under_sample_data.columns != 'Class']
y_undersample = under_sample_data.iloc[:, under_sample_data.columns
== 'Class']
```

Above, we randomly selected the same amount of non-frauds as the fraud and created a new dataset. With the down-sized data, re-train the DNN model. In the end, under-sampling gives a precision of 100%, a recall of 87.6%, and an F1 score of 93.4%. Much better than DNN without under-sampling !

8.2 SMOTE

The simplest way of over-sampling is to duplicate data in the minority class, but no new information will be added to the

model. Alternatively, we can synthesize data from existing ones, referred as Synthetic Minority Over-sampling, or SMOTE for short.

SMOTE, initially presented by Nitesh Chawla in 2002, works by selecting data that are close or similar in the feature space and drawing a line between data and make new data at a point on the line. *It is effective because new data are close to the minority class in the feature space. If you want to learn about SMOTE, please feel free to read Nitesh's article.*

Now, let's implement SMOTE and re-train the model. Specifically,

```
from imblearn.over_sampling import SMOTE
X_resample, y_resample = SMOTE().fit_sample(X, y)
```

The above code created a balanced '*y_resample*' with 284,315 frauds and 284,315 non-frauds. After re-training the DNN model, SMOTE gives a precision of 99.8%, a recall of 100%, and an F1 score of 99.9%. Much better than DNN with and without under-sampling !

Summary

As a summary, we created 5 models, DNN, Decision Tree, and Random Forest, DNN with under-sampling, and DNN with SMOTE. As shown in the Table below, DNN with SMOTE performs best.