National College of Ireland

Configuration Manual

MSc Research Project
MSc in Data Analytics

Debayan Biswas
Student ID: x22242821

School of Computing
National College of Ireland

Supervisor: Dr. Bharat Agarwal

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Debayan Biswas
Student ID:	x22242821
Programme:	MSc in Data Analytics
Year:	2024
Module:	MSc Research Project
Supervisor:	Dr. Bharat Agarwal
Submission Due Date:	12/08/2024
Project Title:	Configuration Manual
Word Count:	734
Page Count:	16

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Debayan Biswas
Date:	10th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Debayan Biswas
x22242821

1 Introduction

This configuration manual provides a comprehensive guide about the setup steps and techniques that has been implemented along with software and hardware configuration, source of dataset and the complete process of project implementation. Techniques of Conditional generative adversarial network (cGAN) to generate synthetic data for class imbalance handling and the implementation of a robust XGBoost classifier has been discussed in this manual.

2 System Configuration

Software Components	**Version**
Operating System	Windows 11 23H2
Python	3.11.5
Jupyter Notebook	6.5.4
TensorFlow	2.15.0

Table 1: Software Configuration

Hardware Components	**Specification**
Processor	Ryzen 5 5625U
RAM	16 GB DDR4
GPU	Radeon Vega 7

Table 2: Hardware Configuration

3 Dataset Source

The datasets used in this project are sourced from the Transiting Exoplanet Survey Satellite (TESS) and Planetary Habitability Laboratory (PHL) websites.

3.1 TESS

TESS is a National Aeronautics and Space Administration (NASA) funded mission maintained by the Mikulski Archive for Space Telescopes (MAST) designed to discover exoplanets. The data is publicly available without licensing restrictions and can be accessed through their website as shown in Figure 1.

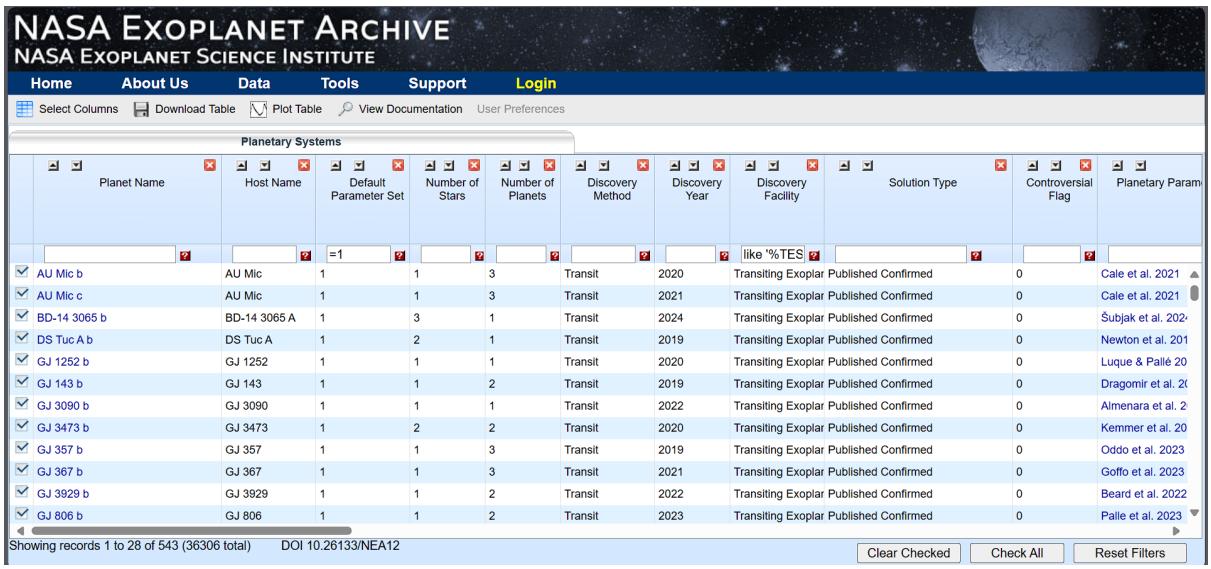

Figure 1: TESS Website

3.2 PHL

The Planetary Habitability Laboratory (PHL) dataset is maintained by the University of Puerto Rico at Arecibo. It provides a comprehensive database of habitable exoplanets. The data is publicly available with a Creative Commons license and can be accessed through their website as shown in Figure 2.

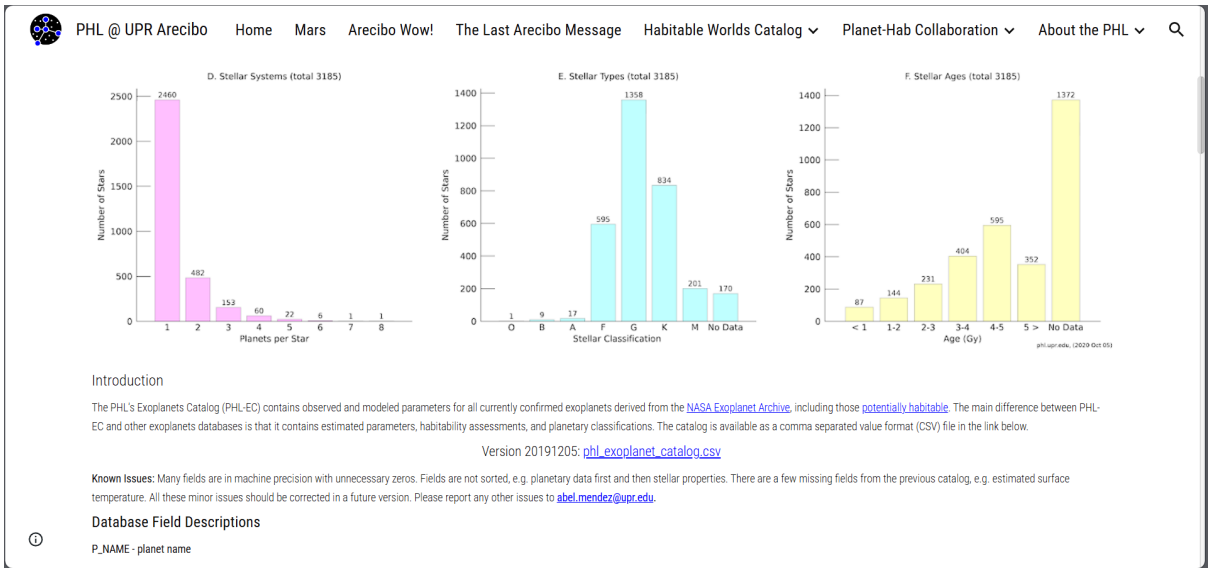

Figure 2: PHL Website

4 Project Implementation

This section provides a step-by-step guide to the project implementation, illustrated with code screenshots.

4.1 Importing Packages and Libraries

The required packages and libraries are imported in Jupyter Notebook as shown in Figure 3. The project used 'pandas' and 'numpy' for data manipulation and data processing. Matplotlib, seaborn and missingno used for statistical and interactive data visualization. Scikit-Learn used for data preprocessing and model evaluation. The tensorflow library used for creating, training and optimization of the generative model. Xgboost used for implementing the classifier.

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
%matplotlib inline

import missingno as msno

from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn import preprocessing
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split, cross_validate, GridSearchCV
from xgboost import XGBClassifier,plot_importance
from sklearn.utils import class_weight

import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras.constraints import Constraint

from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score, classification_report, f1_score
from sklearn.metrics import confusion_matrix, roc_auc_score, log_loss
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import shap
from sklearn.metrics import roc_curve, auc
```

Figure 3: Packages and libraries

4.2 Data Loading and Merging

The downloaded datasets TESS and PHL are loaded into the dataframe as shown in Figure 4.

```
tess = pd.read_csv("tess.csv")
phl = pd.read_csv("phl.csv")
```

Storing in Dataframe

```
tess = pd.DataFrame(tess)
phl = pd.DataFrame(phl)
```

Figure 4: Data Loading

The TESS table is prepared for merging, as seen in Figure 5 with common and additional columns.

Renaming necessary TESS columns

```
column_map = {
    'pl_name': 'P_NAME',
    'hostname': 'S_NAME',
    'pl_orbper': 'P_PERIOD',
    'pl_rade': 'P_RADIUS',
    'pl_bmasse': 'P_MASS',
    'pl_orbeccen': 'P_ECCENTRICITY',
    'st_teff': 'S_TEMPERATURE',
    'st_rad': 'S_RADIUS',
    'st_mass': 'S_MASS',
    'st_met': 'S_METALLICITY',
    'sy_dist': 'S_DISTANCE',
    'dec': 'S_DEC'
}
```

```
extra_col = [
    'P_ESI', 'P_TYPE_TEMP', 'P_TEMP_SURF_MAX', 'P_HABITABLE'
]
```

```
tess_renamed = tess.rename(columns=column_map)
```

Figure 5: Preparing TESS

The PHL table is prepared for merging, as shown in Figure 6.

Preparing PHL for merging

```
common_col = list(column_map.values())
common_col
```

```
['P_NAME',
 'S_NAME',
 'P_PERIOD',
 'P_RADIUS',
 'P_MASS',
 'P_ECCENTRICITY',
 'S_TEMPERATURE',
 'S_RADIUS',
 'S_MASS',
 'S_METALLICITY',
 'S_DISTANCE',
 'S_DEC']
```

```
common_col = [
    'P_NAME',
    'S_NAME',
    'P_PERIOD',
    'P_RADIUS',
    'P_MASS',
    'P_ECCENTRICITY',
    'S_TEMPERATURE',
    'S_RADIUS',
    'S_MASS',
    'S_METALLICITY',
    'S_DISTANCE',
    'S_DEC'
]

phl_only_col = [
    'P_ESI',
    'P_TYPE_TEMP',
    'P_TEMP_SURF_MAX',
    'P_HABITABLE'
]
```

```
phl_filtered = phl[common_col + extra_col]
```

Figure 6: Preparing PHL

The prepared TESS and PHL data are merged to create a combined dataset, as seen in Figure 7.

Loading files from system

```
tess_updated_df = pd.read_csv('tess_updated.csv')
phl_updated_df = pd.read_csv('phl_updated.csv')
```

Selecting common and specific columns in TESS and PHL

```
phl_selected_adjusted = phl_updated_df[common_col + phl_only_col]
tess_selected_adjusted = tess_updated_df[common_col]
```

Merging two updated TESS and PHL tables

```
merged_df = pd.merge(phl_selected_adjusted, tess_selected_adjusted, on='P_NAME', how='outer', suffixes=('_phl', '_tess'))


for column in common_col:
    if f"{column}_phl" in merged_df.columns and f"{column}_tess" in merged_df.columns:
        merged_df[column] = merged_df[f"{column}_phl"].combine_first(merged_df[f"{column}_tess"])


col_drop = [f"{col}_phl" for col in common_col] + [f"{col}_tess" for col in common_col]
merged_df = merged_df.drop(columns=col_drop, errors='ignore')


pname_count = merged_df['P_NAME'].nunique()
print("Total number of rows in the merged table",pname_count)
```

```
Total number of rows in the merged table 5391
```

Figure 7: Data Merging

4.3 Data Preprocessing

Handling of missing values in categorical and numerical columns are illustrated in Figure 8. Numerical columns are handled using K-Nearest Neighbors (KNN) Imputer and the categorical columns are handled using Simple Imputer.

Handling Numeric and Categorical Columns

```
df = copy_df

# Separating the features and label
X = df.drop(columns=['P_HABITABLE'])
y = df['P_HABITABLE']

# Identifying the categorical and numeric columns
categorical_cols = X.select_dtypes(include=['object']).columns.tolist()
numeric_cols = X.select_dtypes(exclude=['object']).columns.tolist()

# using preprocessors for numeric and categorical columns
numeric_preprocessor = Pipeline(steps=[
    ('imputer', KNNImputer(n_neighbors=5))
])

categorical_preprocessor = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent'))
])

# Using Column Transformer to pass the processors
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_preprocessor, numeric_cols),
        ('cat', categorical_preprocessor, categorical_cols)
    ])

# Preprocessing the features
X_preprocessed = preprocessor.fit_transform(X)
X_preprocessed_df = pd.DataFrame(X_preprocessed, columns=numeric_cols + categorical_cols, index=X.index)

# Merging the processed features with the label
processed_df = pd.concat([X_preprocessed_df, y], axis=1)

print("Dataset after preprocessing:\n",processed_df.head())
```

Figure 8: Handling Numeric and Categorical Columns

Handling missing values in the P_HABITABLE column is shown in Figure 9. The missing values are handled using a Random Forest classifier.

Handling P_HABITABLE column for missing data

```
# Separating the known and unknown values in P_HABITABLE
known_mask = processed_df['P_HABITABLE'].notna()
unknown_mask = processed_df['P_HABITABLE'].isna()

X_known = processed_df.loc[known_mask].drop(columns=['P_HABITABLE'])
y_known = processed_df.loc[known_mask, 'P_HABITABLE']
X_unknown = processed_df.loc[unknown_mask].drop(columns=['P_HABITABLE'])

# Training the model for predicting P_HABITABLE
model = RandomForestClassifier(n_estimators=100, random_state=42)

# Cross checking the categorical columns are encoded
categorical_cols_known = X_known.select_dtypes(include=['object']).columns.tolist()
X_known_encoded = pd.get_dummies(X_known, columns=categorical_cols_known, drop_first=True)
X_unknown_encoded = pd.get_dummies(X_unknown, columns=categorical_cols_known, drop_first=True)

# Fitting the model
X_unknown_encoded = X_unknown_encoded.reindex(columns=X_known_encoded.columns, fill_value=0)
model.fit(X_known_encoded, y_known)

# Predicting the blank values in 'P_HABITABLE'
predicted_habitable = model.predict(X_unknown_encoded)

# Filling the blank 'P_HABITABLE' values in the processed_df dataframe
processed_df.loc[unknown_mask, 'P_HABITABLE'] = predicted_habitable

# Final preprocessed data
final_df = processed_df.copy()
print(final_df.head())

# Checking if all the missing values are filled
if final_df['P_HABITABLE'].isna().sum() == 0:
    print("All the blank 'P_HABITABLE' values are filled")
else:
    print("Blank values still present")
```

Figure 9: Handling Missing Values in P_HABITABLE

4.4 cGAN Model

Preparing data for the cGAN model is shown in Figure 10. The unnecessary parameters are dropped and the dataset is split into features and target label. The features are scaled and the target label is encoded.

Preparing the dataset for cGAN implementation

```
# Dropping features and spliting dataframe into features and label
X = project_df.drop(columns=['P_NAME', 'S_NAME', 'P_HABITABLE'])
y = project_df['P_HABITABLE']

# Encoding the feature 'P_TYPE_TEMP'
type_encoder = LabelEncoder()
X['P_TYPE_TEMP'] = type_encoder.fit_transform(X['P_TYPE_TEMP'])

X = X.apply(pd.to_numeric, errors='coerce')

# Scaling all features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Encoding the label
y = y.astype(int)
encoder = OneHotEncoder(sparse=False)
y_encoded = encoder.fit_transform(y.values.reshape(-1, 1))

# Converting datatype for next operations
X_scaled_np = X_scaled.astype(np.float32)
y_encoded = y_encoded.astype(np.float32)

print(X_scaled_np.shape)
print(y_encoded.shape)
```

```
(5392, 13)
(5392, 3)
```

Figure 10: Preparing Data for cGAN

The cGAN model with generator and discriminator functions is shown in Figure 11. The simultaneous working of the generator and discriminator in this model generates synthetic data resembling the real data.

```
# Defining the hyperparameters after Trial and Error. This are the best possible parameters for this cGAN model
batch_size = 32
epochs = 10000
sample_interval = 100
noise_dim = 90

# Using custom Spectral normalization to stabilize the cGAN training
class SpectralNormalization(Constraint):
    def __init__(self, axis=-1):
        self.axis = axis
    def __call__(self, w):
        return w / (1e-9 + tf.reduce_max(tf.abs(w), axis=self.axis, keepdims=True))

# Generator function
def func_generator(noise_dim, class_dim, output_dim):
    noise_input = layers.Input(shape=(noise_dim,))
    class_input = layers.Input(shape=(class_dim,))
    concatenated = layers.Concatenate()([noise_input, class_input])
    x = layers.Dense(256, activation='relu')(concatenated)
    x = layers.BatchNormalization()(x)
    x = layers.Dense(512, activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Dense(256, activation='relu')(x)
    x = layers.BatchNormalization()(x)
    output = layers.Dense(output_dim, activation='sigmoid')(x)
    model = models.Model([noise_input, class_input], output)
    return model

# Discriminator function
def func_discriminator(input_dim, class_dim):
    model = models.Sequential([
        layers.Dense(256, activation='relu', input_dim=input_dim + class_dim, kernel_constraint=SpectralNormalization()),
        layers.Dropout(0.4),
        layers.Dense(128, activation='relu', kernel_constraint=SpectralNormalization()),
        layers.Dropout(0.4),
        layers.Dense(64, activation='relu', kernel_constraint=SpectralNormalization()),
        layers.Dense(1, activation='sigmoid')
    ])
    return model

# fetching shape of the scaled features and encoded label
class_dim = y_encoded.shape[1]
output_dim = X_scaled_np.shape[1]

# Building discriminator by using specific learning rate and loss function. Learning rate of 0.0002 which is farely low
# was not providing good results, so decreased the learning rate further.

discriminator = func_discriminator(output_dim, class_dim)
discriminator.compile(optimizer=optimizers.Adam(learning_rate=0.0001, beta_1=0.5), loss='binary_crossentropy',
metrics=['accuracy'])
generator = func_generator(noise_dim, class_dim, output_dim)
```

Figure 11: Generator and Discriminator

The training loop for the cGAN is illustrated in Figure 12. This training process generates synthetic data similar to real data.

```
# cGAN model
gan_input_noise = layers.Input(shape=(noise_dim,))
gan_input_class = layers.Input(shape=(class_dim,))
generated_dat = generator([gan_input_noise, gan_input_class])

discriminator.trainable = False
gan_output = discriminator(layers.Concatenate()([generated_dat, gan_input_class]))
gan = models.Model([gan_input_noise, gan_input_class], gan_output)
gan.compile(optimizer=optimizers.Adam(learning_rate=0.0002, beta_1=0.5), loss='binary_crossentropy')

# Training the cGAN
for epoch in range(epochs):
    # Selecting random batch of real data
    idx = np.random.randint(0, X_scaled_np.shape[0], batch_size)
    real_data = X_scaled_np[idx]
    real_labels = y_encoded[idx]

    # Generating synthetic data using the generator function
    noise = np.random.normal(0, 1, (batch_size, noise_dim))
    generated_dat = generator.predict([noise, real_labels])

    # Overfitting prevention and label smoothing
    real_data += 0.05 * np.random.normal(0, 1, real_data.shape)
    generated_dat += 0.05 * np.random.normal(0, 1, generated_dat.shape)

    real_labels_smoothed = 0.9 * np.ones((batch_size, 1))
    fake_labels_smoothed = np.zeros((batch_size, 1))

    # Training the discriminator function on real and synthetic data
    d_loss_real = discriminator.train_on_batch(np.hstack((real_data, real_labels)), real_labels_smoothed)
    d_loss_fake = discriminator.train_on_batch(np.hstack((generated_dat, real_labels)), fake_labels_smoothed)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # Training the generator function
    noise = np.random.normal(0, 1, (batch_size, noise_dim))
    sampled_labels = np.random.randint(0, class_dim, batch_size)
    sampled_labels_encoded = np.eye(class_dim)[sampled_labels]
    g_loss = gan.train_on_batch([noise, sampled_labels_encoded], np.ones((batch_size, 1)))


    if epoch % sample_interval == 0:
        print(f"{epoch} [D loss: {d_loss[0]}] [G loss: {g_loss}]")
```

```
9900 [D loss: 0.38553212210536003] [G loss: 1.5873527526855469]
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
```

Figure 12: cGAN Training Loop

Combining synthetic data generated for the minority classes using cGAN with real data is shown in Figure 13.

Calculation of required synthetic samples to be generated based on the count of Majority cl

```
# Count of majority class
num_class_0 = c0

# Calculating necessary minority classes required
num_class_1 = num_class_0 - c1
num_class_2 = num_class_0 - c2

print(f"Generating {num_class_1} synthetic samples for Minotrity class 1.")
print(f"Generating {num_class_2} synthetic samples for Minority class 2.")
```

```
Generating 5309 synthetic samples for Minotrity class 1.
Generating 5292 synthetic samples for Minority class 2.
```

Synthetic data generation for Minority Class 1

```
noise = np.random.normal(0, 1, (num_class_1, noise_dim))
sampled_labels_1 = np.ones(num_class_1)
sampled_labels_encoded_1 = np.eye(class_dim)[sampled_labels_1.astype(int)]
synthetic_samples_1 = generator.predict([noise, sampled_labels_encoded_1])
```

```
166/166 [==============================] - 0s 1ms/step
```

Synthetic data generation for Minority Class 2

```
noise = np.random.normal(0, 1, (num_class_2, noise_dim))
sampled_labels_2 = np.ones(num_class_2) * 2
sampled_labels_encoded_2 = np.eye(class_dim)[sampled_labels_2.astype(int)]
synthetic_samples_2 = generator.predict([noise, sampled_labels_encoded_2])
```

```
166/166 [==============================] - 0s 1ms/step
```

Combining both Original and Synthetic samples grenerated from cGAN

```
# Synthetic features and labels for Class 1
X_synth_1 = pd.DataFrame(synthetic_samples_1, columns=X.columns)
y_synth_1 = pd.Series(sampled_labels_1, name='P_HABITABLE')

# Synthetic features and labels for Class 2
X_synth_2 = pd.DataFrame(synthetic_samples_2, columns=X.columns)
y_synth_2 = pd.Series(sampled_labels_2, name='P_HABITABLE')

# Merging features for both Class 1 and 2
X_balanced = pd.concat([pd.DataFrame(X_scaled_np, columns=X.columns), X_synth_1, X_synth_2])

# Merging labels for both Class 1 and 2
y_balanced = pd.concat([pd.Series(y, name='P_HABITABLE'), y_synth_1, y_synth_2])
```

Figure 13: Combining Synthetic and Real Data

4.5 Model Training

The combined dataset is trained on a custom XGBoost classifier, as shown in Figure 14. The features and target labels are encoded prior to training. The model is trained on the dataset based on specific hyperparameters achieved using param grid on grid search hyperparameter tuning process.

Splitting the features and label into train and test sets

```
X_train, X_test, y_train, y_test = train_test_split(X_balanced, y_balanced, test_size=0.2, random_state=42)
```

Standardizing the features using Standard Scalar

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Encoding the label using Label Encoder

```
encoder = LabelEncoder()
y_train_encoded = encoder.fit_transform(y_train)
y_test_encoded = encoder.transform(y_test)
```

Fixing the parameter grids for training. These are the best found parameters after trial and error

```
param_grid = {
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
    'n_estimators': [100, 200, 300],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0]
}
```

XGBoost classifier

```
xgb_model = XGBClassifier()
```

Using Grid Search to find the best possible paramters for training the classifier

```
grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid, cv=5, scoring='accuracy', verbose=1, n_jobs=-1)
grid_search.fit(X_train_scaled, y_train_encoded)
```

```
Fitting 5 folds for each of 108 candidates, totalling 540 fits
```

GridSearchCV
estimator: XGBClassifier
XGBClassifier

Best possible paramters combination

```
best_params = grid_search.best_params_
print("The best parameters found for training: ", best_params)
best_xgb_model = grid_search.best_estimator_
```

```
The best parameters found for training:  {'colsample_bytree': 1.0, 'learning_rate': 0.1, 'max_depth': 5,
 'n_estimators': 300, 'subsample': 0.8}
```

Figure 14: Model Training

4.6 Model Evaluation

The classifier is evaluated for accuracy and other metrics, as shown in Figure 15. The quantitative metric shows accuracy, precision, recall, F1 score, log loss, AUC-ROC score and confusion matrix. These metrics are important for performance evaluation of the model, mainly for classification.

```
y_pred = best_xgb_model.predict(X_test_scaled)
```

```
y_prob = best_xgb_model.predict_proba(X_test_scaled)
```

```
accuracy = accuracy_score(y_test, y_pred)
print(f"The Final Accuracy is: {accuracy}")
```

```
The Final Accuracy is: 0.9596748984057518
```

Classification report

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00      1064
         1.0       0.95      0.93      0.94      1108
         2.0       0.93      0.95      0.94      1027

    accuracy                           0.96      3199
   macro avg       0.96      0.96      0.96      3199
weighted avg       0.96      0.96      0.96      3199
```

Metrics

```
f1 = f1_score(y_test, y_pred, average='weighted')
conf_matrix = confusion_matrix(y_test, y_pred)
auc = roc_auc_score(y_test, y_prob, multi_class='ovr')
logloss = log_loss(y_test, y_prob)
```

```
print(f"The final weighted F1 Score: {f1}")
print(f"Confusion Matrix:\n{conf_matrix}")
print(f"The AUC-ROC Score: {auc}")
print(f"The Log Loss: {logloss}")
```

```
The final weighted F1 Score: 0.9596872922672619
Confusion Matrix:
[[1064    0    0]
 [   0 1031   77]
 [   0   52  975]]
The AUC-ROC Score: 0.9957563383427092
The Log Loss: 0.10103874634047534
```

Figure 15: Quantitative Metrics

The code for the ROC-AUC curve plot of the evaluation is shown in Figure 16.

ROC plot

```
n_classes = 3
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_encoded, y_prob[:, i], pos_label=i)
    roc_auc[i] = auc(fpr[i], tpr[i])

plt.figure()
colors = ['cornflowerblue', 'red', 'yellow']
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=2, label=f'Class {i} (AUC = {roc_auc[i]:0.2f})')
    plt.fill_between(fpr[i], tpr[i], alpha=0.2, color=color)

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positives')
plt.ylabel('True Positives')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

Figure 16: ROC-AUC Curve

The classifier specific qualitative analysis of Shapley additive explanations (SHAP) and feature importance are depicted in Figure 17. These metrics interpret and evaluate the contribution of features in model prediction.

SHAP

```
explainer = shap.TreeExplainer(best_xgb_model)
shap_values = explainer.shap_values(X_test_scaled)
```

```
shap.summary_plot(shap_values, X_test, feature_names=X_test.columns)
```

Feature importance

```
feature_importances = best_xgb_model.get_booster().get_score(importance_type='gain')
importance_df = pd.DataFrame(feature_importances.items(), columns=['Feature', 'Importance'])
importance_df['Feature'] = importance_df['Feature'].apply(lambda x: X_train.columns[int(x[1:])])
importance_df = importance_df.sort_values(by='Importance', ascending=False)
```

```
print(importance_df)
```

Figure 17: SHAP Analysis and Feature Importance

4.7 Model Validation

The model's effectiveness is validated, with predictions on new data as shown in Figure 18.

Function to predict habitability

```
def predict_habitability(input_data):
    input_data_scaled = scaler.transform(input_data)
    prediction = best_xgb_model.predict(input_data_scaled)
    descriptions = [get_class_description(label) for label in prediction]
    return descriptions
```

Defining the class 0,1,2

```
def get_class_description(label):
    if label == 0:
        return "Not Habitable"
    elif label == 1:
        return "Potentially Habitable"
    elif label == 2:
        return "Confirmed Habitable"
    else:
        return "Unknown Class"
```

Exoplanet data for confirmed habitable

```
new_exoplanet_data = pd.DataFrame({
    'P_ESI': [0.80235065],
    'P_TEMP_SURF_MAX': [324.9606],
    'P_PERIOD': [17.8719],
    'P_RADIUS': [1.65908],
    'P_MASS': [3.4102945],
    'P_ECCENTRICITY': [0.11],
    'S_TEMPERATURE': [3342],
    'S_RADIUS': [0.31],
    'S_MASS': [0.29],
    'S_METALLICITY': [-0.09],
    'S_DISTANCE': [4.30592],
    'S_DEC': [-12.667687]
})
```

Prediction

```
habitability_prediction = predict_habitability(new_exoplanet_data)
print("Habitability prediction for new exoplanet data:")
for description in habitability_prediction:
    print(description)
```

Figure 18: Model Prediction