# Final Project: Fake Tasks & CGAN

## ELG5142[EG] Ubiquitous Sensing / Smart Cities

**Group Number: G_26**


**Prepared by:**

**Sawsan Awad (300327224)**

**Sondos Ali (300327219)**

**Toka Mostafa (300327284)**

## 1. Overview

Nowadays the world is suffering from fake tasks. So, the purpose of Generative Adversarial Networks (GAN) is to build two networks one network called generator which is specified to generate fake tasks, and the other network called discriminator which is used to distinguish between the fake and legitimate tasks. But the GAN has some limitation is being overcome by conditional GAN (cGAN) which is passed the label class to the network. So, in this project we first establish our models which are random forest, naïve based, and Adaboost to just classify fake and legitimate tasks. Then we applied a generator to generate 2000 fake tasks and then passed these fake tasks to models to see its performance which is be very low because he will test data see it for first time. After that, we applied a discriminator to those fake tasks to be distinguished and labeled then we applied the fake tasks to the models again to see its performances after the discriminator which is increased from 59% to 92% for a random forest algorithm and for Adaboost 57% to 93%.

## 2. Methodolgy

We followed some defined steps to obtain the aimed results:

### 2.1. Install important libraries:

- NumPy library: it provides a lot of supporting functions that make working with ndarray very easy.
- Pandas library: it helps us to analyze and understand data better.
- Matplotlib.pyplot library: used to create 2D graphs and plots by using python scripts. It has a module named pyplot which makes things easy for plotting by providing feature to control line styles, font properties, formatting axes etc.
- from sklearn.model_selection import train_test_split: is a function in Sklearn model selection for splitting data arrays into two subsets for training data and for testing data. With this function, we don't need to divide the dataset manually. By default, Sklearn train_test_split will make random partitions for the two subsets.
- from sklearn.preprocessing import MinMaxScaler: is the Python object from the Scikit-learn library that is used for normalising our data.

- from sklearn.ensemble import RandomForestClassifier: is a classification algorithm consisting of many decision trees. It uses bagging and feature randomness when building each individual tree to try to create an uncorrelated forest of trees whose prediction by committee is more accurate than that of any individual tree.
- from sklearn.naive_bayes import GaussianNB: A Gaussian Naive Bayes algorithm is a special type of NB algorithm. It's specifically used when the features have continuous values. It's also assumed that all the features are following a gaussian distribution i.e, normal distribution.
    - Naïve Bayes (NB) Classifier: is Bayesian graphical model that has nodes corresponding to each of the columns or features. It is called naive because, it ignores prior distribution of parameters and assume independence of all features and all rows. Ignoring prior has both an advantage and disadvantage.
- from sklearn.ensemble import AdaBoostClassifier: is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.
- from sklearn.metrics import classification_report, accuracy_score:
    - Classification_report: is a performance evaluation metric in machine learning. It is used to show the precision, recall, F1 Score, and support of your trained classification model, and it will return accuracy.
    - The accuracy_score: is function computes the accuracy, either the fraction (default) or the count (normalize=False) of correct predictions.
- from sklearn.ensemble import VotingClassifier: is a machine learning model that trains on an ensemble of numerous models and predicts an output (class) based on their highest probability of chosen class as the output.
- Import tensorflow as tf: is a machine learning framework that is provided by Google. It is an open-source framework used in conjunction with Python to implement algorithms, deep learning applications and much more. It is used in research and for production purposes. It has optimization techniques that help in performing complicated mathematical operations quickly.

- from tensorflow import keras: is a deep learning API, which is written in Python. It is a high-level API that has a productive interface that helps solve machine learning problems. It runs on top of Tensorflow framework. It was built to help experiment in a quick manner. It provides essential abstractions and building blocks that are essential in developing and encapsulating machine learning solutions. Keras is already present within the Tensorflow package.
- from keras.models import Sequential: It allows us to create a deep learning model by adding layers to it. Here, every unit in a layer is connected to every unit in the previous layer.
- from tensorflow.keras import layers: are the basic building blocks of neural networks in Keras. A layer consists of a tensor-in tensor-out computation function (the layer's call method) and some state, held in TensorFlow variables (the layer's weights).
- from keras.layers import InputLayer, Dense, Dropout,BatchNormalization:
    - InputLayer: the input layer itself is not a layer, but a tensor. It's the starting tensor you send to the first hidden layer. This tensor must have the same shape as your training data.
    - Dense: is the regular deeply connected neural network layer. It is most common and frequently used layer. Dense layer does the below operation on the input and return the output.
        - output = activation(dot(input, kernel) + bias)
    - Dropout: is a regularization technique to prevent overfitting in a neural network model training. The method randomly drops out or ignores a certain number of neurons in the network. Dropout technique is useful when we train two-dimensional convolutional neural networks to reduce overfitting with huge numbers of nodes in a network.
    - BatchNormalization: is a technique to normalize the activation between the layers in neural networks to improve the training speed and accuracy (by regularization) of the model. It is intended to reduce the internal covariate shift for neural networks. It works well with image data training

and it is widely used in training of Generative Adversarial Networks (GAN) models.

- from keras.layers.advanced_activations import LeakyReLU: Leaky version of a Rectified Linear Unit. It allows a small gradient when the unit is not active
- from tensorflow.keras.optimizers import Adam: Optimizer that implements the Adam algorithm. Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.
- Other libraries will be shown their importance in the code.

## Importing Libraries

```
[1] # Essential libraries
    import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt

    # Sklearn libraries
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import MinMaxScaler
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.naive_bayes import GaussianNB
    from sklearn.ensemble import AdaBoostClassifier
    from sklearn.metrics import classification_report, accuracy_score
    from sklearn.ensemble import VotingClassifier

    # Tensorflow libraries
    import tensorflow as tf
    from tensorflow import keras
    from keras.models import Sequential
    from tensorflow.keras import layers
    from keras.layers import InputLayer, Dense, Dropout,BatchNormalization
    from keras.layers.advanced_activations import LeakyReLU
    from tensorflow.keras.optimizers import Adam
```

## 2.2.Importing datasets:

- Firstly, we use .read_csv to read the dataset.
- Secondly, we use .head() function to display the first five rows of the data frame by default.

Importing the dateset

```
[4] data = pd.read_csv('/content/MCSDatasetNEXTCONLab.csv')
    data.head()
```

| | ID | Latitude | Longitude | Day | Hour | Minute | Duration | RemainingTime | Resources | Coverage | OnPeakHours | GridNumber | Ligitimacy |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 45.442142 | -75.303369 | 1 | 4 | 13 | 40 | 40 | 9 | 91 | 0 | 131380 | 1 |
| 1 | 1 | 45.442154 | -75.304366 | 1 | 4 | 23 | 40 | 30 | 9 | 91 | 0 | 131380 | 1 |
| 2 | 1 | 45.442104 | -75.303963 | 1 | 4 | 33 | 40 | 20 | 9 | 91 | 0 | 121996 | 1 |
| 3 | 1 | 45.441868 | -75.303577 | 1 | 4 | 43 | 40 | 10 | 9 | 91 | 0 | 121996 | 1 |
| 4 | 2 | 45.447727 | -75.147722 | 2 | 15 | 49 | 30 | 30 | 5 | 47 | 0 | 140784 | 1 |

- Thirdly, , we split the dataset by a unique function called .iloc[], which is used to select a value that belongs to a particular row or column from a set of values of a data frame or dataset. For example, from our code, we gave it [:,:-1] in inputs, which means that retrieve all rows and all columns except the last column.
- Fourthly, we use .iloc[:,-1].values with y to select all rows and only the last column for output.

```
[5] # Splitting the dataset into inputs and outputs
    x = data.iloc[:,:-1].values
    y = data.iloc[:,-1].values
    x

    array([[ 1.00000000e+00,  4.54421419e+01, -7.53033693e+01, ...,
             9.10000000e+01,  0.00000000e+00,  1.31380000e+05],
           [ 1.00000000e+00,  4.54421541e+01, -7.53043661e+01, ...,
             9.10000000e+01,  0.00000000e+00,  1.31380000e+05],
           [ 1.00000000e+00,  4.54421041e+01, -7.53039633e+01, ...,
             9.10000000e+01,  0.00000000e+00,  1.21996000e+05],
           ...,
           [ 4.00000000e+03,  4.54366819e+01, -7.51524163e+01, ...,
             6.30000000e+01,  0.00000000e+00,  1.22015000e+05],
           [ 4.00000000e+03,  4.54369777e+01, -7.51532778e+01, ...,
             6.30000000e+01,  0.00000000e+00,  1.22015000e+05],
           [ 4.00000000e+03,  4.54369829e+01, -7.51532401e+01, ...,
             6.30000000e+01,  0.00000000e+00,  1.22015000e+05]])

[6] y

    array([1, 1, 1, ..., 1, 1, 1])
```

## 2.3.Splitting dataset into train and test sets:

- Here we use train_test_split function to split the data for splitting data arrays into two subsets for training data and for testing data, ( x_train, y_train, x_test and y_test).

### Splitting dataset into train and test sets

```
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size = 0.2, random_state = 0)
print(x_train)
print(x_test),
print(y_train)
print(y_test)
```

```
[[ 1.71000000e+02  4.54643174e+01 -7.52088358e+01 ...  8.00000000e+01
   1.00000000e+00  1.68928000e+05]
 [ 2.33500000e+03  4.55283185e+01 -7.51242916e+01 ...  5.50000000e+01
   0.00000000e+00  2.72162000e+05]
 [ 1.25700000e+03  4.54930959e+01 -7.52199227e+01 ...  5.70000000e+01
   0.00000000e+00  2.15847000e+05]
 ...
 [ 2.73800000e+03  4.54682185e+01 -7.52736848e+01 ...  6.10000000e+01
   0.00000000e+00  1.68920000e+05]
 [ 2.99600000e+03  4.54448243e+01 -7.52308821e+01 ...  4.40000000e+01
   0.00000000e+00  1.31389000e+05]
 [ 7.44000000e+02  4.55425901e+01 -7.51854975e+01 ...  8.80000000e+01
   0.00000000e+00  3.00307000e+05]]
[[ 3.64600000e+03  4.55401162e+01 -7.51799010e+01 ...  6.50000000e+01
   0.00000000e+00  3.00308000e+05]
 [ 3.60200000e+03  4.55174092e+01 -7.53040321e+01 ...  6.20000000e+01
   0.00000000e+00  2.53372000e+05]
 [ 3.86000000e+03  4.55070362e+01 -7.52475136e+01 ...  7.90000000e+01
   0.00000000e+00  2.34611000e+05]
 ...
 [ 4.50000000e+02  4.55517722e+01 -7.52773530e+01 ...  4.60000000e+01
   0.00000000e+00  3.19063000e+05]
 [ 2.94700000e+03  4.55377989e+01 -7.51025611e+01 ...  7.60000000e+01
   0.00000000e+00  2.90933000e+05]
 [ 6.22000000e+02  4.54061924e+01 -7.52354614e+01 ...  5.00000000e+01
   1.00000000e+00  6.57010000e+04]]
[1 1 1 ... 1 1 1]
[1 1 1 ... 1 1 0]
```

## 2.4. Scaling:

- After we applied train test split we applied scaling using minmaxscaler to transform our data numbers to 0 & 1 to avoid our models to skewed for a particular feature and neglicate other.

### Scaling

```
[8]  sc = MinMaxScaler()
     x_train = sc.fit_transform(x_train)
     x_test  = sc.transform(x_test)
```

## 2.5. Modeling:

- ✓ n_estimators: is the number of trees you want to build before taking the maximum voting or averages of predictions.
- ✓ random_state: is used to set the seed for the random generator so that we can ensure that the results that we get can be reproduced. Because of the nature of splitting the data in train and test is randomized you would get different data assigned to the train and test data unless you can control for the random factor.
- Firstly, we built function which contains all classic machine learning model (for generalization) and gave 3 parameters to it (x_train, y_train, x_test) to be able to all models to make fit and predict.
- Secondly, we use the Random Forest Classifier and make train and predict and we got new variable (y_pred_RF) from predicting.
- Thirdly, we use the Naïve Bayes Classifier and make train and predict and we got new variable (y_pred_NB) from predicting.
- Forthly, we use the Adaboost Classifier and make train and predict and we got new variable (y_pred_Adaboost) from predicting.

### Modeling

```
[9]  # Building function contains all classic machine learning model
     def model(x_train,y_train,x_test):
       #Random Forest
       RF = RandomForestClassifier(n_estimators=100,random_state =0)
       RF.fit(x_train,y_train)
       y_pred_RF = RF.predict(x_test)

       # Naive bayes
       NB = GaussianNB()
       NB.fit(x_train,y_train)
       y_pred_NB = NB.predict(x_test)

       #Adaboost
       Adaboost = AdaBoostClassifier(n_estimators = 100, random_state = 0)
       Adaboost.fit(x_train,y_train)
       y_pred_Adaboost = Adaboost.predict(x_test)

       return y_pred_RF, y_pred_NB, y_pred_Adaboost,RF,NB,Adaboost
```

```
[10] y_pred_RF, y_pred_NB,y_pred_Adaboost,RF,NB,Adaboost = model(x_train,y_train,x_test)
```

## 2.6. Evaluation:

- Firstly, we built function which contains evaluation techniques which we used in this project (for generalization).

Evaluation

```
[11] # Building function contain all evaluation techniques we used in this project
     def accuracy (y_test,y_pred):
       cr = classification_report(y_test,y_pred)
       acc = accuracy_score(y_test,y_pred)
       print(cr)
       return acc
```

### 2.6.1. Random Forest (RF) Classifier:

- Secondly, we calculated the accuracy of the Random Forest Classifier.
- Accuracy: **99.6%**

1. Random Forest

```
[12] accuracy(y_test,y_pred_RF)

              precision    recall  f1-score   support

           0       1.00      0.98      0.99       390
           1       1.00      1.00      1.00      2507

    accuracy                           1.00      2897
   macro avg       1.00      0.99      0.99      2897
weighted avg       1.00      1.00      1.00      2897

0.9968933379357956
```

### 2.6.2. Naïve Bayes (NB) Classifier:

- Thirdly, we calculated the accuracy of the Naïve Bayes Classifier.
- Accuracy: **84.9%**

2. Naive Bayes

```
[13] accuracy(y_test,y_pred_NB)

              precision    recall  f1-score   support

           0       0.44      0.40      0.42       390
           1       0.91      0.92      0.91      2507

    accuracy                           0.85      2897
   macro avg       0.67      0.66      0.66      2897
weighted avg       0.84      0.85      0.85      2897

0.8498446668967898
```

### 2.6.3. Adaboost Classifier:

- Forthly, we calculated the accuracy of the Adaboost Classifier.
- Accuracy: **96.7%**

3. Adaboost

```
[14] accuracy(y_test,y_pred_Adaboost)

              precision    recall  f1-score   support

           0       0.92      0.84      0.88       390
           1       0.98      0.99      0.98      2507

    accuracy                           0.97      2897
   macro avg       0.95      0.91      0.93      2897
weighted avg       0.97      0.97      0.97      2897

0.9678978253365551
```

## 2.7. Final decision method (1):

- Here we applied the 3 models in data frame to apply sum method to check if the output greater than 2, we will put it as legitimate or one otherwise we will put it as fake or zero.

Final decision method 1

```
[15] all_models = {'RF':y_pred_RF,'NB':y_pred_NB,'Adaboost':y_pred_Adaboost}
     df = pd.DataFrame(all_models)
     Aggregator = df.sum(axis=1)
     final_decision = []
     for i in Aggregator:
       if i >= 2:
         final_decision.append(1)
       else:
         final_decision.append(0)
     final_decision
```

```
 1,
 1,
 1,
 1,
 1,
 1,
 1,
 1,
 1,
 0,
 1,
 1,
 0,
 0,
 1,
 1,
```

### 2.8. Final decision method (2):

- Firstly, we used the majority voting-based aggregator to make final decision for each task.
- Secondly, we used the weighted sum aggregation to make final decision for each task.

Final decision method 2

```
y_RF = RF.predict(x_train)
y_NB = NB.predict(x_train)
y_Adaboost = Adaboost.predict(x_train)
X = accuracy(y_train,y_RF)
Y = accuracy(y_train,y_Adaboost)
Z = accuracy(y_train,y_NB)
W_RF = X/(X+Y+Z)
W_Adaboost = Y/(X+Y+Z)
W_NB = Z/(X+Y+Z)
Aggregated_output = y_pred_RF * W_RF + y_pred_NB * W_NB + y_pred_Adaboost * W_Adaboost
print(Aggregated_output)
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1507
           1       1.00      1.00      1.00     10080

    accuracy                           1.00     11587
   macro avg       1.00      1.00      1.00     11587
weighted avg       1.00      1.00      1.00     11587

              precision    recall  f1-score   support

           0       0.92      0.86      0.89      1507
           1       0.98      0.99      0.98     10080

    accuracy                           0.97     11587
   macro avg       0.95      0.93      0.94     11587
weighted avg       0.97      0.97      0.97     11587

              precision    recall  f1-score   support

           0       0.45      0.42      0.44      1507
           1       0.91      0.92      0.92     10080

    accuracy                           0.86     11587
   macro avg       0.68      0.67      0.68     11587
weighted avg       0.85      0.86      0.86     11587

[1.         1.         1.         ... 1.         1.         0.30314673]
```

```
final_decision_2 = []
for i in Aggregated_output:
  if i > 0.5 :
    final_decision_2.append(1)
  else:
    final_decision_2.append(0)
```

```
[18] final_decision_2
```

```
1,
1,
1,
0,
1,
1,
1,
```

### 2.9. Comparison:

- Here we compare between all generated accuracy.

Comparision

```
[19] Ensemble_vote = accuracy(y_test,final_decision)
     Ensemble_wieghted = accuracy(y_test,final_decision_2)
     acc_RF = accuracy(y_test,y_pred_RF)
     acc_NB = accuracy(y_test,y_pred_NB)
     acc_Adaboost = accuracy(y_test,y_pred_Adaboost)
```

```
              precision    recall  f1-score   support

           0       0.97      0.83      0.90       390
           1       0.97      1.00      0.99      2507

    accuracy                           0.97      2897
   macro avg       0.97      0.91      0.94      2897
weighted avg       0.97      0.97      0.97      2897
              precision    recall  f1-score   support

           0       0.97      0.83      0.90       390
           1       0.97      1.00      0.99      2507

    accuracy                           0.97      2897
   macro avg       0.97      0.91      0.94      2897
weighted avg       0.97      0.97      0.97      2897
              precision    recall  f1-score   support

           0       1.00      0.98      0.99       390
           1       1.00      1.00      1.00      2507

    accuracy                           1.00      2897
   macro avg       1.00      0.99      0.99      2897
weighted avg       1.00      1.00      1.00      2897
              precision    recall  f1-score   support

           0       0.44      0.40      0.42       390
           1       0.91      0.92      0.91      2507

    accuracy                           0.85      2897
   macro avg       0.67      0.66      0.66      2897
weighted avg       0.84      0.85      0.85      2897
              precision    recall  f1-score   support

           0       0.92      0.84      0.88       390
           1       0.98      0.99      0.98      2507

    accuracy                           0.97      2897
   macro avg       0.95      0.91      0.93      2897
weighted avg       0.97      0.97      0.97      2897
```

## 2.10. Visualization:

- ✓ import seaborn as sns: is an open-source Python library built on top of matplotlib. It is used for data visualization and exploratory data analysis. Seaborn works easily with data frames and the Pandas library. The graphs created can also be customized easily.
- Here we can see the difference between all the accuracy, the Random Forest (RF) has the largest accuracy. Because the Random Forest makes voting between number of random decision trees then take the highest one, and when it repeats this iteration, it won't take the same decision trees


## 2.11. CGAN:

- ✓ CGAN: the conditional generative adversarial network, or cGAN for short, is a type of GAN that involves the conditional generation of images by a generator model. Image generation can be conditional on a class label, if available, allowing the targeted generated of images of a given type.
- ✓ Batch size: defines number of samples that going to be propagated through the network.
- ✓ latent_dim: Latent dimensions/latent variables are variables which we do not directly observe, but which we assume to exist (in at least some instrumental sense) in order to explain patterns of variation in observed or manifest variables.
- ✓ Generator is used to generate fake tasks and Discriminator is used to distinguish between fake and legitimate tasks and two networks are works

against each other generators trained to fool the discriminator, and discriminator works to find out the fake task from legitimate task.

- Firstly, we defined many variables and gave to them fixed values as batch_size, num_channels, num_classes and latent_dim .
- Secondly, we perpared new training data by using keras.utils.to_categorical to converts a class vector (integers) to binary class matrix.
- Thirdly, we used from_tensor_slices to make a dataset where each input tensor is column of your data; so all tensors must be the same length, and the elements (rows) of the resulting dataset are tuples with one element from each column.
- Fourthly, we used from_tensor_slices to make a dataset where each input tensor is column of your data; so all tensors must be the same length, and the elements (rows) of the resulting dataset are tuples with one element from each column.
- Fifthly, we used .suffle() method to reorganize the order of the items. It changed the original list beacuse it didn't return a new list.
- Sixly, we printed the shape of training images and shape of training labels.
- Seventhly, we defined the generator_in_channels as the summation of latent_dim and num_classes, and the discriminator_in_channels as the summation of num_channels and num_classes.

GAN

Constant variables

```
[21] batch_size = 64
     num_channels = 12
     num_classes = 2
     latent_dim = 128
```

Prepare the training data

```
[22] y_train_new = keras.utils.to_categorical(y_train, 2)
     dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train_new))
     dataset = dataset.shuffle(buffer_size=1024).batch(batch_size)

     print(f"Shape of training images: {x_train.shape}")
     print(f"Shape of training labels: {y_train_new.shape}")

     Shape of training images: (11587, 12)
     Shape of training labels: (11587, 2)

[23] generator_in_channels = latent_dim + num_classes
     discriminator_in_channels = num_channels + num_classes
     print(generator_in_channels, discriminator_in_channels)

     130 14
```

### 2.11.1. Building Generator and Discriminator models:

- Firstly, we created the discriminator to distinguish between fake and legitimate tasks.
- Secondly, we crreated the generator to generate fake tasks.

Building Generator and Discriminator models

```
[24] # Create the discriminator.
     discriminator = Sequential(
         [
             Dense(512,input_dim = discriminator_in_channels),
             LeakyReLU(alpha=0.2),
             Dense(512),
             LeakyReLU(alpha=0.2),
             Dropout(0.4),
             Dense(512),
             LeakyReLU(alpha=0.2),
             Dropout(0.4),
             Dense(1, activation='sigmoid')
         ],
         name="discriminator"
     )

     # Create the generator.
     generator = Sequential(
         [
             Dense(256,input_dim = generator_in_channels),
             LeakyReLU(alpha=0.2),
             BatchNormalization(momentum=0.8),
             Dense(512),
             LeakyReLU(alpha=0.2),
             BatchNormalization(momentum=0.8),
             Dense(1024),
             LeakyReLU(alpha=0.2),
             BatchNormalization(momentum=0.8),
             Dense(12)],
         name="generator"
     )
```

### 2.11.2. Building CGAN:

- Firstly, we created class called ConditionalGAN.
- Secondly, we created many functions for generlization.
- Thirly, we unpacked the data.
- Fourthly, we added dummy dimensions to the labels so that they can be concatenated with the tasks. This is for the discriminator.
- Fifthly, we sampled random points in the latent space and concatenate the labels. This is for the generator.

- Sixthly, we assemble labels that say "all real images".
- Seventhly, we trained the generator.
  - Note: We should *not* update the weights of the discriminator)!
- Eightly, monitor the loss.

Conditional GAN

```
[25] class ConditionalGAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super(ConditionalGAN, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.gen_loss_tracker = keras.metrics.Mean(name="generator_loss")
        self.disc_loss_tracker = keras.metrics.Mean(name="discriminator_loss")

    @property
    def metrics(self):
        return [self.gen_loss_tracker, self.disc_loss_tracker]

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super(ConditionalGAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn

    def train_step(self, data):
        # Unpack the data.
        real_tasks, one_hot_labels = data

        # Add dummy dimensions to the labels so that they can be concatenated with the tasks. This is for the discriminator.
        task_one_hot_labels = one_hot_labels[:, :, None, None]
        task_one_hot_labels = tf.reshape(task_one_hot_labels, (-1, num_classes))


        # Sample random points in the latent space and concatenate the labels. This is for the generator.
        batch_size = tf.shape(real_tasks)[0]
        random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
        random_vector_labels = tf.concat([random_latent_vectors, one_hot_labels], axis=1)
```

```
        # Decode the noise (guided by labels) to fake tasks.
        generated_tasks = self.generator(random_vector_labels)

        # Combine them with real tasks. Note that we are concatenating the labels with these tasks here.
        real_tasks = tf.cast(real_tasks, tf.float32)
        fake_task_and_labels = tf.concat([generated_tasks, task_one_hot_labels], -1)
        real_task_and_labels = tf.concat([real_tasks, task_one_hot_labels], -1)
        combined_tasks = tf.concat([fake_task_and_labels, real_task_and_labels], axis=0)

        # Assemble labels discriminating real from fake tasks.
        labels = tf.concat([tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))], axis=0)

        # Train the discriminator.
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_tasks)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(zip(grads, self.discriminator.trainable_weights))

        # Sample random points in the latent space.
        random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
        random_vector_labels = tf.concat([random_latent_vectors, one_hot_labels], axis=1)

        # Assemble labels that say "all real images".
        misleading_labels = tf.zeros((batch_size, 1))

        # Train the generator (note that we should *not* update the weights of the discriminator)!
        with tf.GradientTape() as tape:
            fake_tasks = self.generator(random_vector_labels)
            fake_task_and_labels = tf.concat([fake_tasks, task_one_hot_labels], -1)
            predictions = self.discriminator(fake_task_and_labels)
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))
```

```python
        # Monitor loss.
        self.gen_loss_tracker.update_state(g_loss)
        self.disc_loss_tracker.update_state(d_loss)
        return {
            "g_loss": self.gen_loss_tracker.result(),
            "d_loss": self.disc_loss_tracker.result(),
        }
```

- Finally, we fitted the model and calculated the loss to the generator and discriminator.
- The generator loss: **0.8776**
- The discriminator loss: **0.6778**

```
[26] cond_gan = ConditionalGAN(
         discriminator=discriminator, generator=generator, latent_dim=latent_dim
     )
     cond_gan.compile(
         d_optimizer=keras.optimizers.Adam(learning_rate=0.0003),
         g_optimizer=keras.optimizers.Adam(learning_rate=0.0003),
         loss_fn=keras.losses.BinaryCrossentropy(),
     )

     cond_gan.fit(dataset, epochs=20)

     Epoch 1/20
     182/182 [==============================] - 9s 36ms/step - g_loss: 1.3895 - d_loss: 0.6644
     Epoch 2/20
     182/182 [==============================] - 7s 37ms/step - g_loss: 0.7756 - d_loss: 0.7241
     Epoch 3/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.7446 - d_loss: 0.6976
     Epoch 4/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.6853 - d_loss: 0.7063
     Epoch 5/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.6917 - d_loss: 0.7122
     Epoch 6/20
     182/182 [==============================] - 7s 37ms/step - g_loss: 0.7326 - d_loss: 0.7177
     Epoch 7/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.7273 - d_loss: 0.6952
     Epoch 8/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.7183 - d_loss: 0.7016
     Epoch 9/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.8398 - d_loss: 0.7296
     Epoch 10/20
     182/182 [==============================] - 7s 37ms/step - g_loss: 0.7943 - d_loss: 0.6948
     Epoch 11/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.6353 - d_loss: 0.7087
     Epoch 12/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.7537 - d_loss: 0.6882
     Epoch 13/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.7285 - d_loss: 0.7132
     Epoch 14/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.7900 - d_loss: 0.6864
     Epoch 15/20
     182/182 [==============================] - 9s 48ms/step - g_loss: 0.7339 - d_loss: 0.7005
     Epoch 16/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.7218 - d_loss: 0.7207
     Epoch 17/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.7126 - d_loss: 0.6970
     Epoch 18/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.7311 - d_loss: 0.7094
     Epoch 19/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.7284 - d_loss: 0.6886
     Epoch 20/20
     182/182 [==============================] - 7s 36ms/step - g_loss: 0.8776 - d_loss: 0.6778
     <keras.callbacks.History at 0x7f4367e13610>
```

### 2.11.3. Generate fake task via generator network:

- Firstly, we extracted the generator network from our conditional GAN class.

- Secondly, we decided to generate 2000 fake tasks using the generator networks.
- Thirly, we created the noise signal from which the generator network is used to generate a fake task. We make our latent space dimension 128. Then, we repeated the noise to be 2000 times to generate 2000 fake tasks. After that, we reshape our noise dimension.
- Fourthly, we assigned the label to the fake task equal to 1.
- Fifthly, we passed it to the generator network the noise and label it to begin to generate 2000 fake tasks.

Generate fake task via generator network

```
[ ] # We first extract the trained generator from our Conditiona GAN.
    trained_gen = cond_gan.generator

    # Choose the number of intermediate tasks that would be generated
    num_interpolation = 2000  # @param {type:"integer"}

    # Sample noise for the interpolation.
    interpolation_noise = tf.random.normal(shape=(1, latent_dim))
    interpolation_noise = tf.repeat(interpolation_noise, repeats=num_interpolation)
    interpolation_noise = tf.reshape(interpolation_noise, (num_interpolation, latent_dim))


    def interpolate_class(class1):
        # Convert the start label to one-hot encoded vectors.
        first_label = keras.utils.to_categorical([class1]*num_interpolation, num_classes)

        # Combine the noise and the labels and run inference with the generator.
        noise_and_labels = tf.concat([interpolation_noise, first_label], 1)
        fake = trained_gen.predict(noise_and_labels)
        return fake


    class1 = 1

    fake_tasks = interpolate_class(class1)
```

num_interpolation: 2000

### 2.11.4. Mix fake task with the original test dataset:

- Here we used .concatenate() method to concatenate x_test and fake_tasks.

Mix fake task with the original test dataset

```
[28] x_test_new = np.concatenate((x_test,fake_tasks),axis = 0)
     x_test_new

     array([[ 0.91147787,  0.79765584,  0.62859103, ...,  0.5       ,
              0.        ,  0.79999094],
            [ 0.90047512,  0.6936453 ,  0.12188244, ...,  0.45714286,
              0.        ,  0.67495598],
            [ 0.96499125,  0.64613102,  0.35259348, ...,  0.7       ,
              0.        ,  0.62497769],
            ...,
            [ 1.19679046,  0.72414821,  0.78746676, ..., -0.57519746,
              0.47345996,  0.52010113],
            [ 1.19679046,  0.72414821,  0.78746676, ..., -0.57519746,
              0.47345996,  0.52010113],
            [ 1.19679046,  0.72414821,  0.78746676, ..., -0.57519746,
              0.47345996,  0.52010113]])

[29] fake_task_label = np.zeros(num_interpolation)
     y_test_new = np.concatenate((y_test,fake_task_label),axis = 0)
     y_test_new

     array([1., 1., 1., ..., 0., 0., 0.])
```

#### 2.11.5. Train the machine learning models with new test set

- Firstly, we trained the previous all ML models with the new test set.

Train the machine learning models with new test set

```
[30] y_pred_RF_new, y_pred_NB_new, y_pred_Adaboost_new,RF_new,NB_new,Adaboost_new = model(x_train,y_train,x_test_new)
```

##### 2.11.5.1. Random Forest (RF) Classifer:

- Secondly, we calculated the accuracy of the Random Forest Classifier.
- Accuracy: **59%**

```
[31] acc_RF_new = accuracy(y_test_new,y_pred_RF_new)

              precision    recall  f1-score   support

         0.0       1.00      0.16      0.28      2390
         1.0       0.56      1.00      0.71      2507

    accuracy                           0.59      4897
   macro avg       0.78      0.58      0.49      4897
weighted avg       0.77      0.59      0.50      4897
```

##### 2.11.5.2. Adaboost Classifer:

- Thirdly, we calculated the accuracy of the Adaboost Classifier.
- Accuracy: **62%**

```
[32] acc_Adaboost_new = accuracy(y_test_new,y_pred_Adaboost_new)

              precision    recall  f1-score   support

         0.0       0.95      0.23      0.37      2390
         1.0       0.57      0.99      0.73      2507

    accuracy                           0.62      4897
   macro avg       0.76      0.61      0.55      4897
weighted avg       0.76      0.62      0.55      4897
```

### 2.11.5.3. Naïve Bayes (NB) Classifer:

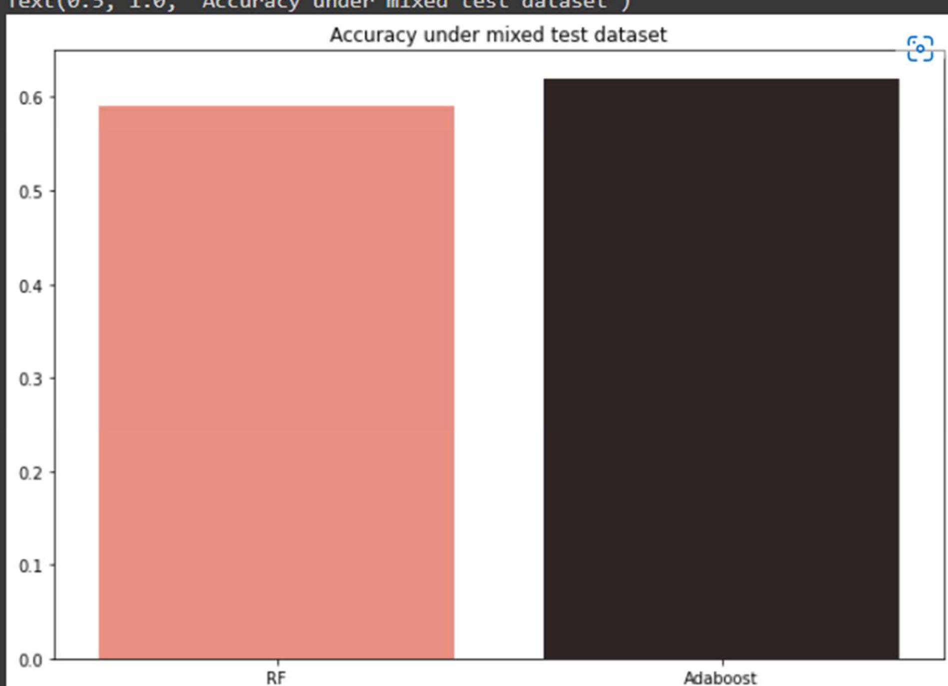
- Fourthly, we calculated the accuracy of the Naïve Bayes Classifier.
- Accuracy: **50%**

```
[33] acc_NB_new = accuracy(y_test_new,y_pred_NB_new)

              precision    recall  f1-score   support

         0.0       0.44      0.07      0.12      2390
         1.0       0.51      0.92      0.65      2507

    accuracy                           0.50      4897
   macro avg       0.48      0.49      0.39      4897
weighted avg       0.48      0.50      0.39      4897
```

- Fifthly, we plotted the highest 2 accuracies which were Random Forest and Adaboost .

```
figsize(10,7)
sns.barplot(x=['RF','Adaboost'], y = [acc_RF_new,acc_Adaboost_new],palette='dark:salmon_r')
plt.title("Accuracy under mixed test dataset")
```

```
Text(0.5, 1.0, 'Accuracy under mixed test dataset')
```



### 2.11.6. Cascade framework:

- Firstly, np.round is used to approximate the numbers  from 0.99 to 1 and 0.2 to 0 and so on.
- Secondly, we used np.where to find the index which its label is 1.
- Thirly, we updata our test data to be ounly the taks that its label is our to feed it to our models.
- Fourthly, we compute the Evaluation.

Cascade framework

```
[35] y = keras.utils.to_categorical(y_test_new, 2)
     data = tf.concat([x_test_new,y],axis=1)
     new_prediction = cond_gan.discriminator.predict(data)
     new_prediction

     array([[0.5358621 ],
            [0.50779533],
            [0.52528095],
            ...,
            [0.79594505],
            [0.79594505],
            [0.79594505]], dtype=float32)

[36] new_prediction = np.round(new_prediction).astype(int)
     new_prediction

     array([[1],
            [1],
            [1],
            ...,
            [1],
            [1],
            [1]])
```

- Fifthly, we applied it to all ML classifiers.

```
[37] idx = np.where(new_prediction == 1)[0]
     x_real_cascade = x_test_new[idx]
     y_real_discriminator = np.ones(x_real_cascade.shape[0])

[38] y_pred_RF_cascade, y_pred_NB_cascade, y_pred_Adaboost_cascade,RF_cascade,NB_cascade,Adaboost_cascade =  model(x_train,y_train,x_real_cascade)
```

### 2.11.6.1. Random Forest (RF) Classifer:

- Sixtly, we calculated the accuracy of the Random Forest Classifier.
- Accuracy: **95%**

```
[39] acc_RF_cascade = accuracy(y_real_discriminator,y_pred_RF_cascade)

              precision    recall  f1-score   support

         0.0       0.00      0.00      0.00         0
         1.0       1.00      0.95      0.98      3378

    accuracy                           0.95      3378
   macro avg       0.50      0.48      0.49      3378
weighted avg       1.00      0.95      0.98      3378
```

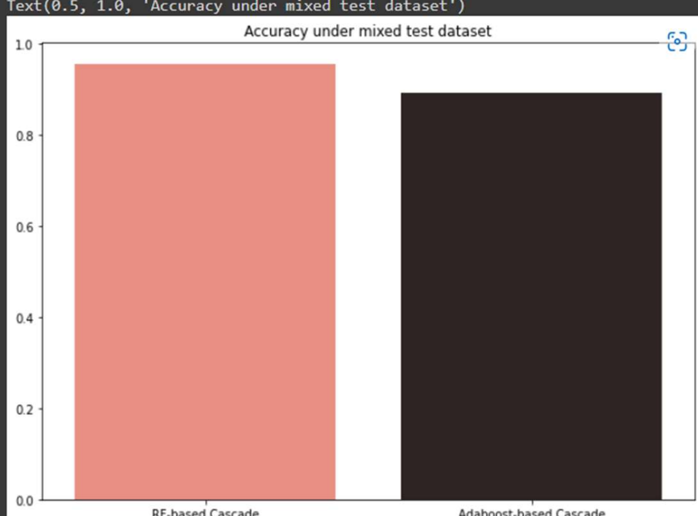
### 2.11.6.2. Adaboost Classifer:

- Seventhly, we calculated the accuracy of the Adaboost Classifier.
- Accuracy: **89%**

```
acc_Adaboost_cascade = accuracy(y_real_discriminator,y_pred_Adaboost_cascade)

              precision    recall  f1-score   support

         0.0       0.00      0.00      0.00         0
         1.0       1.00      0.89      0.94      3378

    accuracy                           0.89      3378
   macro avg       0.50      0.45      0.47      3378
weighted avg       1.00      0.89      0.94      3378
```

- Eightly, The bar chart of RF and Adaboost.

```
figsize(10,7)
sns.barplot(x=['RF-based Cascade','Adaboost-based Cascade'], y = [acc_RF_cascade,acc_Adaboost_cascade],palette='dark:salmon_r')
plt.title("Accuracy under mixed test dataset")

Text(0.5, 1.0, 'Accuracy under mixed test dataset')
```



## 3. Conclusion

In this project, we used three different stages to classify legitimate and fake tasks. First, we admit classical machine learning algorithms such as random forest, Adaboost, naïve based, ensemble vote, and ensemble weighted which are 97% for ensemble weighted, ensemble vote, Adaboost, 100% for the random forest, and finally 85% for naïve based classifier. The second technique was to generate 2000 fake tasks using a generator network and then we applied random forest, Adaboost, and naïve-based algorithms we notice that the accuracy is very low which are 59%, 57%, and 50% respectively because the model classifies data as the first time, we learn on it. The third technique used the discriminator network to distinguish between legitimate and fake tasks then we applied the machine learning algorithms we notice the accuracy increased hugely which are 92% for the random forest, and 93% for the Adaboost classifier.