📖 Complete Code Breakdown - Line by Line

Table of Contents

1️⃣ threat_detector.py - The Main Brain

Part 1: Imports (Lines 1-15)

```python
"""
AI-Powered Threat Detection System with Explainability
"""
```

What it does: This is a docstring - a description of what this file does. It doesn't execute, just documents.

```python
import os
```

What it does: Imports the `os` module for working with files and folders

Example: Creating folders, checking if files exist

```python
import numpy as np
```

What it does: Imports NumPy for math operations on arrays

Think of it as: A super-powered calculator for lots of numbers at once

Example: `np.mean([1,2,3])` calculates average = 2

```python
import pandas as pd
```

What it does: Imports Pandas for working with tables of data (like Excel)

Think of it as: Excel in Python

Example: `pd.read_csv('file.csv')` reads a CSV file into a table

```python
import matplotlib.pyplot as plt
```

What it does: Imports Matplotlib for creating charts and graphs

Think of it as: Making visual charts like bar graphs, line graphs

Example: Creates the confusion matrix image

```python
import seaborn as sns
```

What it does: Imports Seaborn for prettier charts (built on top of Matplotlib)

Think of it as: Matplotlib with automatic nice colors and styling

```python
from sklearn.model_selection import train_test_split
```

What it does: Imports function to split data into training and testing sets

Think of it as: Dividing flashcards into "practice pile" and "test pile"

Example: 70% for learning, 30% for testing

```python
from sklearn.preprocessing import StandardScaler, LabelEncoder
```

What it does: Imports tools to prepare data

- **StandardScaler:** Makes all numbers have similar scale (0-1 range)

- **LabelEncoder:** Converts text labels to numbers

Example: "DDoS" → 1, "BENIGN" → 0

```python
from sklearn.ensemble import RandomForestClassifier
```

What it does: Imports the Random Forest algorithm

Think of it as: A team of 100 decision trees voting together

Analogy: Like asking 100 experts and taking majority vote

```python
from sklearn.metrics import classification_report, confusion_matrix
```

What it does: Imports tools to measure how good the model is

Think of it as: Grading the model's test results

Example: Accuracy, precision, recall scores

```python
import shap
```

What it does: Imports SHAP for explaining predictions

Think of it as: The "why" explainer - shows which features mattered most

Example: "High packet rate (+0.3) made it predict DDoS"

```python
import joblib
```

```

**What it does:** Imports tool to save and load the trained model

**Think of it as:** Saving your trained model like saving a video game

**Example:** Save once, load later without retraining

```python
import warnings
warnings.filterwarnings('ignore')
```

What it does: Turns off warning messages to keep output clean

Think of it as: Muting non-critical notifications

```python
plt.switch_backend('Agg')
```

What it does: Tells Matplotlib to save images to files instead of showing on screen

Why: Works better in scripts that don't have a display (like servers)

Part 2: The ThreatDetectionSystem Class (Lines 17-50)

```python
class ThreatDetectionSystem:
    """
    Main class for network threat detection with explainability
    """
```

What it does: Creates a blueprint (class) for our threat detection system

Think of it as: A recipe that contains all methods for building the detector

Analogy: Like a car blueprint that shows all parts and how they work
```

```python
    def __init__(self, model_type='random_forest'):
```

What it does: Initializes (sets up) the class when you create it

Think of it as: The constructor - runs automatically when you create an instance

Example: `detector = ThreatDetectionSystem()` ← This runs `__init__`

```python
        self.model_type = model_type
        self.model = None
```

What it does:

- `self.model_type` stores which ML algorithm to use (default: random_forest)

- `self.model = None` creates empty placeholder for the trained model

Think of it as: Setting up empty boxes to fill later

```python
        self.scaler = StandardScaler()
        self.label_encoder = LabelEncoder()
```

What it does:

- Creates a scaler object (normalizes feature values)

- Creates label encoder (converts text to numbers)

Example:

  - Scaler: 1000 → 0.5, 2000 → 1.0 (normalized)

  - Encoder: "DDoS" → 1, "BENIGN" → 0

```python
        self.feature_names = None
        self.explainer = None
```

```

**What it does:** Creates placeholders for:

- Feature names (list of column names)

- SHAP explainer object

**Think of it as:** Empty variables to fill during training

---

## Part 3: Load and Preprocess Data Method (Lines 52-85)

```python
    def load_and_preprocess_data(self, filepath):
        """
        Load CICIDS2017 dataset and perform preprocessing
        """
```

What it does: Defines a method to load CSV file and clean it

Parameters: `filepath` = location of CSV file

Returns: Cleaned dataframe and label column name

```python
        print("[*] Loading dataset...")
        print(f"[*] Reading file: {filepath}")
```

What it does: Prints status messages so user knows what's happening

Think of it as: Progress updates

Output: `[*] Loading dataset...`

```python
        if not os.path.exists(filepath):
            raise FileNotFoundError(f"Dataset not found at: {filepath}")

```
What it does: Checks if file exists; if not, shows error

Think of it as: Safety check before trying to open file

Example: If you typed wrong path, it tells you immediately

```python
    df = pd.read_csv(filepath, encoding='utf-8')
```

What it does: Reads CSV file into a Pandas DataFrame (table)

Think of it as: Opening an Excel file

Result: `df` contains all rows and columns from CSV

Visual representation:

```
CSV File:            DataFrame (df):
┌──────────┬────┐    ┌──────────┬────┐
│ Port│Size│Label│    │ Port│Size│Label│
├──────────┼────┤    ├──────────┼────┤
│  80 │500 │BENIGN│   │  80 │500 │BENIGN│
│  443│600 │DDoS │    │  443│600 │DDoS │
│  80 │550 │BENIGN│   │  80 │550 │BENIGN│
└──────────┴────┘    └──────────┴────┘
```

```python
    print(f"[+] Dataset shape: {df.shape}")
```

What it does: Prints how many rows and columns

Example output: `[+] Dataset shape: (225745, 79)` = 225,745 rows, 79 columns

```python

```
    df = df.replace([np.inf, -np.inf], np.nan)
```

What it does: Replaces infinity values with NaN (Not a Number)

Why: Math errors can create infinity; we need to handle them

Example: Division by zero → infinity → NaN

```python
    df = df.fillna(0)
```

What it does: Replaces all NaN (missing) values with 0

Why: Machine learning models can't handle missing data

Example: If "Packet Size" is missing → set to 0

```python
    df = df.drop_duplicates()
```

What it does: Removes exact duplicate rows

Why: Duplicates can bias the model

Example: If same row appears 10 times, model thinks it's very important

```python
    df.columns = df.columns.str.strip()
```

What it does: Removes extra spaces from column names

Example: `" Destination Port "` → `"Destination Port"`

```python
    label_col = 'Label' if 'Label' in df.columns else df.columns[-1]
```

What it does: Finds which column contains the labels (answers)

Logic:

- If column named "Label" exists, use it
- Otherwise, use the last column

Result: `label_col = "Label"`

```python
        print(f"[+] Label column: {label_col}")
        print(f"[+] Attack types distribution:")
        print(df[label_col].value_counts())
```

What it does: Shows how many of each attack type

Example output:

```
[+] Label column: Label
[+] Attack types distribution:
BENIGN    150000
DDoS      45000
PortScan   15000
```

```python
        return df, label_col
```

What it does: Sends back the cleaned dataframe and label column name

Think of it as: Function's output that other code can use

Part 4: Feature Engineering Method (Lines 87-110)

```python
    def feature_engineering(self, df, label_col):
```

    """
    Extract and engineer features from raw data
    """
```

**What it does:** Separates data into features (X) and labels (y)

**Think of it as:** Separating questions from answers

```python
    print("\n[*] Performing feature engineering...")
```

What it does: Prints status update

The `\n`: Creates a blank line before printing

```python
    X = df.drop(columns=[label_col])
    y = df[label_col]
```

What it does:

- `X` = All columns EXCEPT the label (the features/inputs)
- `y` = Only the label column (the answers/outputs)

Visual:

```
Original DataFrame:
┌────┬────┬────┬────────┐
│Port│Size│IAT │Label   │
├────┼────┼────┼────────┤
│ 80 │500 │100 │BENIGN │  → X: Port, Size, IAT
│ 443│600 │50  │DDoS   │  → y: BENIGN, DDoS, BENIGN
│ 80 │550 │120 │BENIGN │
└────┴────┴────┴────────┘

```

```python
    numeric_cols = X.select_dtypes(include=[np.number]).columns
    X = X[numeric_cols]
```

What it does: Keeps only numeric columns (removes text columns)

Why: ML models need numbers, not text

Example: Keeps "Port: 80" but removes "Protocol: TCP"

```python
    self.feature_names = X.columns.tolist()
```

What it does: Saves column names in a list for later use

Example: `['Destination Port', 'Flow Duration', 'Total Fwd Packets', ...]`

```python
    y_encoded = self.label_encoder.fit_transform(y)
```

What it does: Converts text labels to numbers

Process:

1. `fit`: Learn the unique labels (BENIGN, DDoS, PortScan)

2. `transform`: Convert them to numbers

Example:

```
Before:          After:
['BENIGN']    →    [0]
['DDoS']      →    [1]
['BENIGN']    →    [0]
['PortScan']  →    [2]

```

```python
        print(f"[+] Features extracted: {len(self.feature_names)}")
        print(f"[+] Classes: {list(self.label_encoder.classes_)}")
```
What it does: Shows how many features and what classes exist

Example output:
```
[+] Features extracted: 77
[+] Classes: ['BENIGN', 'DDoS', 'PortScan']
```

```python
        return X, y_encoded
```
What it does: Returns the features (X) and encoded labels (y)

Part 5: Train Model Method (Lines 112-145)

```python
    def train_model(self, X_train, y_train, n_estimators=100):
        """
        Train the threat detection model
        """
```
What it does: Trains the Random Forest model on the training data

Parameters:

- `X_train`: Training features
```

- `y_train`: Training labels

- `n_estimators=100`: Number of decision trees (default 100)

```python
    print("\n[*] Training Random Forest model...")
    print(f"[*] Training samples: {len(X_train)}")
```

What it does: Shows training progress

Example: `[*] Training samples: 157021`

```python
    X_train_scaled = self.scaler.fit_transform(X_train)
```

What it does: Normalizes the training data

Process:

1. `fit`: Learn the mean and std of each feature

2. `transform`: Scale values to standard range

Example:

```
Before scaling:        After scaling:
Packet Size: 1500  →  0.5
Packet Size: 3000  →  1.0
Packet Size: 750   →  0.25

Formula: (value - mean) / std
```

Why scale?

- Feature "Bytes" might be 1,000,000

- Feature "Packets" might be 10

- Without scaling, model focuses only on large numbers
- After scaling, all features have equal importance

```python
        self.model = RandomForestClassifier(
            n_estimators=n_estimators,
            max_depth=20,
            min_samples_split=10,
            random_state=42,
            n_jobs=-1,
            class_weight='balanced',
            verbose=0
        )
```
What it does: Creates the Random Forest model with specific settings

Each parameter explained:

- `n_estimators=100`: Create 100 decision trees
  - More trees = more accurate but slower
  - Like asking 100 experts instead of 1

- `max_depth=20`: Each tree can be maximum 20 levels deep
  - Prevents overfitting (memorizing training data)
  - Like limiting how detailed questions can get

- `min_samples_split=10`: Need at least 10 samples to split a node
  - Prevents tiny splits on few examples
  - Ensures decisions are based on enough data

- `random_state=42`: Random seed for reproducibility
  - Using 42 always gives same random results

  - Makes experiments repeatable

- `n_jobs=-1`: Use all CPU cores for parallel processing
  - Trains faster by using multiple processors
  - -1 means "use all available cores"

- `class_weight='balanced'`: Handle imbalanced data
  - If 90% BENIGN, 10% DDoS, model might always predict BENIGN
  - This balances importance of rare classes

- `verbose=0`: Don't print training progress details

How Random Forest works:
```
Tree 1: Port=80? → Size>500? → Predict: DDoS
Tree 2: Size>600? → IAT<100? → Predict: DDoS
Tree 3: Port=443? → Size>550? → Predict: BENIGN
...
Tree 100: (makes its prediction)

Final Prediction: Majority vote
- 70 trees say DDoS
- 30 trees say BENIGN
→ Prediction: DDoS (70%)
```

```python
        self.model.fit(X_train_scaled, y_train)
```
What it does: Actually trains the model (the learning happens here)
Process:

1. Model looks at features and labels
2. Finds patterns (e.g., high packet rate → DDoS)
3. Builds 100 decision trees
4. Each tree learns slightly different patterns

What's happening inside:
```
For each tree:
  1. Randomly select features
  2. Find best way to split data
  3. Keep splitting until max_depth
  4. Store the tree structure
```

Time: Takes 2-5 minutes depending on data size

```python
        print("[+] Model training completed")
        return X_train_scaled
```
What it does: Confirms training finished and returns scaled data

---

Part 6: Evaluate Model Method (Lines 147-190)

```python
    def evaluate_model(self, X_test, y_test):
        """
        Evaluate model performance
        """
```

```

**What it does:** Tests how accurate the model is on unseen data

```python
    print("\n[*] Evaluating model...")
    X_test_scaled = self.scaler.transform(X_test)
```

What it does: Scales test data using SAME scaling as training

Important: Use `transform` (not `fit_transform`)

Why: We scale test data the same way as training data

Example:
```
Training: Mean=1000, Std=200
Test value: 1200
Scaled: (1200 - 1000) / 200 = 1.0

We DON'T recalculate mean/std for test data!
```

```python
    y_pred = self.model.predict(X_test_scaled)
    y_pred_proba = self.model.predict_proba(X_test_scaled)
```

What it does:

- `y_pred`: Predicted class (0, 1, 2)
- `y_pred_proba`: Probability for each class

Example:
```
Input: Network flow features

y_pred: 1 (DDoS)

y_pred_proba: [0.05, 0.90, 0.05]

            BENIGN DDoS PortScan

Interpretation: 90% confident it's DDoS
```

```python
       print("\n" + "="*60)
       print("CLASSIFICATION REPORT")
       print("="*60)
       print(classification_report(y_test, y_pred,
                              target_names=self.label_encoder.classes_))
```
What it does: Prints detailed accuracy metrics

Example output:
```
============================================================
CLASSIFICATION REPORT
============================================================
            precision    recall  f1-score   support

      BENIGN       0.99      0.98      0.99     45000
        DDoS       0.97      0.99      0.98     13500
    PortScan       0.96      0.95      0.96      4500

    accuracy                           0.98     63000
```

What each metric means:
```

- **Precision:** When model says "DDoS", how often is it right?
  - Formula: True Positives / (True Positives + False Positives)
  - Example: Predicted 100 as DDoS, 97 actually were = 97% precision

- **Recall:** Of all actual DDoS attacks, how many did we catch?
  - Formula: True Positives / (True Positives + False Negatives)
  - Example: 100 real DDoS, caught 99 = 99% recall

- **F1-Score:** Balance between precision and recall
  - Formula: 2 * (Precision * Recall) / (Precision + Recall)
  - Example: Good when both precision and recall are high

- **Support:** Number of actual examples in test set

```python
    cm = confusion_matrix(y_test, y_pred)
```
What it does: Creates confusion matrix (table showing predictions vs reality)

Visual explanation:
```
Confusion Matrix:
              Predicted
             BENIGN  DDoS  PortScan
Actual BENIGN [44100]  [600]    [300]     ← 44,100 correctly predicted
     DDoS    [  200][13300]   [    0]    ← 13,300 correctly predicted
     PortScan[  100][   50]   [4350]    ← 4,350 correctly predicted

Diagonal = Correct predictions
Off-diagonal = Mistakes
```

```python
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
        xticklabels=self.label_encoder.classes_,
        yticklabels=self.label_encoder.classes_)
```

What it does: Creates visual heatmap of confusion matrix

Parameters:

- `figsize=(10, 8)`: Size of image in inches
- `annot=True`: Show numbers in cells
- `fmt='d'`: Format numbers as integers (not decimals)
- `cmap='Blues'`: Color scheme (blue gradient)
- `xticklabels/yticklabels`: Labels for axes

```python
plt.title('Threat Detection Confusion Matrix')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.tight_layout()
```

What it does: Adds title and axis labels, adjusts spacing

```python
output_path = 'outputs\\confusion_matrix.png'
os.makedirs('outputs', exist_ok=True)
plt.savefig(output_path, dpi=300, bbox_inches='tight')
plt.close()
```

What it does:

1. Sets output filename

2. Creates `outputs` folder if it doesn't exist

3. Saves image at 300 DPI (high quality)

4. Closes the plot to free memory

Parameters explained:

- `dpi=300`: Dots per inch (higher = better quality)

- `bbox_inches='tight'`: Remove extra whitespace

- `plt.close()`: Prevents memory leaks

```python
        print(f"[+] Confusion matrix saved to '{output_path}'")
```

What it does: Confirms file was saved

```python
        self.plot_feature_importance()
        return X_test_scaled, y_pred, y_pred_proba
```

What it does:

1. Calls method to plot feature importance

2. Returns scaled test data and predictions

Part 7: Feature Importance Method (Lines 192-215)

```python
    def plot_feature_importance(self, top_n=20):
        """
        Visualize feature importance
        """
```

```

**What it does:** Shows which features matter most for predictions

```python
    if hasattr(self.model, 'feature_importances_'):
```

What it does: Checks if model has feature importance scores

Why: Only tree-based models have this attribute

```python
        importances = self.model.feature_importances_
        indices = np.argsort(importances)[-top_n:]
```

What it does:

- Gets importance score for each feature
- Finds top 20 most important features
- `np.argsort`: Returns indices that would sort the array
- `[-top_n:]`: Takes last 20 (highest values)

Example:

```
Features:       ['Port', 'Size', 'Duration', 'IAT']
Importances:    [0.05,   0.30,   0.15,      0.40]
Sorted indices: [0,      2,      1,         3]
Top 2:          [1,      3]  (Size and IAT)
```

```python
        plt.figure(figsize=(10, 8))
        plt.barh(range(len(indices)), importances[indices], color='steelblue')
```
```

What it does: Creates horizontal bar chart

- `barh`: Horizontal bars (easier to read long names)

- `range(len(indices))`: Y-axis positions (0, 1, 2, ...)

- `importances[indices]`: X-axis values (importance scores)

```python
        plt.yticks(range(len(indices)),
                   [self.feature_names[i] for i in indices])
```

What it does: Labels Y-axis with feature names

List comprehension breakdown:

```python
[self.feature_names[i] for i in indices]

Step by step:
for i in indices:  # For each index
    self.feature_names[i]  # Get feature name at that index
    # Add to list

Result: ['Feature 1', 'Feature 2', ...]
```

```python
        plt.xlabel('Feature Importance')
        plt.title(f'Top {top_n} Most Important Features')
        plt.tight_layout()
```

What it does: Adds labels and adjusts layout

```python
        output_path = 'outputs\\feature_importance.png'
```

        plt.savefig(output_path, dpi=300, bbox_inches='tight')
        plt.close()
        print(f"[+] Feature importance saved to '{output_path}'")
```
**What it does:** Saves chart to file and confirms

---

## Part 8: SHAP Explainability (Lines 217-280)

```python
    def initialize_explainer(self, X_train_scaled):
        """
        Initialize SHAP explainer for model interpretability
        """
```
What it does: Sets up SHAP to explain model predictions

```python
        print("\n[*] Initializing SHAP explainer...")
        self.explainer = shap.TreeExplainer(self.model)
```
What it does: Creates SHAP explainer for Random Forest
TreeExplainer: Optimized for tree-based models (fast and accurate)

How SHAP works:
```
For each prediction:
1. Calculate baseline (average prediction)
2. For each feature:
   - See how prediction changes with/without it

   - Calculate contribution (SHAP value)
3. Sum all contributions = Final prediction

Example:
Baseline: 50% threat
+ Packet rate (+30%)
+ Port 80 (+10%)
- Normal size (-5%)
= 85% threat (DDoS)
```

```python
        print("[+] SHAP explainer initialized")
        return self.explainer
```
What it does: Confirms setup and returns explainer

```python
    def explain_prediction(self, sample_index, X_test_scaled, y_test, y_pred):
```
What it does: Explains why model made a specific prediction
Parameters:
- `sample_index`: Which test sample to explain
- `X_test_scaled`: Scaled test features
- `y_test`: True labels
- `y_pred`: Predicted labels

```python
        print(f"\n[*] Explaining prediction for sample {sample_index}...")
        sample = X_test_scaled[sample_index:sample_index+1]
```
```

What it does:

- Prints status

- Gets specific sample (row) from test data

- `[sample_index:sample_index+1]`: Slicing to keep 2D shape

Why [index:index+1]?

```python
X[0]      → 1D array [1.2, 0.5, 3.1]
X[0:1]    → 2D array [[1.2, 0.5, 3.1]]

Model needs 2D input (even for 1 sample)
```

```python
        shap_values = self.explainer.shap_values(sample)
```

What it does: Calculates SHAP values for this sample

Output: Array showing contribution of each feature

Example:

```
Features:     ['Port', 'Size', 'Duration', 'IAT']
Values:       [  80,    1500,   120000,    50  ]
SHAP values:  [+0.1,   +0.3,   -0.05,    +0.4 ]

Interpretation:
- Port contributes +0.1 to threat probability
- Size contributes +0.3 (important!)
- Duration reduces threat by -0.05
- IAT contributes +0.4 (most important!)
```

```python
        true_label = self.label_encoder.classes_[y_test[sample_index]]
        pred_label = self.label_encoder.classes_[y_pred[sample_index]]
```

What it does: Converts numeric predictions back to text labels

Example:

```
y_test[0] = 1  →  true_label = "DDoS"
y_pred[0] = 1  →  pred_label = "DDoS"
```

```python
        print(f"\n{'='*60}")
        print("PREDICTION EXPLANATION")
        print(f"{'='*60}")
        print(f"True Label: {true_label}")
        print(f"Predicted Label: {pred_label}")
        print(f"Prediction Correct: {true_label == pred_label}")
```

What it does: Prints comparison of true vs predicted label

Example output:

```
============================================================
PREDICTION EXPLANATION
============================================================
True Label: DDoS
Predicted Label: DDoS
Prediction Correct: True
```

```python
    if isinstance(shap_values, list) and len(shap_values) > 0:
        shap_vals = shap_values[y_pred[sample_index]]
    else:
        shap_vals = shap_values
```

What it does: Handles multi-class SHAP values

Explanation:

- For binary: SHAP returns single array
- For multi-class: SHAP returns list of arrays (one per class)
- We select array for the predicted class

```python
    try:
        plt.figure(figsize=(10, 6))
        expected_val = (self.explainer.expected_value[y_pred[sample_index]]
                       if isinstance(self.explainer.expected_value, np.ndarray)
                       else self.explainer.expected_value)
```

What it does:

- Creates new figure
- Gets baseline/expected value (average prediction)
- Handles both array and single value cases

```python
        shap.waterfall_plot(
            shap.Explanation(
                values=shap_vals[0],
                base_values=expected_val,
                data=sample[0],
                feature_names=self.feature_names
```

    ),

    show=False

)

```

**What it does:** Creates SHAP waterfall plot

**Components:**

- `values`: SHAP contributions for each feature

- `base_values`: Starting point (average)

- `data`: Actual feature values

- `feature_names`: Names for labels

- `show=False`: Don't display, just prepare to save

**Waterfall plot visualization:**

```

E[f(X)] = 0.5 (baseline)

   ↓ +0.3 (Packet rate high)

   ↓ +0.1 (Port 80)

   ↓ -0.05 (Normal duration)

   ↓ +0.2 (Low IAT)

f(x) = 1.05 → DDoS (105% probability, capped at 100%)

```

```python

  plt.tight_layout()

  output_path = f'outputs\\explanation_sample_{sample_index}.png'

  plt.savefig(output_path, dpi=300, bbox_inches='tight')

  plt.close()

  print(f"[+] Explanation saved to '{output_path}'")

```

What it does: Saves the waterfall plot
```

```python
        except Exception as e:
            print(f"[!] Could not generate waterfall plot: {e}")
```

What it does: Catches and reports any errors

Why: SHAP plotting can sometimes fail; we don't want to crash

```python
        return shap_vals
```

What it does: Returns SHAP values for further analysis

Part 9: Save/Load Model (Lines 282-320)

```python

<function_calls> <invoke name="artifacts"> <parameter name="command">update</parameter> <parameter name="id">code_explanation_guide</parameter> <parameter name="old_str">```python </parameter> <parameter name="new_str">def save_model(self, filepath='threat_detector.pkl'): """ Save trained model and preprocessing objects """

**What it does:** Saves everything needed to use the model later

**Parameter:** `filepath` = where to save (default: threat_detector.pkl)

```python
        model_data = {
            'model': self.model,
            'scaler': self.scaler,
            'label_encoder': self.label_encoder,
            'feature_names': self.feature_names
        }

```

What it does: Creates dictionary with all components

Why save all these?

- `model`: The trained Random Forest

- `scaler`: To scale new data the same way

- `label_encoder`: To convert predictions back to text

- `feature_names`: To know feature order

Think of it as: Packing everything into a suitcase

```python
        joblib.dump(model_data, filepath)
```

What it does: Serializes (saves) the dictionary to disk

File format: .pkl (pickle) - compressed binary format

Size: Usually 10-50 MB depending on model

What's inside the .pkl file:

threat_detector.pkl

├── Random Forest (100 trees)

├── StandardScaler (mean/std for each feature)

├── LabelEncoder (class mappings)

└── Feature names (list of strings)

```python
        print(f"[+] Model saved to '{filepath}'")
```

What it does: Confirms save completed

```python
    def load_model(self, filepath='threat_detector.pkl'):
        """

    Load trained model
    """
```

What it does: Loads previously saved model

```python
    if not os.path.exists(filepath):
        raise FileNotFoundError(f"Model file not found: {filepath}")
```

What it does: Safety check - make sure file exists before trying to load

```python
    model_data = joblib.load(filepath)
```

What it does: Deserializes (loads) the dictionary from disk

Time: Fast, usually < 1 second

```python
    self.model = model_data['model']
    self.scaler = model_data['scaler']
    self.label_encoder = model_data['label_encoder']
    self.feature_names = model_data['feature_names']
```

What it does: Unpacks dictionary and assigns to class attributes

Think of it as: Unpacking the suitcase

```python
    print(f"[+] Model loaded from '{filepath}'")
```

What it does: Confirms load completed

Usage example:

```python
# Save after training
detector.save_model('my_model.pkl')

# Later, in a different script:
detector = ThreatDetectionSystem()
detector.load_model('my_model.pkl')
# Now ready to make predictions without retraining!
```

Part 10: Main Execution Function (Lines 322-400)
```python
def main():
    """
    Main execution pipeline
    """
```
What it does: Orchestrates the entire training process
Think of it as: The conductor of an orchestra
```python
    print("="*60)
    print("AI-POWERED THREAT DETECTION SYSTEM")
    print("Windows Edition")
    print("="*60)
```
What it does: Prints nice header banner
```python
    detector = ThreatDetectionSystem(model_type='random_forest')
```
What it does: Creates instance of our class
Equivalent to:
```python

detector = ThreatDetectionSystem()  # Using default

```

What happens: Runs `__init__` method, sets up empty model

```python
    filepath = r'data\raw\Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv'
```

What it does: Sets path to dataset

The `r` before string: Raw string (treats backslashes literally)

Why: In Windows, paths use backslashes

```python
    try:
        df, label_col = detector.load_and_preprocess_data(filepath)
    except FileNotFoundError:
        print("\n[!] Dataset not found!")
        print("[!] Please:")
        print("    1. Download CICIDS2017")
        print("    2. Place CSV file in: data\\raw\\")
        print("    3. Or run: python src\\generate_sample_data.py")
        return
```

What it does: Tries to load data; if fails, shows helpful error

try/except: Error handling

- `try`: Attempt this code

- `except`: If error occurs, do this instead

- `return`: Exit function early

Flow:

Try to load data

↓

Success? → Continue

↓
```

Failure? → Show error message → Exit

```python
    if len(df) > 100000:
        print(f"\n[*] Large dataset detected. Sampling 100,000 rows...")
        df = df.sample(n=100000, random_state=42)
```
What it does: If dataset is huge, use smaller sample

Why: Training on 2.8M rows takes hours; 100K is faster for testing

`random_state=42`: Ensures same sample every time

```python
    X, y = detector.feature_engineering(df, label_col)
```
What it does: Separates features and labels

Returns:

- `X`: DataFrame with features
- `y`: Array with encoded labels

```python
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=42, stratify=y
    )
```
What it does: Splits data into training (70%) and testing (30%) sets

Parameters:

- `test_size=0.3`: 30% for testing
- `random_state=42`: Reproducible split
- `stratify=y`: Maintain class distribution in both sets

Stratify explanation:

Original data:

70% BENIGN, 30% DDoS

Without stratify:

Train: 80% BENIGN, 20% DDoS ← Unbalanced!

Test: 50% BENIGN, 50% DDoS

With stratify:

Train: 70% BENIGN, 30% DDoS ← Balanced!

Test: 70% BENIGN, 30% DDoS

```python
   print(f"\n[+] Training set: {X_train.shape}")
   print(f"[+] Test set: {X_test.shape}")
```

What it does: Shows split sizes

Example: `[+] Training set: (70000, 77)`

```python
   X_train_scaled = detector.train_model(X_train, y_train, n_estimators=100)
```

What it does: Trains the model

What's happening:

1. Scales training data

2. Creates Random Forest with 100 trees

3. Fits model to training data

4. Returns scaled data

Timeline:

0% Training...

25% Building trees...

50% Halfway...

75% Almost done...

100% Complete!

```python
   X_test_scaled, y_pred, y_pred_proba = detector.evaluate_model(X_test, y_test)
```

What it does: Tests model accuracy

Returns:

- `X_test_scaled`: Scaled test features
- `y_pred`: Predicted classes
- `y_pred_proba`: Prediction probabilities

```python
   detector.initialize_explainer(X_train_scaled)
```

What it does: Sets up SHAP for explanations

```python
   detector.explain_prediction(0, X_test_scaled, y_test, y_pred)
   detector.explain_prediction(10, X_test_scaled, y_test, y_pred)
```

What it does: Explains 2 example predictions (samples 0 and 10)

Why 2? Shows variety - one might be correct, one incorrect

```python
   detector.generate_shap_summary(X_test_scaled)
```

What it does: Creates global SHAP summary plot

```python
   detector.save_model('threat_detector.pkl')
```

What it does: Saves trained model to disk

Result: File `threat_detector.pkl` created in root folder

```python
    print("\n" + "="*60)
    print("ANALYSIS COMPLETE")
    print("="*60)
    print("\nGenerated files:")
    print("  - threat_detector.pkl (trained model)")
    print("  - outputs\\confusion_matrix.png")
    print("  - outputs\\feature_importance.png")
    print("  - outputs\\explanation_sample_*.png")
    print("  - outputs\\shap_summary.png")
```

What it does: Shows summary of what was created

```python
    print("\nNext step: Run the dashboard")
    print("  python src\\dashboard.py")
```

What it does: Tells user what to do next

```python
if __name__ == "__main__":
    main()
```

What it does: Runs main() only when script is executed directly

Why needed: Allows importing functions without running main()

Explanation:

```python
# When you run: python threat_detector.py
__name__ == "__main__"  → True → Runs main()

# When you import: from threat_detector import ThreatDetectionSystem
__name__ == "threat_detector"  → False → Doesn't run main()
```

```

---

This completes the line-by-line explanation of `threat_detector.py`!

**Summary of what this file does:**

1. ✅ Loads and cleans network traffic data
2. ✅ Trains Random Forest classifier
3. ✅ Evaluates model performance
4. ✅ Generates SHAP explanations
5. ✅ Creates visualizations (charts/plots)
6. ✅ Saves trained model to disk

**Next file:** `dashboard.py` - The web application

```</parameter>