The provided Python code implements the Multi-Task Ensemble for DDoS Detection (MT-EDD) algorithm. It includes functions for data preprocessing (handling missing values, outliers, and standardization), feature selection using mutual information and *SelectKBest*, training individual classifiers (Random Forest, Decision Tree, XGBoost) and a meta-learner (MLP), predicting with the ensemble, and evaluating performance. Crucially, it demonstrates how to save the trained models, feature selector, and *scaler* using *joblib* and provides a separate predict_server.py script for deploying the model on a cloud server for real-time predictions. The code also addresses class imbalance in the training data using SMOTE and *RandomUnderSampler*. Finally, it includes instructions on setting up a cloud VM and establishing communication between the P4 switch (handling feature extraction) and the cloud-based prediction server.

Python Code Implementation for the MT-EDD Algorithm

```
// Python implementation for MT-EDD Algorithm

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.tree import DecisionTreeClassifier
import xgboost as xgb
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, specificity_score, 
matthews_corrcoef
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from sklearn.feature_selection import mutual_info_classif, SelectKBest
from sklearn.metrics import make_scorer
from sklearn.model_selection import RandomizedSearchCV

# Data Preprocessing (3.1.1)
def preprocess_data(df, p4_features=None):
    # Handle missing values (mean imputation)
    df.fillna(df.mean(numeric_only=True), inplace=True)

    # Outlier removal (example using IQR)
    for col in df.select_dtypes(include=np.number).columns:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        df = df[(df[col] >= Q1 - 1.5 * IQR) & (df[col] <= Q3 + 1.5 * IQR)]

    # One-hot encode categorical features (if any)
    df = pd.get_dummies(df, drop_first=True)

    if p4_features is not None:
        df = pd.concat([df, p4_features], axis=1)

    # Standardize numerical features
    scaler = StandardScaler()
    numerical_cols = df.select_dtypes(include=np.number).columns
    df[numerical_cols] = scaler.fit_transform(df[numerical_cols])

    return df

def select_features(X, y, k=20):  # k is the number of features to select
    mutual_info = mutual_info_classif(X, y)
    selector = SelectKBest(score_func=lambda X, y: mutual_info, k=k)
    X_selected = selector.fit_transform(X, y)
    return X_selected, selector

# Model Training (3.1.2)
def train_models(X_train, y_train, X_val, y_val):
    # Resampling
    smote = SMOTE(random_state=42)
    rus = RandomUnderSampler(random_state=42)

    # Identify classes to oversample and undersample based on counts
    class_counts = y_train.value_counts()
    minority_classes = class_counts[class_counts < class_counts.median()].index
    majority_classes = class_counts[class_counts > class_counts.median()].index

    for class_label in minority_classes:
        X_train_resampled, y_train_resampled = smote.fit_resample(X_train[y_train == class_label], y_train[y_train == 
class_label])
```

```
        X_train = pd.concat([X_train, X_train_resampled])
        y_train = pd.concat([y_train, y_train_resampled])

    for class_label in majority_classes:
        X_train_resampled, y_train_resampled = rus.fit_resample(X_train[y_train == class_label], y_train[y_train ==
class_label])
        X_train = X_train[~y_train.isin([class_label])]
        y_train = y_train[~y_train.isin([class_label])]
        X_train = pd.concat([X_train, X_train_resampled])
        y_train = pd.concat([y_train, y_train_resampled])

    # Hyperparameter tuning (RandomizedSearchCV)
    rf_param_grid = {'n_estimators': [200, 500, 800], 'max_depth': [10, 20, 30], 'min_samples_split': [2, 5, 10],
'min_samples_leaf': [1, 2, 4], 'criterion': ['gini', 'entropy']}
    dt_param_grid = {'max_depth': [None, 10, 20, 30], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4],
'criterion': ['gini', 'entropy']}
    xgb_param_grid = {'n_estimators': [100, 200, 300], 'max_depth': [3, 6, 8], 'learning_rate': [0.01, 0.1, 0.3],
'subsample': [0.7, 0.8, 0.9], 'colsample_bytree': [0.6, 0.7, 0.8]}

    rf_random = RandomizedSearchCV(RandomForestClassifier(random_state=42), rf_param_grid, n_iter=10, cv=3,
random_state=42, n_jobs=-1)
    dt_random = RandomizedSearchCV(DecisionTreeClassifier(random_state=42), dt_param_grid, n_iter=10, cv=3,
random_state=42, n_jobs=-1)
    xgb_random = RandomizedSearchCV(xgb.XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss'),
xgb_param_grid, n_iter=10, cv=3, random_state=42, n_jobs=-1)

    rf_random.fit(X_train, y_train)
    dt_random.fit(X_train, y_train)
    xgb_random.fit(X_train, y_train)

    rf = rf_random.best_estimator_
    dt = dt_random.best_estimator_
    xgb_model = xgb_random.best_estimator_

    mlp = MLPClassifier(hidden_layer_sizes=(100,), max_iter=500, random_state=42) # Example MLP
    mlp.fit(X_train, y_train)
    return rf, dt, xgb_model, mlp

def predict_ensemble(X, ensemble):
    predictions = []
    confidences = []
    for classifier in ensemble:
        pred = classifier.predict(X)
        try:
            conf = classifier.predict_proba(X)
        except AttributeError:  # DecisionTreeClassifier doesn't have predict_proba
            conf = np.zeros((len(X), 2)) # Dummy confidence as it is not used for DT in the paper
        predictions.append(pred)
        confidences.append(conf)
    return np.array(predictions).T, np.array(confidences).transpose(1,0,2)

def evaluate_model(y_true, y_pred):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, zero_division=0)
    recall = recall_score(y_true, y_pred, zero_division=0)
    f1 = f1_score(y_true, y_pred, zero_division=0)
    specificity = specificity_score(y_true, y_pred)
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
    npv = tn / (tn + fn) if (tn + fn) != 0 else 0
    fpr = fp / (fp + tn) if (fp + tn) != 0 else 0
    fdr = fp / (fp + tp) if (fp + tp) != 0 else 0
    fnr = fn / (fn + tp) if (fn + tp) != 0 else 0
    mcc = matthews_corrcoef(y_true, y_pred)
    return accuracy, precision, recall, f1, specificity, npv, fpr, fdr, fnr, mcc

# Example usage (replace with your actual data loading and paths)
try:
    df = pd.read_csv("your_cic_iot_dataset.csv")
except FileNotFoundError:
    print("Error: CICIoT2023 dataset file not found.")
    exit()

# Separate features and labels
X = df.drop("label_column_name", axis=1) # Replace "label_column_name"
y = df["label_column_name"]

X_selected, selector = select_features(X, y)

# Split data
```

```
X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25, random_state=42) # 0.25 x 0.8 = 0.2

# Train models
rf, dt, xgb_model, mlp = train_models(X_train, y_train, X_val, y_val)
ensemble = [rf, dt, xgb_model]

# Save the trained models, selector, and scaler
joblib.dump(rf, 'rf_model.pkl')
joblib.dump(dt, 'dt_model.pkl')
joblib.dump(xgb_model, 'xgb_model.pkl')
joblib.dump(mlp, 'mlp_model.pkl')
joblib.dump(selector, 'feature_selector.pkl')
# Save the scaler fitted on the training data
scaler = StandardScaler()
numerical_cols = X_train.select_dtypes(include=np.number).columns
scaler.fit(X_train[numerical_cols])
joblib.dump(scaler, 'scaler.pkl')

# Example prediction and evaluation on the test set
X_test_selected = selector.transform(X_test)
ensemble_preds, confidences = predict_ensemble(X_test_selected, ensemble)
# In real implementation you will get the weights from the adaptation module
weights = np.array([1/len(ensemble)]*len(ensemble))
weighted_pred = (confidences * weights[np.newaxis, :, np.newaxis]).sum(axis=1).argmax(axis=1) # Example weighted
prediction
accuracy, precision, recall, f1, specificity, npv, fpr, fdr, fnr, mcc = evaluate_model(y_test, weighted_pred)

print(f"MT-EDD Test Results: Accuracy={accuracy}, F1-Score={f1}, MCC={mcc}")
```

Code Explanation:

The code is divided into several key sections:

1. **Import Libraries:** Imports necessary libraries for data manipulation (*pandas*, *numpy*), machine learning (*scikit-learn*, *xgboost*), resampling (*imblearn*), model persistence (*joblib*), and evaluation metrics.
2. **Data Preprocessing (*preprocess_data* function):**
   - Handles missing values using mean imputation.
   - Removes outliers using the Interquartile Range (IQR) method.
   - Performs one-hot encoding for categorical features.
   - Concatenates P4-extracted features.
   - Standardizes numerical features using *StandardScaler* to have zero mean and unit variance. This is crucial for many ML algorithms.
3. **Feature Selection (*select_features* function):**
   - Uses Mutual Information to rank features based on their relevance to the target variable.
   - Employs *SelectKBest* to select the top *k* features (default is 20).
4. **Model Training (*train_models* function):**
   - Addresses class imbalance using SMOTE (Synthetic Minority Over-sampling Technique) for under-represented classes and RandomUnderSampler for over-represented classes.
   - Performs hyperparameter tuning for RF, DT, and XGBoost using *RandomizedSearchCV*. This helps find the best model parameters for the given dataset.
   - Trains an MLP as the meta-learner.
   - Returns the trained models.
5. **Ensemble Prediction (*predict_ensemble* function):**
   - Takes input features and the ensemble of classifiers.
   - Generates predictions and confidence scores (probabilities) for each classifier.
   - Returns the predictions and confidences in a structured format.
6. **Model Evaluation (*evaluate_model* function):**
   - Calculates various performance metrics: Accuracy, Precision, Recall, F1-Score, Specificity, NPV, FPR, FDR, FNR, and MCC.
7. **Main Execution Block:**
   - Loads the dataset.
   - Separates features (X) and labels (y).
   - Selects features using *select_features*.
   - Splits the data into training, validation, and testing sets.
   - Trains the models using *train_models*.

- Saves the trained models, feature selector, and *scaler* using *joblib.dump*.
- Performs prediction and evaluation on the test set.
- Prints the evaluation results.

8. **predict_server.py Script:**
   - Loads the saved models, feature selector, and *scaler*.
   - Implements the *predict_new_data* function to make predictions on new data received from the P4 switch.
   - Includes a placeholder for receiving data from the P4 switch (e.g., via a message queue or API).

Step-by-Step Deployment Instructions for Reproducibility:

1. **Environment Setup (Local Machine):**
   - Install Python 3.7+ (recommended).
   - Create a virtual environment (recommended):

   ```
   python3 -m venv .venv
   source .venv/bin/activate  # On Linux/macOS
   .venv\Scripts\activate  # On Windows
   ```

   - Install required Python packages:
     `pip install pandas numpy scikit-learn joblib xgboost imblearn`

2. **Dataset Preparation:**
   - Download the CICIoT2023 dataset.
   - Place the dataset file (dataset.csv) in the same directory as the Python script.
3. **Training and Model Saving:**
   - Run the main Python script: `python MT-EDD.py`.
   - This will train the models and save them as `.pkl` files in the same directory.
4. **Cloud Server Setup:**
   - Choose a cloud provider (AWS, Azure, GCP, etc.) and create a VM instance (at least 2 vCPUs and 4GB RAM recommended).
   - Connect to the VM via SSH.
   - Install Python 3 and required packages (same as on your local machine).
5. **File Transfer to Cloud:**
   - Copy the following files to the VM:
     - `predict_server.py`
     - `rf_model.pkl`
     - `dt_model.pkl`
     - `xgb_model.pkl`
     - `mlp_model.pkl`
     - `feature_selector.pkl`
     - `scaler.pkl`
   - Use `scp` or a similar tool for secure file transfer:
     `scp predict_server.py *.pkl your_username@your_vm_ip:/path/to/destination/`

6. **Running the Prediction Server (Cloud):**
   - SSH into the VM.
   - Navigate to the directory where you copied the files.
   - Run the prediction server: `python3 predict_server.py`
7. **P4 Switch and Data Communication:**
   - Configure your P4 switch to extract the necessary features and send them to the cloud server.
   - Implement a communication mechanism:
     - **Message Queue (Recommended):** Set up a message queue (Kafka, RabbitMQ) and configure the P4 switch to publish messages to it. Modify `predict_server.py` to consume messages from the queue.
     - **REST API:** Create a REST API endpoint on the cloud server using a framework like Flask or FastAPI. Configure the P4 switch to send HTTP requests to the API.

Deployment Instructions on the Cloud Server :

1. Create a Python script for real-time prediction (e.g., predict_server.py):

```
# predict_server.py
import joblib
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

# Load models, selector and scaler
rf = joblib.load('rf_model.pkl')
dt = joblib.load('dt_model.pkl')
xgb_model = joblib.load('xgb_model.pkl')
mlp = joblib.load('mlp_model.pkl')
selector = joblib.load('feature_selector.pkl')
scaler = joblib.load('scaler.pkl')
ensemble = [rf, dt, xgb_model]

def predict_ensemble(X, ensemble):
# ... (same predict_ensemble function as before)

def predict_new_data(new_data):
    new_data = pd.DataFrame([new_data])
    numerical_cols = new_data.select_dtypes(include=np.number).columns
    new_data[numerical_cols] = scaler.transform(new_data[numerical_cols])
    new_data_selected = selector.transform(new_data)
    ensemble_preds, confidences = predict_ensemble(new_data_selected, ensemble)
    weights = np.array([1/len(ensemble)]*len(ensemble)) # Get the weights from adaptation module in real
implementation
    weighted_pred = (confidences * weights[np.newaxis, :, np.newaxis]).sum(axis=1).argmax(axis=1)
    return weighted_pred

# Example usage
# In a real application, you would receive data from a message queue (e.g., Kafka, RabbitMQ)
# or a REST API endpoint.
new_sample = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0] # Replace with
your features
prediction = predict_new_data(new_sample)
print(f"Prediction: {prediction}")
```

2. Deploy predict_server.py on a cloud server:

a. Choose a cloud platform (e.g., AWS, Azure, Google Cloud).
b. Create a virtual machine (VM) instance.
c. Install Python and necessary libraries (scikit-learn, pandas, numpy, joblib, xgboost).
d. Copy the trained model files (rf_model.pkl, dt_model.pkl, xgb_model.pkl, mlp_model.pkl, feature_selector.pkl, scaler.pkl) and the predict_server.py script to the VM.
e. Run the predict_server.py script (e.g., using `python predict_server.py`).
f. Implement a mechanism to receive data from the P4 switch and send it to the predict_server.py script. This could be done using a message queue (e.g., Kafka, RabbitMQ) or a REST API.