About This Document

This PDF serves as an alternative source for the experiment code provided in this repository. While the original Python scripts (.py files) are available in the *Supplementary_Material*, this document compiles their complete content for convenient reference, offline access, and citation.

1. Binary Classification Experiment (*Binary_Classification_Experiment.py)*

```
#!/usr/bin/env python3
# ============================================================
# Binary_Classification_Experiment.py
# ============================================================
# Reproducible script for binary classification on the preprocessed
# "Binary_FeatureSelected_Dataset.csv" dataset.
#
# This dataset already includes:
#   - Selected features after LightGBM-based feature importance analysis
#   - Final binary labels ("Benign" / "Attack")
#
# The script performs model training, evaluation, and latency
# benchmarking for all models reported in the journal.
# No additional feature engineering, undersampling, or export is done.
# ============================================================

import os
import time
import warnings
import numpy as np
import pandas as pd
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Machine Learning Models
import xgboost as xgb
import lightgbm as lgb
from catboost import CatBoostClassifier
from ngboost import NGBClassifier
from ngboost.distns import Bernoulli
from sklearn.ensemble import ExtraTreesClassifier, RandomForestClassifier,
AdaBoostClassifier
from sklearn.linear_model import LogisticRegression, SGDClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB

# Ignore all warnings
warnings.filterwarnings('ignore')
```

```
# ------------------------------------------------------------------
# GPU Detection Function
# ------------------------------------------------------------------
def detect_gpu():
    """Detect available GPU and return device configurations"""
    gpu_available = False
    gpu_reason = "No GPU detected"

    # Check for CUDA
    try:
        import torch
        if torch.cuda.is_available():
            gpu_available = True
            gpu_reason = f"CUDA GPU available:
{torch.cuda.get_device_name(0)}"
        else:
            gpu_reason = "PyTorch CUDA not available"
    except ImportError:
        gpu_reason = "PyTorch not installed for GPU detection"

    # Additional check with tensorflow if available
    try:
        import tensorflow as tf
        gpus = tf.config.list_physical_devices('GPU')
        if gpus:
            gpu_available = True
            gpu_reason = f"TensorFlow GPU available: {gpus}"
    except ImportError:
        pass

    return gpu_available, gpu_reason


# Detect GPU
gpu_available, gpu_info = detect_gpu()
print(f"\n🔍 GPU Detection: {gpu_info}")
print(f"🎯 Using: {'GPU' if gpu_available else 'CPU'} for training")

# ------------------------------------------------------------------
# 1. Configuration
# ------------------------------------------------------------------
DATA_PATH = "Binary_FeatureSelected_Dataset.csv"  # Path to the uploaded
dataset
RANDOM_STATE = 42
TEST_SIZE = 0.2

# ------------------------------------------------------------------
# 2. Load and prepare data
# ------------------------------------------------------------------
print("\n[INFO] Loading dataset...")
df = pd.read_csv(DATA_PATH)
label_col = "binary_label"

X = df.drop(columns=[label_col]).values
y = (df[label_col].astype(str).str.lower() == "attack").astype(int).values
feature_names = [c for c in df.columns if c != label_col]
```

```
print(f"Dataset shape: {df.shape}")
print(f"Number of features: {len(feature_names)}")

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=TEST_SIZE, stratify=y, random_state=RANDOM_STATE
)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

neg, pos = np.bincount(y_train)
scale_pos_weight = neg / pos
print(f"scale_pos_weight = {scale_pos_weight:.2f}")

# For k-NN subsampling
KN_SAMPLE = 100_000
if X_train.shape[0] > KN_SAMPLE:
    knn_idx = np.random.choice(X_train.shape[0], KN_SAMPLE, replace=False)
    X_knn_train = X_train[knn_idx]
    y_knn_train = y_train[knn_idx]
else:
    X_knn_train, y_knn_train = X_train, y_train

# ----------------------------------------------------------------
# 3. Define models (EXACT CONFIGURATIONS FROM NOTEBOOK) with GPU fallback
# ----------------------------------------------------------------
# Set device parameters based on GPU availability
if gpu_available:
    xgb_device = "cuda"
    lgb_device = "gpu"
    catboost_task_type = "GPU"
else:
    xgb_device = "cpu"
    lgb_device = "cpu"
    catboost_task_type = "CPU"
    print("⚠  GPU not available - falling back to CPU training")

models = [
    ("XGBoost", xgb.XGBClassifier(
        objective="binary:logistic",
        eval_metric="logloss",
        n_estimators=500,
        learning_rate=0.12,
        max_depth=14,
        subsample=0.8,
        colsample_bytree=0.8,
        scale_pos_weight=scale_pos_weight,
        random_state=RANDOM_STATE,
        tree_method="hist",
        device=xgb_device  # Dynamic device selection
    )),

    ("LightGBM", lgb.LGBMClassifier(
        objective="binary",
```

```
        boosting_type="gbdt",
        n_estimators=500,
        learning_rate=0.12,
        max_depth=14,
        subsample=0.8,
        colsample_bytree=0.8,
        scale_pos_weight=scale_pos_weight,
        random_state=RANDOM_STATE,
        device=lgb_device,  # Dynamic device selection
        verbose=-1
    )),

    ("CatBoost", CatBoostClassifier(
        iterations=500,
        learning_rate=0.12,
        depth=14,
        loss_function="Logloss",
        eval_metric="Logloss",  # From notebook
        scale_pos_weight=scale_pos_weight,  # From notebook
        random_seed=RANDOM_STATE,
        task_type=catboost_task_type,  # Dynamic device selection
        verbose=False
    )),

    ("NGBoost", NGBClassifier(
        Dist=Bernoulli,
        n_estimators=100,  # From notebook (100 instead of 500)
        learning_rate=0.12,
        natural_gradient=False,  # From notebook
        verbose=False,  # From notebook
        random_state=RANDOM_STATE
    )),

    ("ExtraTrees", ExtraTreesClassifier(
        n_estimators=300,  # From notebook
        max_depth=14,
        class_weight="balanced",
        n_jobs=-1,
        random_state=RANDOM_STATE
    )),

    ("LogisticRegression", LogisticRegression(
        class_weight="balanced",
        max_iter=500,
        random_state=RANDOM_STATE
    )),
]

# Baseline traditional models (EXACT FROM NOTEBOOK)
baseline_models = [
    (DecisionTreeClassifier(
        max_depth=10,
        min_samples_leaf=50,
        class_weight='balanced',
        random_state=RANDOM_STATE
    ), "Decision Tree"),
```

```
    (RandomForestClassifier(
        n_estimators=100,
        max_depth=14,
        max_samples=0.1,
        class_weight='balanced',
        n_jobs=-1,
        random_state=RANDOM_STATE
    ), "Random Forest"),

    (LogisticRegression(  # Second LogisticRegression from notebook
        solver='saga',
        max_iter=500,
        class_weight='balanced',
        random_state=RANDOM_STATE
    ), "Logistic Regression (SAGA)"),

    (SGDClassifier(
        loss='hinge',
        learning_rate='optimal',
        max_iter=1000,
        tol=1e-3,
        class_weight='balanced',
        random_state=RANDOM_STATE
    ), "Linear SVM (SGD)"),

    (KNeighborsClassifier(n_neighbors=5, n_jobs=-1),
     "k-NN (subsampled)"),

    (GaussianNB(), "Gaussian Naive Bayes"),

    (AdaBoostClassifier(
        n_estimators=50,
        learning_rate=0.1,
        random_state=RANDOM_STATE
    ), "AdaBoost"),
]

# Combine all models
all_models = models + [(model, name) for model, name in baseline_models]


# ------------------------------------------------------------------
# 4. Helper: Latency Measurement
# ------------------------------------------------------------------
def measure_latency(model, X):
    """Return total and per-sample inference time in ms."""
    _ = model.predict(X[:1])  # Warm-up
    start = time.perf_counter()
    _ = model.predict(X)
    end = time.perf_counter()
    total_ms = (end - start) * 1000
    return total_ms, total_ms / X.shape[0]


def print_confusion_matrix(cm, model_name):
    """Print confusion matrix in a formatted way."""
```

```
    print(f"\n📊 CONFUSION MATRIX - {model_name}:")
    print(" " * 20 + "Predicted")
    print(" " * 15 + "Benign    Attack")
    print(" " * 10 + "┌──────────┬──────────┐")
    print(f"True Benign  │    {cm[0, 0]:^6}  │    {cm[0, 1]:^6}  │")
    print(" " * 10 + "├──────────┼──────────┤")
    print(f"True Attack  │    {cm[1, 0]:^6}  │    {cm[1, 1]:^6}  │")
    print(" " * 10 + "└──────────┴──────────┘")


def evaluate_model(model, name, X_test, y_test):
    """Evaluate model and print results to console."""
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=["Benign",
"Attack"], digits=4)

    print(f"\n{'=' * 80}")
    print(f"📊 {name} Evaluation Results")
    print(f"{'=' * 80}")
    print(f"☑ Accuracy: {acc * 100:.4f}%")

    # Print confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    print_confusion_matrix(cm, name)

    print(f"\n📋 Classification Report:")
    print(report)

    return acc, classification_report(y_test, y_pred, target_names=["Benign",
"Attack"], digits=4, output_dict=True)


# ------------------------------------------------------------------
# 5. Train, Evaluate, and Report with tqdm
# ------------------------------------------------------------------
print(f"\n🎯 Starting training and evaluation of {len(all_models)}
models...")
print("=" * 80)

results = []
trained_models = []

for name, model in tqdm(all_models, desc="Training Models", unit="model"):
    print(f"\n🚀 Training {name}...")

    # Special handling for k-NN subsampling
    if name.startswith("k-NN"):
        model.fit(X_knn_train, y_knn_train)
    else:
        model.fit(X_train, y_train)

    # Evaluate model
    acc, report_dict = evaluate_model(model, name, X_test, y_test)
```

```
    # Measure latency
    total_ms, avg_ms = measure_latency(model, X_test)

    # Store results
    results.append({
        "Model": name,
        "Accuracy": acc,
        "Macro_F1": report_dict["macro avg"]["f1-score"],
        "Weighted_F1": report_dict["weighted avg"]["f1-score"],
        "Total_ms": total_ms,
        "PerSample_ms": avg_ms
    })

    trained_models.append((model, name))

    print(f"⏱️  Inference Time: {total_ms:.1f} ms total, {avg_ms:.4f} ms per
sample")

# ----------------------------------------------------------------
# 6. Display Final Summary
# ----------------------------------------------------------------
print(f"\n{'=' * 80}")
print("🎉 FINAL RESULTS SUMMARY")
print(f"{'=' * 80}")

results_df = pd.DataFrame(results)
results_df = results_df.sort_values("Accuracy", ascending=False)

print("\n" + "=" * 100)
print(
    f"{'Model':<25} {'Accuracy':<12} {'Macro F1':<12} {'Weighted F1':<12}
{'Total Time (ms)':<15} {'Per Sample (ms)':<15}")
print("=" * 100)

for _, res in results_df.iterrows():
    print(
        f"{res['Model']:<25} {res['Accuracy']:<12.4f}
{res['Macro_F1']:<12.4f} {res['Weighted_F1']:<12.4f} {res['Total_ms']:<15.1f}
{res['PerSample_ms']:<15.4f}")

print("=" * 100)

# Print best model
best_model = results_df.iloc[0]
print(f"\n🏆 BEST MODEL: {best_model['Model']}")
print(f"   📈 Accuracy: {best_model['Accuracy']:.4f} ({best_model['Accuracy']
* 100:.2f}%)")
print(f"   🎯 Macro F1: {best_model['Macro_F1']:.4f}")
print(f"   🏅 Weighted F1: {best_model['Weighted_F1']:.4f}")
print(f"   ⏱️  Inference: {best_model['PerSample_ms']:.4f} ms per sample")

print(f"\n☑️ Experiment completed successfully!")
print(f"💻 Training Device: {'GPU' if gpu_available else 'CPU'}")
```

2. Eight-Class Classification Experiment (*EightClass_Classification_Experiment.py*)

```
#!/usr/bin/env python3
# ============================================================
# EightClass_Classification_Experiment.py
# ============================================================
# Reproducible experiment for 8-class classification using the
# preprocessed "EightClass_FeatureSelected_Dataset.csv" dataset.
#
# This dataset already includes:
#   - Selected features after LightGBM-based feature selection
#   - 8 aggregated classes mapped from original CICIoT2023 labels
#
# The script performs:
#   - Train/test split
#   - Standardization
#   - Model training and evaluation (XGBoost, CatBoost, LightGBM)
#   - Latency benchmarking
#
# No feature extraction, plotting, or Excel exporting is performed.
# ============================================================

import os
import time
import warnings
import numpy as np
import pandas as pd
from collections import Counter
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# ML libraries
import xgboost as xgb
import lightgbm as lgb
from catboost import CatBoostClassifier

# Ignore all warnings
warnings.filterwarnings('ignore')


# ------------------------------------------------------------
# GPU Detection Function
# ------------------------------------------------------------
def detect_gpu():
    """Detect available GPU and return device configurations"""
    gpu_available = False
    gpu_reason = "No GPU detected"

    # Check for CUDA
    try:
        import torch
        if torch.cuda.is_available():
```

```
            gpu_available = True
            gpu_reason = f"CUDA GPU available:
{torch.cuda.get_device_name(0)}"
        else:
            gpu_reason = "PyTorch CUDA not available"
    except ImportError:
        gpu_reason = "PyTorch not installed for GPU detection"

    # Additional check with tensorflow if available
    try:
        import tensorflow as tf
        gpus = tf.config.list_physical_devices('GPU')
        if gpus:
            gpu_available = True
            gpu_reason = f"TensorFlow GPU available: {gpus}"
    except ImportError:
        pass

    return gpu_available, gpu_reason


# Detect GPU
gpu_available, gpu_info = detect_gpu()
print(f"\n🔍 GPU Detection: {gpu_info}")
print(f"🎯 Using: {'GPU' if gpu_available else 'CPU'} for training")

# ----------------------------------------------------------------
# 1. Configuration
# ----------------------------------------------------------------
DATA_PATH = "EightClass_FeatureSelected_Dataset.csv"  # Path to dataset
LABEL_COL = "label"
RANDOM_STATE = 42
TEST_SIZE = 0.2

# ----------------------------------------------------------------
# 2. Load and prepare data
# ----------------------------------------------------------------
print("\n[INFO] Loading dataset...")
df = pd.read_csv(DATA_PATH)
assert LABEL_COL in df.columns, f"'{LABEL_COL}' not found in dataset
columns."

X = df.drop(columns=[LABEL_COL]).values
y = df[LABEL_COL].astype(str)

classes = sorted(y.unique())
label_to_int = {lbl: i for i, lbl in enumerate(classes)}
y_num = y.map(label_to_int)

print(f"Dataset shape: {df.shape}")
print(f"Number of features: {X.shape[1]}")
print(f"Classes: {classes}")

X_train, X_test, y_train, y_test = train_test_split(
    X, y_num, test_size=TEST_SIZE, stratify=y_num, random_state=RANDOM_STATE
)
```

```
# Standardize
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Compute sample weights for balancing
train_counts = Counter(y_train)
n_classes = len(classes)
sample_weights = [len(y_train) / (n_classes * train_counts[c]) for c in
range(n_classes)]
weights = [sample_weights[label] for label in y_train]

# ----------------------------------------------------------------
# 3. Define Models (EXACT CONFIGURATIONS FROM NOTEBOOK) with GPU fallback
# ----------------------------------------------------------------
# Set device parameters based on GPU availability
if gpu_available:
    xgb_device = "cuda"
    xgb_predictor = "gpu_predictor"
    catboost_task_type = "GPU"
    lgb_device = "gpu"
else:
    xgb_device = "cpu"
    xgb_predictor = "cpu_predictor"
    catboost_task_type = "CPU"
    lgb_device = "cpu"
    print("⚠  GPU not available - falling back to CPU training")

models = [
    ("XGBoost", xgb.XGBClassifier(
        device=xgb_device,  # Dynamic device selection
        predictor=xgb_predictor,  # Dynamic predictor selection
        n_estimators=500,
        learning_rate=0.1,
        max_depth=10,
        use_label_encoder=False,
        eval_metric="mlogloss",
        random_state=RANDOM_STATE
    )),

    ("CatBoost", CatBoostClassifier(
        iterations=500,
        learning_rate=0.1,
        depth=10,
        loss_function="MultiClass",
        eval_metric="MultiClass",
        task_type=catboost_task_type,  # Dynamic device selection
        random_seed=RANDOM_STATE,
        verbose=False
    )),

    ("LightGBM", lgb.LGBMClassifier(
        objective="multiclass",
        num_class=n_classes,
        n_estimators=500,
```

```
        learning_rate=0.1,
        max_depth=7,
        num_leaves=63,
        subsample=1.0,
        colsample_bytree=0.8,
        min_child_samples=300,
        min_split_gain=0.01,
        random_state=RANDOM_STATE,
        device=lgb_device,  # Dynamic device selection
        verbose=-1
    )),
]


# ----------------------------------------------------------------
# 4. Helper: Measure Inference Latency
# ----------------------------------------------------------------
def measure_inference_latency(model, X):
    """Return total and per-sample inference time in ms."""
    _ = model.predict(X[:1])  # Warm-up
    start = time.perf_counter()
    _ = model.predict(X)
    end = time.perf_counter()
    total_ms = (end - start) * 1000
    return total_ms, total_ms / X.shape[0]


def print_confusion_matrix_8class(cm, model_name, classes):
    """Print formatted confusion matrix for 8 classes."""
    print(f"\n📊 CONFUSION MATRIX - {model_name}:")
    print(" " * 15 + "Predicted →")
    header = "True ↓" + " " * 9
    for cls in classes:
        header += f"{cls[:8]:<10}"
    print(header)
    print(" " * 10 + "─" * (len(classes) * 10 + 5))

    for i, true_class in enumerate(classes):
        row = f"{true_class[:8]:<10}"
        for j in range(len(classes)):
            row += f"{cm[i, j]:<10}"
        print(row)
    print()


def evaluate_8class_model(model, name, X_test, y_test, classes):
    """Evaluate model and print results to console."""
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=classes,
digits=4)

    print(f"\n{'=' * 80}")
    print(f"🎯 {name} - 8-Class Evaluation Results")
    print(f"{'=' * 80}")
    print(f"☑ Accuracy: {acc * 100:.4f}%")
```

```
    # Print confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    print_confusion_matrix_8class(cm, name, classes)

    print(f"📋 Classification Report:")
    print(report)

    return acc, classification_report(y_test, y_pred, target_names=classes,
digits=4, output_dict=True)


# ------------------------------------------------------------------
# 5. Train, Evaluate, and Report with tqdm
# ------------------------------------------------------------------
print(f"\n🎯 Starting 8-class training and evaluation of {len(models)}
models...")
print("=" * 80)

results = []
trained_models = []

for name, model in tqdm(models, desc="Training Models", unit="model"):
    print(f"\n🚀 Training {name}...")

    # Train with exact notebook configurations
    if name == "CatBoost":
        model.fit(X_train, y_train, sample_weight=weights, eval_set=(X_test,
y_test))
    elif name == "LightGBM":
        model.fit(X_train, y_train, sample_weight=weights, eval_set=[(X_test,
y_test)],
                  eval_metric="multi_logloss",
                  callbacks=[lgb.early_stopping(stopping_rounds=30),
lgb.log_evaluation(period=50)])
    else:
        model.fit(X_train, y_train, sample_weight=weights, eval_set=[(X_test,
y_test)], verbose=50)

    # Evaluate model
    acc, report_dict = evaluate_8class_model(model, name, X_test, y_test,
classes)

    # Measure latency
    total_ms, avg_ms = measure_inference_latency(model, X_test)

    # Store results
    results.append({
        "Model": name,
        "Accuracy": acc,
        "Macro_F1": report_dict["macro avg"]["f1-score"],
        "Weighted_F1": report_dict["weighted avg"]["f1-score"],
        "Total_ms": total_ms,
        "PerSample_ms": avg_ms
    })
```

```
    trained_models.append((model, name))

    print(f"⏱️  Inference Time: {total_ms:.1f} ms total, {avg_ms:.4f} ms per
sample")

# ----------------------------------------------------------------
# 6. Display Final Summary
# ----------------------------------------------------------------
print(f"\n{'=' * 80}")
print("🎉 FINAL 8-CLASS RESULTS SUMMARY")
print(f"{'=' * 80}")

results_df = pd.DataFrame(results)
results_df = results_df.sort_values("Accuracy", ascending=False)

print("\n" + "=" * 100)
print(
    f"{'Model':<15} {'Accuracy':<12} {'Macro F1':<12} {'Weighted F1':<12}
{'Total Time (ms)':<15} {'Per Sample (ms)':<15}")
print("=" * 100)

for _, res in results_df.iterrows():
    print(
        f"{res['Model']:<15} {res['Accuracy']:<12.4f}
{res['Macro_F1']:<12.4f} {res['Weighted_F1']:<12.4f} {res['Total_ms']:<15.1f}
{res['PerSample_ms']:<15.4f}")

print("=" * 100)

# Print best model
best_model = results_df.iloc[0]
print(f"\n🏆 BEST MODEL: {best_model['Model']}")
print(f"   📈 Accuracy: {best_model['Accuracy']:.4f} ({best_model['Accuracy']
* 100:.2f}%)")
print(f"   🎯 Macro F1: {best_model['Macro_F1']:.4f}")
print(f"   🏅 Weighted F1: {best_model['Weighted_F1']:.4f}")
print(f"   ⏱️  Inference: {best_model['PerSample_ms']:.4f} ms per sample")

# Print class distribution
print(f"\n📊 CLASS DISTRIBUTION:")
print(f"   Training samples: {len(y_train):,}")
print(f"   Testing samples: {len(y_test):,}")
print(f"   Number of classes: {n_classes}")
train_class_counts = Counter(y_train)
print(f"   Training class distribution:")
for class_idx, count in train_class_counts.items():
    class_name = classes[class_idx]
    print(f"     {class_name}: {count:,} samples")

print(f"\n☑️ 8-Class experiment completed successfully!.")
print(f"💻 Training Device: {'GPU' if gpu_available else 'CPU'}")
```

3. Thirty-Four-Class Classification Experiment (*ThirtyFourClass_Classification_Experiment.py*)

```
#!/usr/bin/env python3
# ================================================================
# ThirtyFourClass_Classification_Experiment.py
# ================================================================
# Reproducible 34-class classification experiment
# using the preprocessed "ThirtyFourClass_FeatureSelected_Dataset.csv".
#
# Dataset already includes:
#   - Selected features (after LightGBM-based selection)
#   - Stratified undersampling of original CICIoT2023 data
#   - 34 aggregated classes
#
# Script performs:
#   - Train/test split
#   - Feature standardization
#   - Model training (XGBoost, CatBoost, LightGBM)
#   - Performance evaluation and latency benchmarking
#
# Outputs:
#   - Printed metrics (Accuracy, Macro-F1, Weighted-F1, latency)
#   - No figures or Excel files are generated.
# ================================================================

import os
import time
import warnings
import numpy as np
import pandas as pd
from collections import Counter
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# ML frameworks
import xgboost as xgb
import lightgbm as lgb
from catboost import CatBoostClassifier

# Ignore all warnings
warnings.filterwarnings('ignore')


# ----------------------------------------------------------------
# GPU Detection Function
# ----------------------------------------------------------------
def detect_gpu():
    """Detect available GPU and return device configurations"""
```

```
    gpu_available = False
    gpu_reason = "No GPU detected"

    # Check for CUDA
    try:
        import torch
        if torch.cuda.is_available():
            gpu_available = True
            gpu_reason = f"CUDA GPU available:
{torch.cuda.get_device_name(0)}"
        else:
            gpu_reason = "PyTorch CUDA not available"
    except ImportError:
        gpu_reason = "PyTorch not installed for GPU detection"

    # Additional check with tensorflow if available
    try:
        import tensorflow as tf
        gpus = tf.config.list_physical_devices('GPU')
        if gpus:
            gpu_available = True
            gpu_reason = f"TensorFlow GPU available: {gpus}"
    except ImportError:
        pass

    return gpu_available, gpu_reason


# Detect GPU
gpu_available, gpu_info = detect_gpu()
print(f"\n🔍 GPU Detection: {gpu_info}")
print(f"🎯 Using: {'GPU' if gpu_available else 'CPU'} for training")

# ------------------------------------------------------------
# 1. Configuration
# ------------------------------------------------------------
DATA_PATH = "ThirtyFourClass_FeatureSelected_Dataset.csv"
LABEL_COL = "label"
RANDOM_STATE = 42
TEST_SIZE = 0.2

# ------------------------------------------------------------
# 2. Load Dataset
# ------------------------------------------------------------
print("\n[INFO] Loading 34-class dataset...")
df = pd.read_csv(DATA_PATH)
assert LABEL_COL in df.columns, f"'{LABEL_COL}' column missing."

X = df.drop(columns=[LABEL_COL]).values
y = df[LABEL_COL].astype(str)

classes = sorted(y.unique())
label_to_int = {lbl: i for i, lbl in enumerate(classes)}
y_num = y.map(label_to_int)

print(f"Dataset shape: {df.shape}")
```

```
print(f"Number of features: {X.shape[1]}")
print(f"Total classes: {len(classes)}")
print(f"Classes: {classes}")

# ----------------------------------------------------------------
# 3. Train/Test Split and Standardization
# ----------------------------------------------------------------
X_train, X_test, y_train, y_test = train_test_split(
    X, y_num, test_size=TEST_SIZE, stratify=y_num, random_state=RANDOM_STATE
)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Compute sample weights for class balance
train_counts = Counter(y_train)
n_classes = len(classes)
spw = [len(y_train) / (n_classes * train_counts[c]) for c in
range(n_classes)]
weights = [spw[label] for label in y_train]

# ----------------------------------------------------------------
# 4. Define Models (EXACT CONFIGURATIONS FROM NOTEBOOK) with GPU fallback
# ----------------------------------------------------------------
# Set device parameters based on GPU availability
if gpu_available:
    xgb_device = "cuda"
    xgb_predictor = "gpu_predictor"
    catboost_task_type = "GPU"
    lgb_device = "gpu"
else:
    xgb_device = "cpu"
    xgb_predictor = "cpu_predictor"
    catboost_task_type = "CPU"
    lgb_device = "cpu"
    print("⚠  GPU not available - falling back to CPU training")

models = [
    ("XGBoost", xgb.XGBClassifier(
        device=xgb_device,  # Dynamic device selection
        predictor=xgb_predictor,  # Dynamic predictor selection
        n_estimators=500,
        learning_rate=0.1,
        max_depth=10,
        use_label_encoder=False,
        eval_metric="mlogloss",
        random_state=RANDOM_STATE
    )),

    ("CatBoost", CatBoostClassifier(
        iterations=500,
        learning_rate=0.1,
        depth=10,
        loss_function="MultiClass",
        eval_metric="MultiClass",
```

```
        task_type=catboost_task_type,  # Dynamic device selection
        random_seed=RANDOM_STATE,
        verbose=False
    )),

    ("LightGBM", lgb.LGBMClassifier(
        objective="multiclass",
        num_class=n_classes,
        n_estimators=300,  # From notebook (300 instead of 500)
        learning_rate=0.1,
        max_depth=14,  # From notebook (14 instead of 10)
        num_leaves=63,
        subsample=1.0,
        colsample_bytree=0.8,
        min_child_samples=300,
        min_split_gain=0.01,
        random_state=RANDOM_STATE,
        device=lgb_device,  # Dynamic device selection
        verbose=-1
    )),
]


# ---------------------------------------------------------------
# 5. Helper for Inference Latency
# ---------------------------------------------------------------
def measure_inference_latency(model, X):
    """Return (total_ms, per_sample_ms) latency of model.predict()."""
    _ = model.predict(X[:1])  # Warm-up
    start = time.perf_counter()
    _ = model.predict(X)
    end = time.perf_counter()
    total_ms = (end - start) * 1000
    return total_ms, total_ms / X.shape[0]


def print_confusion_matrix_34class(cm, model_name, classes):
    """Print formatted confusion matrix summary for 34 classes."""
    print(f"\n📊 CONFUSION MATRIX SUMMARY - {model_name}:")
    print(f"   Shape: {cm.shape[0]}x{cm.shape[1]} (34x34 classes)")

    # Calculate key metrics
    correct_predictions = np.trace(cm)
    total_predictions = np.sum(cm)
    overall_accuracy = correct_predictions / total_predictions

    print(f"   Correct predictions: {correct_predictions:,}")
    print(f"   Total predictions: {total_predictions:,}")
    print(f"   Overall accuracy from CM: {overall_accuracy:.4f}")

    # Show per-class accuracy (diagonal / row sums)
    print(f"\n   Per-class accuracy (diagonal/row_sum):")
    class_accuracies = []
    for i in range(len(classes)):
        row_sum = np.sum(cm[i, :])
        if row_sum > 0:
```

```
            accuracy = cm[i, i] / row_sum
            class_accuracies.append(accuracy)
            if i < 5:  # Show first 5 classes as sample
                print(f"     {classes[i][:15]:<15}: {accuracy:.4f}")

    if len(classes) > 5:
        print(f"     ... and {len(classes) - 5} more classes")

    print(f"   Average per-class accuracy: {np.mean(class_accuracies):.4f}")


def evaluate_34class_model(model, name, X_test, y_test, classes):
    """Evaluate model and print results to console."""
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, target_names=classes,
digits=4)

    print(f"\n{'=' * 80}")
    print(f"🎯 {name} - 34-Class Evaluation Results")
    print(f"{'=' * 80}")
    print(f"☑ Accuracy: {acc * 100:.4f}%")

    # Print confusion matrix summary
    cm = confusion_matrix(y_test, y_pred)
    print_confusion_matrix_34class(cm, name, classes)

    print(f"\n📋 Classification Report (showing first 10 classes):")
    # Print only first 10 classes to avoid overwhelming output
    report_lines = report.split('\n')
    header_lines = report_lines[:4]  # Keep precision, recall, f1-score,
support headers
    class_lines = []

    for line in report_lines[4:]:
        if line.strip() and not any(avg in line for avg in ['macro avg',
'weighted avg', 'accuracy']):
            class_lines.append(line)
        elif any(avg in line for avg in ['macro avg', 'weighted avg',
'accuracy']):
            # Keep the average lines
            class_lines.append(line)

    # Show first 10 classes + averages
    output_lines = header_lines + class_lines[:10] + ['...'] + class_lines[-
3:]
    print('\n'.join(output_lines))

    return acc, classification_report(y_test, y_pred, target_names=classes,
digits=4, output_dict=True)


# ------------------------------------------------------------
# 6. Train and Evaluate Models with tqdm
# ------------------------------------------------------------
print(f"\n🎯 Starting 34-class training and evaluation of {len(models)}
```

```
models...")
print("=" * 80)

results = []
trained_models = []

for name, model in tqdm(models, desc="Training Models", unit="model"):
    print(f"\n🚀 Training {name}...")

    # Train with exact notebook configurations
    if name == "CatBoost":
        model.fit(X_train, y_train, sample_weight=weights, eval_set=(X_test,
y_test))
    elif name == "LightGBM":
        model.fit(X_train, y_train, sample_weight=weights, eval_set=[(X_test,
y_test)],
                  eval_metric="multi_logloss",
                  callbacks=[lgb.log_evaluation(period=50)])  # From notebook
    else:
        model.fit(X_train, y_train, sample_weight=weights, eval_set=[(X_test,
y_test)], verbose=50)  # From notebook

    # Evaluate model
    acc, report_dict = evaluate_34class_model(model, name, X_test, y_test,
classes)

    # Measure latency
    total_ms, avg_ms = measure_inference_latency(model, X_test)

    # Store results
    results.append({
        "Model": name,
        "Accuracy": acc,
        "Macro_F1": report_dict["macro avg"]["f1-score"],
        "Weighted_F1": report_dict["weighted avg"]["f1-score"],
        "Total_ms": total_ms,
        "PerSample_ms": avg_ms
    })

    trained_models.append((model, name))

    print(f"⏱️  Inference Time: {total_ms:.1f} ms total, {avg_ms:.4f} ms per
sample")

# ------------------------------------------------------------------
# 7. Display Final Summary
# ------------------------------------------------------------------
print(f"\n{'=' * 80}")
print("🎉 FINAL 34-CLASS RESULTS SUMMARY")
print(f"{'=' * 80}")

results_df = pd.DataFrame(results)
results_df = results_df.sort_values("Accuracy", ascending=False)

print("\n" + "=" * 100)
print(
```

```
    f"{'Model':<15} {'Accuracy':<12} {'Macro F1':<12} {'Weighted F1':<12}
{'Total Time (ms)':<15} {'Per Sample (ms)':<15}")
print("=" * 100)

for _, res in results_df.iterrows():
    print(
        f"{res['Model']:<15} {res['Accuracy']:<12.4f}
{res['Macro_F1']:<12.4f} {res['Weighted_F1']:<12.4f} {res['Total_ms']:<15.1f}
{res['PerSample_ms']:<15.4f}")

print("=" * 100)

# Print best model
best_model = results_df.iloc[0]
print(f"\n🏆 BEST MODEL: {best_model['Model']}")
print(f"   📈 Accuracy: {best_model['Accuracy']:.4f} ({best_model['Accuracy']
* 100:.2f}%)")
print(f"   🎯 Macro F1: {best_model['Macro_F1']:.4f}")
print(f"   ⚖️  Weighted F1: {best_model['Weighted_F1']:.4f}")
print(f"   ⏱️  Inference: {best_model['PerSample_ms']:.4f} ms per sample")

# Print class distribution summary
print(f"\n📊 CLASS DISTRIBUTION SUMMARY:")
print(f"   Training samples: {len(y_train):,}")
print(f"   Testing samples: {len(y_test):,}")
print(f"   Number of classes: {n_classes}")
train_class_counts = Counter(y_train)
print(f"   Training class distribution (sample):")
for i, (class_idx, count) in enumerate(train_class_counts.most_common()):
    if i < 10:  # Show top 10 classes
        class_name = classes[class_idx]
        print(f"     {class_name}: {count:,} samples")
    elif i == 10:
        print(f"     ... and {len(classes) - 10} more classes")

print(f"\n☑️ 34-Class experiment completed successfully!")
print(f"💻 Training Device: {'GPU' if gpu_available else 'CPU'}")
print(f"🎯 Total classes: {n_classes}")
```