SDN-Xplain: LLM-Driven Anomaly Detection and Human-Readable Explanation

Technical report

System Design Description

The system integrates a traditional machine-learning intrusion detection model with a large-language-model explanation layer, producing an IDS capable of both high-accuracy anomaly detection and operator-friendly interpretability. It is built around four main components: the data preprocessing pipeline, the XGBoost classifier, the SHAP interpretability module, and the LLM-based explanation generator. Together, these modules form a coherent architecture capable of receiving raw SDN flow features, classifying them, interpreting the decision, and finally producing a clear, natural-language justification suitable for human analysts.

The design begins with the ingestion of network flow data, specifically the KDD dataset, which is loaded, cleaned, and transformed into a numerical representation appropriate for machine learning. This task is handled in code_local.py, where categorical features such as protocol, service, and flag are encoded using LabelEncoder, and rare attack classes are filtered to improve model stability. The processed dataset is stratified into training and test splits, after which an XGBoost multiclass classifier is trained. The model uses soft-probability outputs, enabling both precise classification and well-calibrated confidence reporting. Once trained, the full model state—including encoders, feature names, and label mapping—is serialized into model_params.pkl, ensuring consistent preprocessing and prediction at inference time. SHAP is also applied at training time to verify model behavior and generate summary visualizations of feature contributions.

During deployment, the inference architecture defined in load_model.py receives SDN flow records in dictionary form. The file loads the serialized model, preprocesses incoming traffic to mirror training conditions, and handles unseen categorical values gracefully by reverting to default classes. After preprocessing, the classifier emits both the predicted attack class and the full probability distribution across all classes. Immediately afterward, SHAP is used to extract per-feature contributions specific to this single flow. This yields a structured explanation containing the base SHAP value, the predicted score, and a detailed breakdown of each feature's impact. These values are organized in a consistent dictionary format that the explanation system can use directly.

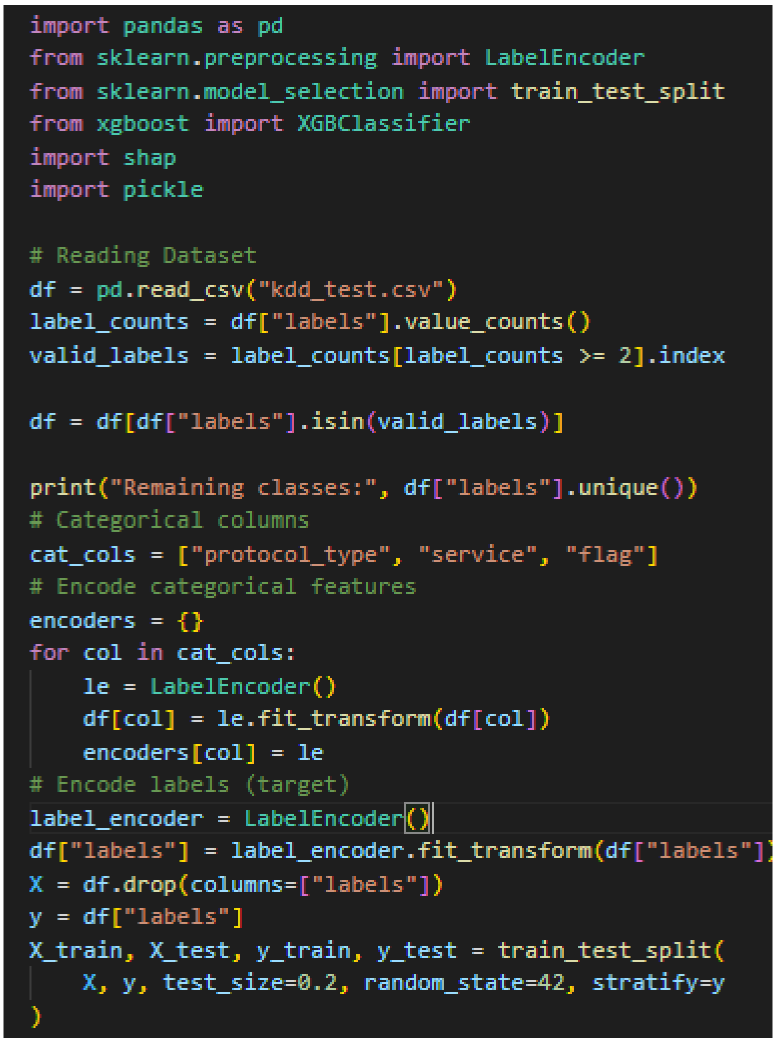
The explanation generator, defined in explanation_generator.py, transforms the machine-level interpretation produced by SHAP into a human-readable narrative. It constructs a structured prompt that embeds the predicted attack class, confidence score, probability vector, and the most influential SHAP features. To help the LLM contextualize the numerical contributions, each feature includes a descriptive definition that characterizes its meaning in network-security terms. The LLM is then instructed to produce a technically rigorous explanation covering the reasoning behind the classification, the typical behavior of the inferred attack type, potential security implications, mitigation recommendations, and any ambiguity suggested by the SHAP output. Through this mechanism, the system converts raw classifier output into an operational, analyst-oriented explanation.

This overall architecture can be summarized as a linear flow: the SDN controller generates flow records; the inference module normalizes them and extracts predictions; SHAP provides the internal rationale; and the LLM converts that rationale into a comprehensible written assessment. Although conceptually simple, this flow ensures that every automated decision remains transparent and traceable, a core requirement in modern SDN environments where explainability is essential for trust and actionable response.

Code Explanation and Snapshots

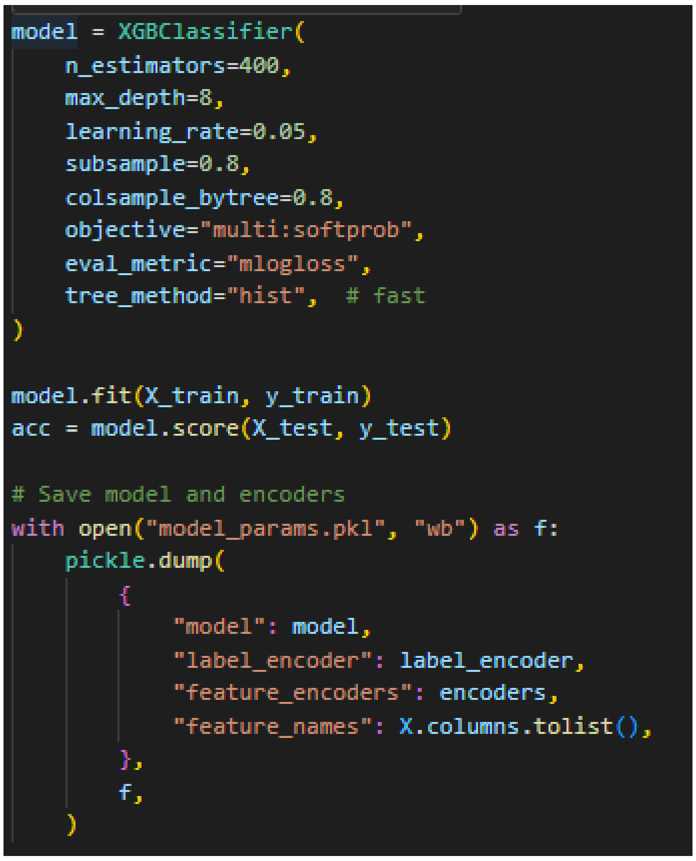
Training Pipeline

The training pipeline in code_local.py consists of a sequence of clearly defined phases. First, it loads the KDD input data, identifies and removes labels with insufficient representation, and encodes all categorical attributes.



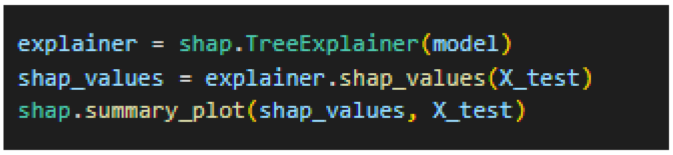
```python
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from xgboost import XGBClassifier
import shap
import pickle

# Reading Dataset
df = pd.read_csv("kdd_test.csv")
label_counts = df["labels"].value_counts()
valid_labels = label_counts[label_counts >= 2].index

df = df[df["labels"].isin(valid_labels)]

print("Remaining classes:", df["labels"].unique())
# Categorical columns
cat_cols = ["protocol_type", "service", "flag"]
# Encode categorical features
encoders = {}
for col in cat_cols:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col])
    encoders[col] = le
# Encode labels (target)
label_encoder = LabelEncoder()
df["labels"] = label_encoder.fit_transform(df["labels"]
X = df.drop(columns=["labels"])
y = df["labels"]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

Afterward, the script trains the XGBoost multiclass model using parameters optimized for balanced performance across attack types. Once the training concludes, it serializes the model along with all encoders and metadata.



```python
model = XGBClassifier(
    n_estimators=400,
    max_depth=8,
    learning_rate=0.05,
    subsample=0.8,
    colsample_bytree=0.8,
    objective="multi:softprob",
    eval_metric="mlogloss",
    tree_method="hist",  # fast
)

model.fit(X_train, y_train)
acc = model.score(X_test, y_test)

# Save model and encoders
with open("model_params.pkl", "wb") as f:
    pickle.dump(
        {
            "model": model,
            "label_encoder": label_encoder,
            "feature_encoders": encoders,
            "feature_names": X.columns.tolist(),
        },
        f,
    )
```

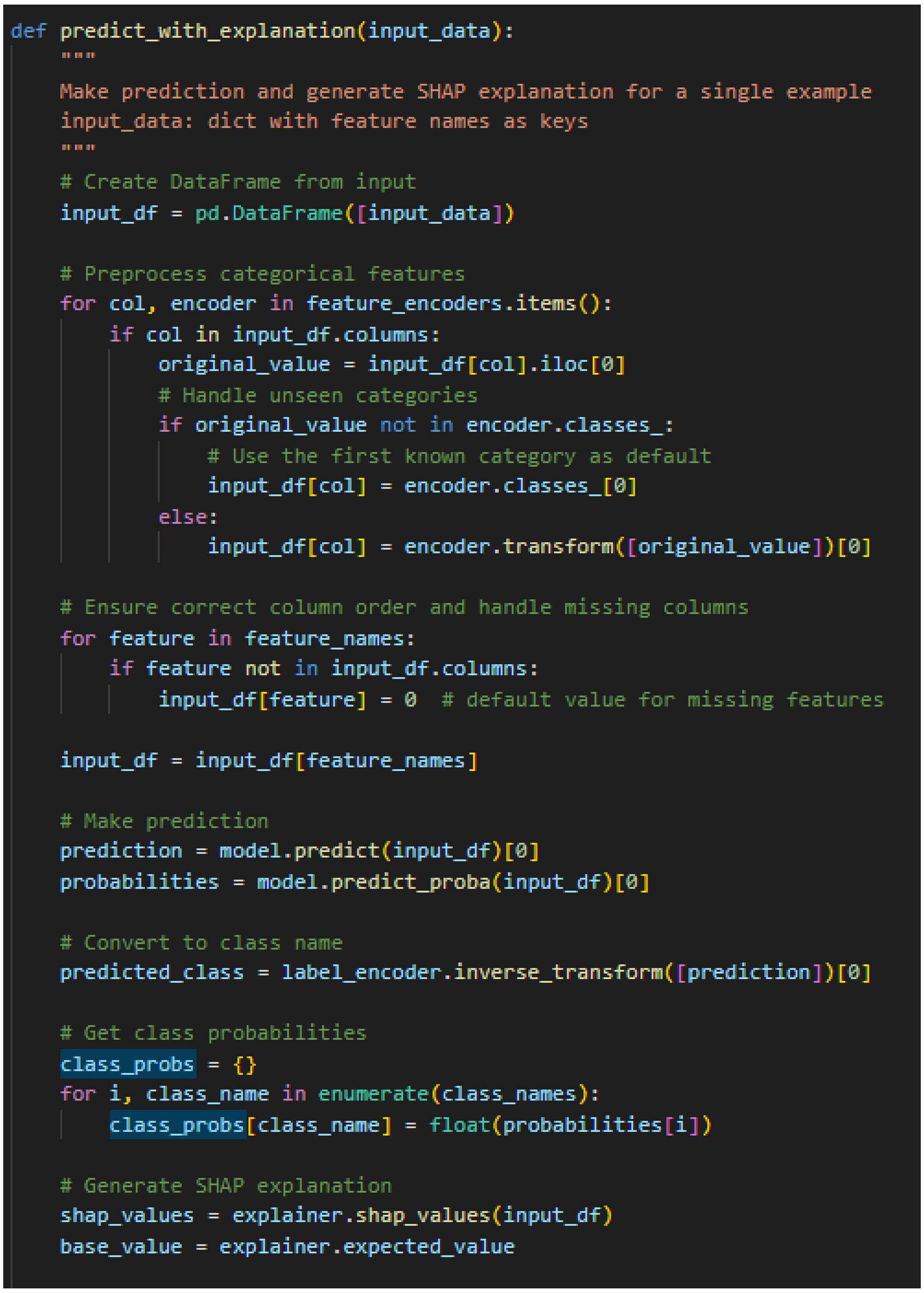
Finally, it initializes a SHAP TreeExplainer, computes SHAP values over the test set, and generates summary plots for model inspection.



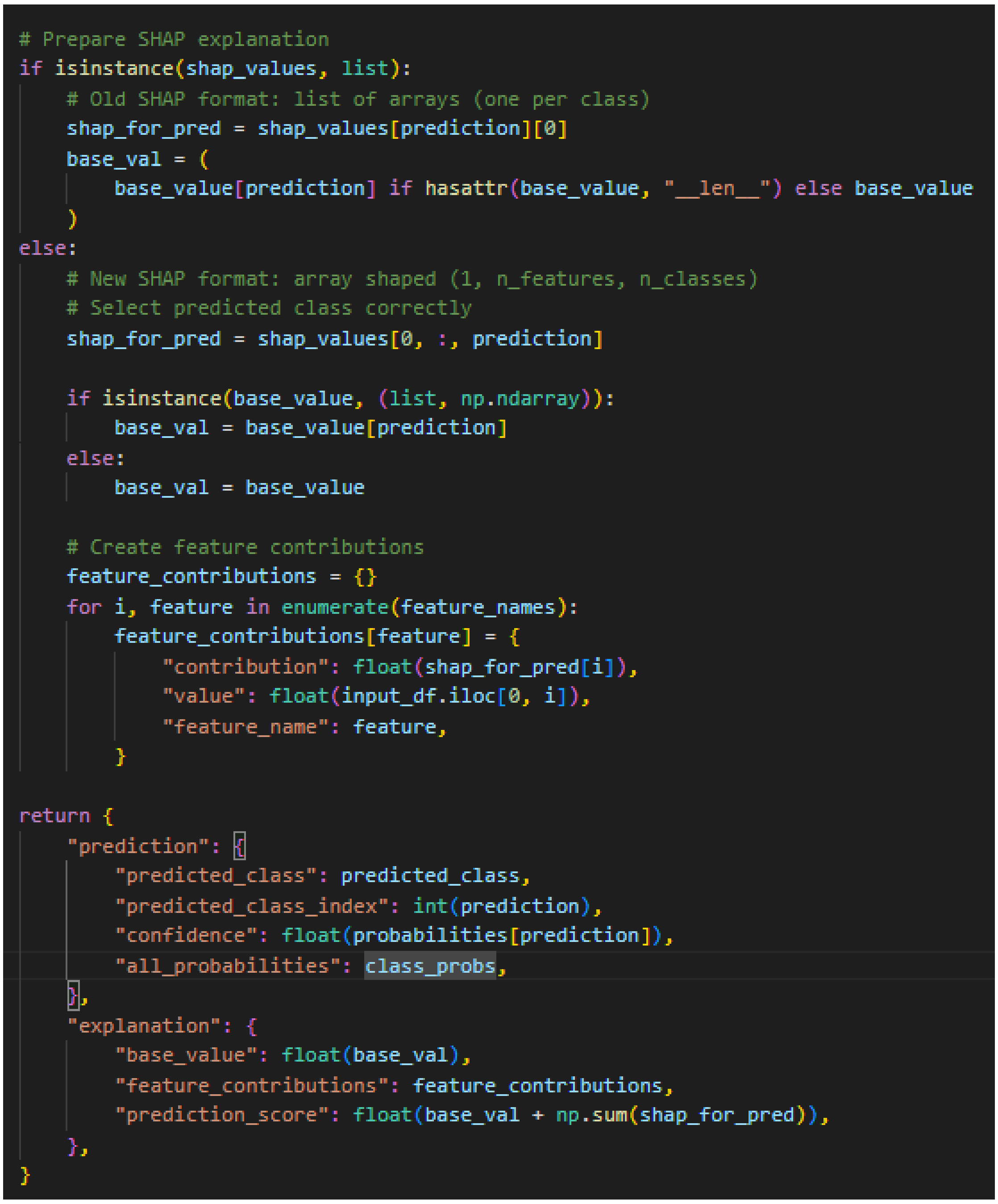
```python
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_test)
shap.summary_plot(shap_values, X_test)
```

Inference and SHAP Explanation Module

The inference and explanation logic resides in load_model.py. The key function, predict_with_explanation, reconstructs the preprocessing pipeline used during training, ensuring that each feature appears in the correct order and that categorical values are encoded consistently. Unseen categories are handled gracefully, preventing runtime failures. Once the input is validated and transformed, the model produces both the predicted label and probability distribution. SHAP is then applied to calculate a per-feature explanation vector. These values are assembled into a structured dictionary containing prediction results, the base SHAP value, and the contribution score for each feature.



```python
def predict_with_explanation(input_data):
    """
    Make prediction and generate SHAP explanation for a single example
    input_data: dict with feature names as keys
    """
    # Create DataFrame from input
    input_df = pd.DataFrame([input_data])

    # Preprocess categorical features
    for col, encoder in feature_encoders.items():
        if col in input_df.columns:
            original_value = input_df[col].iloc[0]
            # Handle unseen categories
            if original_value not in encoder.classes_:
                # Use the first known category as default
                input_df[col] = encoder.classes_[0]
            else:
                input_df[col] = encoder.transform([original_value])[0]

    # Ensure correct column order and handle missing columns
    for feature in feature_names:
        if feature not in input_df.columns:
            input_df[feature] = 0  # default value for missing features

    input_df = input_df[feature_names]

    # Make prediction
    prediction = model.predict(input_df)[0]
    probabilities = model.predict_proba(input_df)[0]

    # Convert to class name
    predicted_class = label_encoder.inverse_transform([prediction])[0]

    # Get class probabilities
    class_probs = {}
    for i, class_name in enumerate(class_names):
        class_probs[class_name] = float(probabilities[i])

    # Generate SHAP explanation
    shap_values = explainer.shap_values(input_df)
    base_value = explainer.expected_value
```



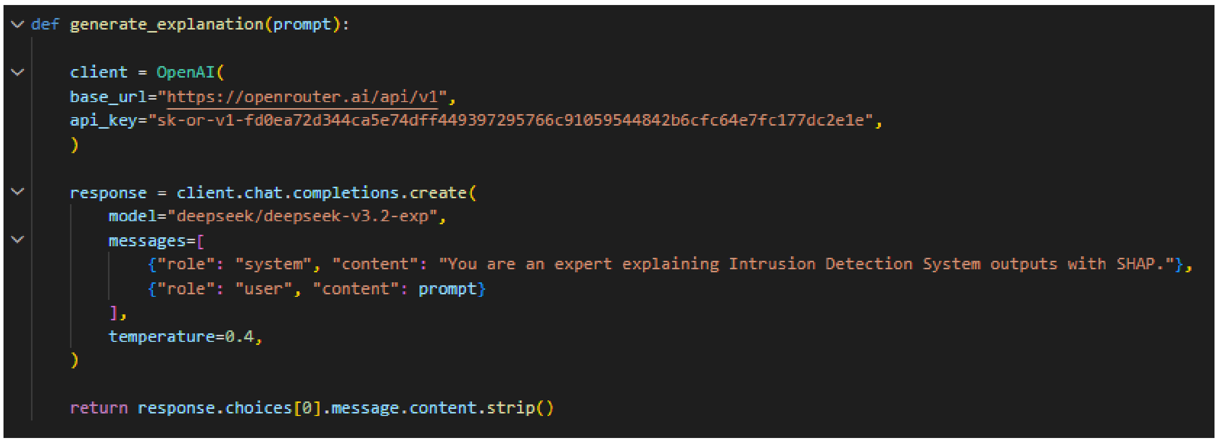
```python
# Prepare SHAP explanation
if isinstance(shap_values, list):
    # Old SHAP format: list of arrays (one per class)
    shap_for_pred = shap_values[prediction][0]
    base_val = (
        base_value[prediction] if hasattr(base_value, "__len__") else base_value
    )
else:
    # New SHAP format: array shaped (1, n_features, n_classes)
    # Select predicted class correctly
    shap_for_pred = shap_values[0, :, prediction]

    if isinstance(base_value, (list, np.ndarray)):
        base_val = base_value[prediction]
    else:
        base_val = base_value

    # Create feature contributions
    feature_contributions = {}
    for i, feature in enumerate(feature_names):
        feature_contributions[feature] = {
            "contribution": float(shap_for_pred[i]),
            "value": float(input_df.iloc[0, i]),
            "feature_name": feature,
        }

return {
    "prediction": {
        "predicted_class": predicted_class,
        "predicted_class_index": int(prediction),
        "confidence": float(probabilities[prediction]),
        "all_probabilities": class_probs,
    },
    "explanation": {
        "base_value": float(base_val),
        "feature_contributions": feature_contributions,
        "prediction_score": float(base_val + np.sum(shap_for_pred)),
    },
}
```

The file also includes helper functions for displaying results in terminal form and for processing entire CSV files in batch or interactive mode, enabling fast experimentation with large traffic logs.

LLM Explanation Module

The explanation-generation logic inside explanation_generator.py is responsible for translating this structured technical data into a narrative format. It builds a detailed prompt summarizing the prediction, probabilities, and top SHAP features. This prompt embeds human-readable descriptions of the features to give the LLM the necessary context. The generate_explanation function then calls the

LLM through the OpenRouter API and returns a fully formed explanation that is suitable for SDN dashboards, analyst tools, or automated reports.



```python
def generate_explanation(prompt):

    client = OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key="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",
    )

    response = client.chat.completions.create(
        model="deepseek/deepseek-v3.2-exp",
        messages=[
            {"role": "system", "content": "You are an expert explaining Intrusion Detection System outputs with SHAP."},
            {"role": "user", "content": prompt}
        ],
        temperature=0.4,
    )

    return response.choices[0].message.content.strip()
```

Usage and Testing

This section provides practical guidance for running the SDN-Xplain system, including model training, inference, explanation generation, and end-to-end validation. The instructions assume a standard Python 3.10+ environment with all dependencies installed (XGBoost, SHAP, pandas, scikit-learn, OpenAI/OpenRouter client, etc.).

Environment Setup

Before running the system, install the required libraries using the command:
`pip install pandas scikit-learn xgboost shap openai`
Ensure that the KDD dataset is available (e.g., `kdd_test.csv`) and that your filesystem contains:

- `code_local.py`: training pipeline
- `load_model.py`: inference + SHAP computation
- `explanation_generator.py`: LLM explanation layer

Training the model

To train the XGBoost IDS model, execute the command:
`python code_local.py`

During execution, the script will:

1. Load the KDD dataset.
2. Filter rare attack classes.
3. Apply LabelEncoder to categorical features.
4. Split the dataset using stratified sampling.

5. Train an XGBoost multiclass classifier.
6. Compute SHAP values on the test set for validation
7. Save the complete model bundle into model_params.pkl

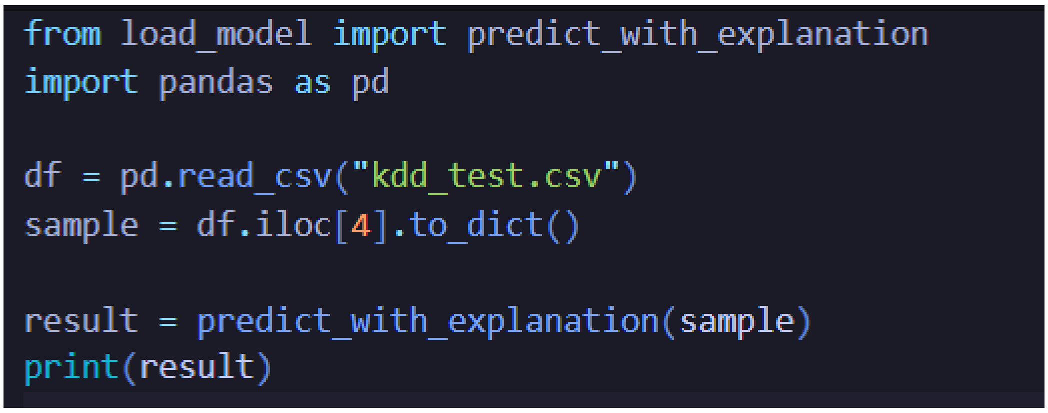
After training, the following file and graphs are generated:

- **model_params.pkl:** serialized model, encoders, and metadata
- **SHAP summary plots**: visual verification of feature behavior
- **Terminal logs**: training accuracy and class filtering results

These artifacts are required for deployment-time inference

Running Inference

To test the model with real SDN flow samples or rows from the dataset use the code:



```python
from load_model import predict_with_explanation
import pandas as pd

df = pd.read_csv("kdd_test.csv")
sample = df.iloc[4].to_dict()

result = predict_with_explanation(sample)
print(result)
```

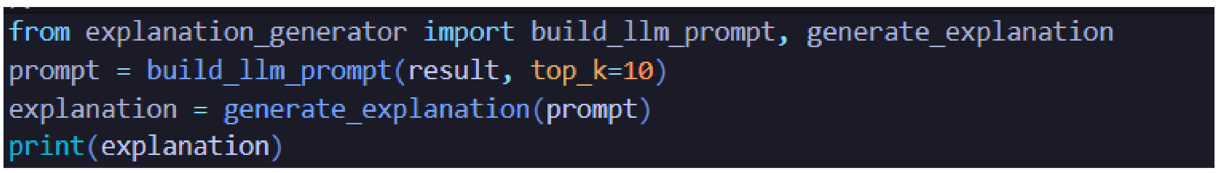
The output includes:

- Predicted class
- Confidence score
- Full probability vector
- Per-feature SHAP contributions
- Base value and expected model output

This output serves as the input to the explanation layer described next.

Generating LLM Explanations

Using explanation_generator.py, convert SHAP-based reasoning into a human-readable analysis. The following code demonstrates how to invoke the explanation generator:



```python
from explanation_generator import build_llm_prompt, generate_explanation
prompt = build_llm_prompt(result, top_k=10)
explanation = generate_explanation(prompt)
print(explanation)
```

The explanation provides:

- A justification for the predicted attack type
- Behavior profile of the attack
- Security implications
- Recommended mitigation steps
- Any uncertainty caused by weak or conflicting SHAP signals

This makes the IDS output interpretable for real operators.

Batch Processing for Large Traffic Logs

The inference module also supports multi-row CSV processing using the command:

```
python load_model.py --csv input_flows.csv --output results.json
```

This mode is used for evaluating large SDN traffic captures or log exports.

Conclusion

These usage and testing procedures ensure that SDN-Xplain operates correctly across all stages—from data preprocessing and model training to SHAP-driven reasoning and LLM-based explanation generation. This enables repeatable evaluation, transparent anomaly detection, and operator-ready interpretability within SDN environments.