## Project Title

ML-Assisted Lithography Hotspot Detection System

---

## Team members

Chetna - 124113046

Cherag - 124113047

Aayush - 124113058

Chetan - 124113060

---

## Acknowledgement

We express our gratitude to our faculty mentors and project colleagues for their continuous support and guidance. Special thanks to the creators and contributors of the Intelectron6 Lithography Hotspot Detection repository and the ICCAD-2012 benchmark dataset, whose resources were instrumental for our research and model development.

---

## Abstract

Lithography hotspots are dangerous manufacturing defects that reduce reliability and yield in modern integrated circuits. This project presents a scalable machine learning workflow using the ICCAD-2012 benchmark PNG dataset. Data processing is optimized for large-scale memory efficiency, advanced feature extraction is performed on each layout pattern, and ensemble-based ML models provide reliable hotspot identification under strong class imbalance.

---

## Introduction

As semiconductor devices continue to scale down, detecting lithography-induced defects becomes increasingly critical for achieving design closure and maximizing manufacturing yield. Machine learning-based hotspot detection has emerged as a fast, scalable, and highly effective alternative to traditional simulation-based methods. Leveraging advanced algorithms, particularly deep learning techniques such as convolutional neural networks, these ML models can capture complex patterns and relationships within intricate IC design data that conventional rule-based or statistical methods often miss. By training on diverse, large datasets, machine learning approaches improve accuracy in identifying potential hotspots while reducing false positives, leading to enhanced yield and reliabilityThe ICCAD-2012 benchmark provides comprehensive layout patterns and hotspot annotations suitable for model training and evaluation in this domain.

---

## Existing System

Traditional approaches rely on simulation or pattern matching, which are slow and resource-consuming. Previous machine learning works used simple classifiers or CNNs on extracted features from benchmark datasets, offering improvements in recall but often suffering from low precision due to severe class imbalance and overfitting.

---

## Proposed System / Methodology

## Data batch processing

The PNG dataset from ICCAD-2012 is processed in benchmark-specific batches to ensure out-of-memory safety. Patterns are loaded and processed chunk-wise, with memory-efficient storage and feature extraction using custom scripts.

## Feature extraction

Each layout pattern undergoes advanced feature engineering, including density metrics, topological analysis, spatial statistics, gradient, geometric, complexity, and frequency domain measures.

python

```
features.update(self._extract_density_features(pattern))
features.update(self._extract_topology_features(pattern))
features.update(self._extract_spatial_features(pattern))
```

## Model training and evaluation

A composite ensemble classifier combines Random Forest, Logistic Regression, and K-Nearest Neighbors. Features are standardized for training. Evaluation is conducted on stratified splits that preserve class imbalance using accuracy, precision, recall, F1-score, and false alarm rate.

python

```
pipeline.train_models(X_train_scaled, y_train)
```

```
metrics = pipeline.evaluate_model(X_test_scaled, y_test)
pipeline.save_model(filepath='models/iccad_hotspot_detector.pkl')
```

---

## Implementation Details

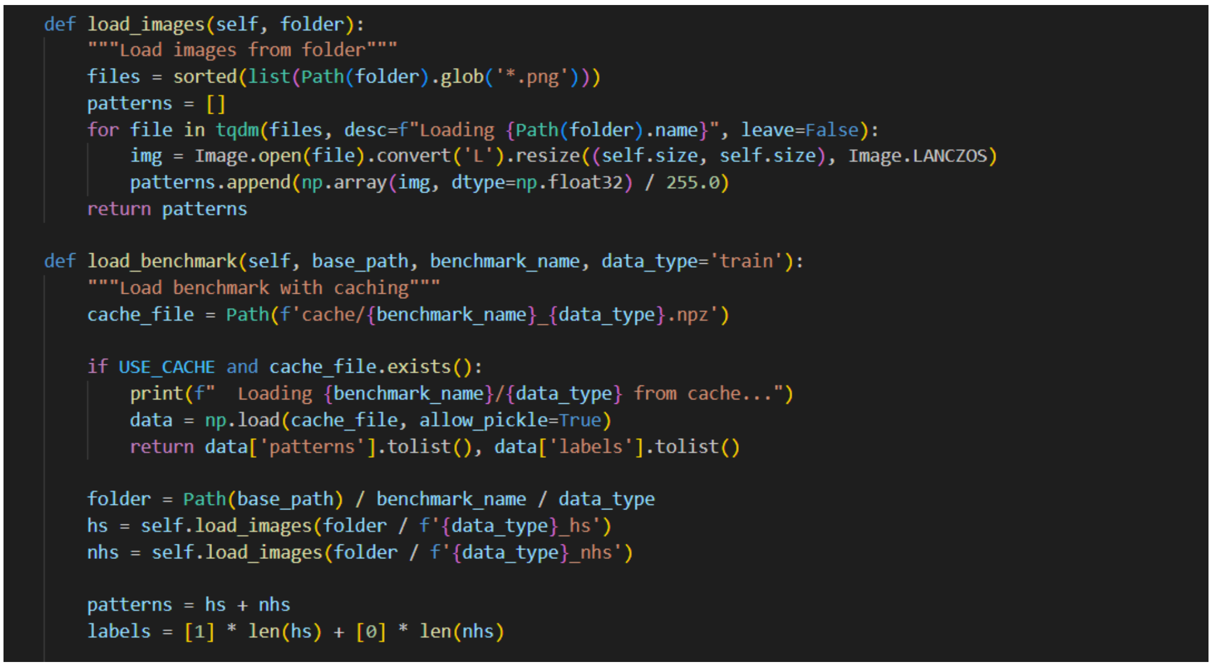
- Benchmark datasets are loaded and processed in batches.
- Imbalance is handled by weighting classes in all models.
- Predictive features are engineered for each pattern using spatial, topological, and complexity analysis.
- Ensembles are trained and optimized by grid search and cross-validation.
- Metrics and visualizations include confusion matrices, ROC and PR curves, and feature distributions.

---

## Code Workflow

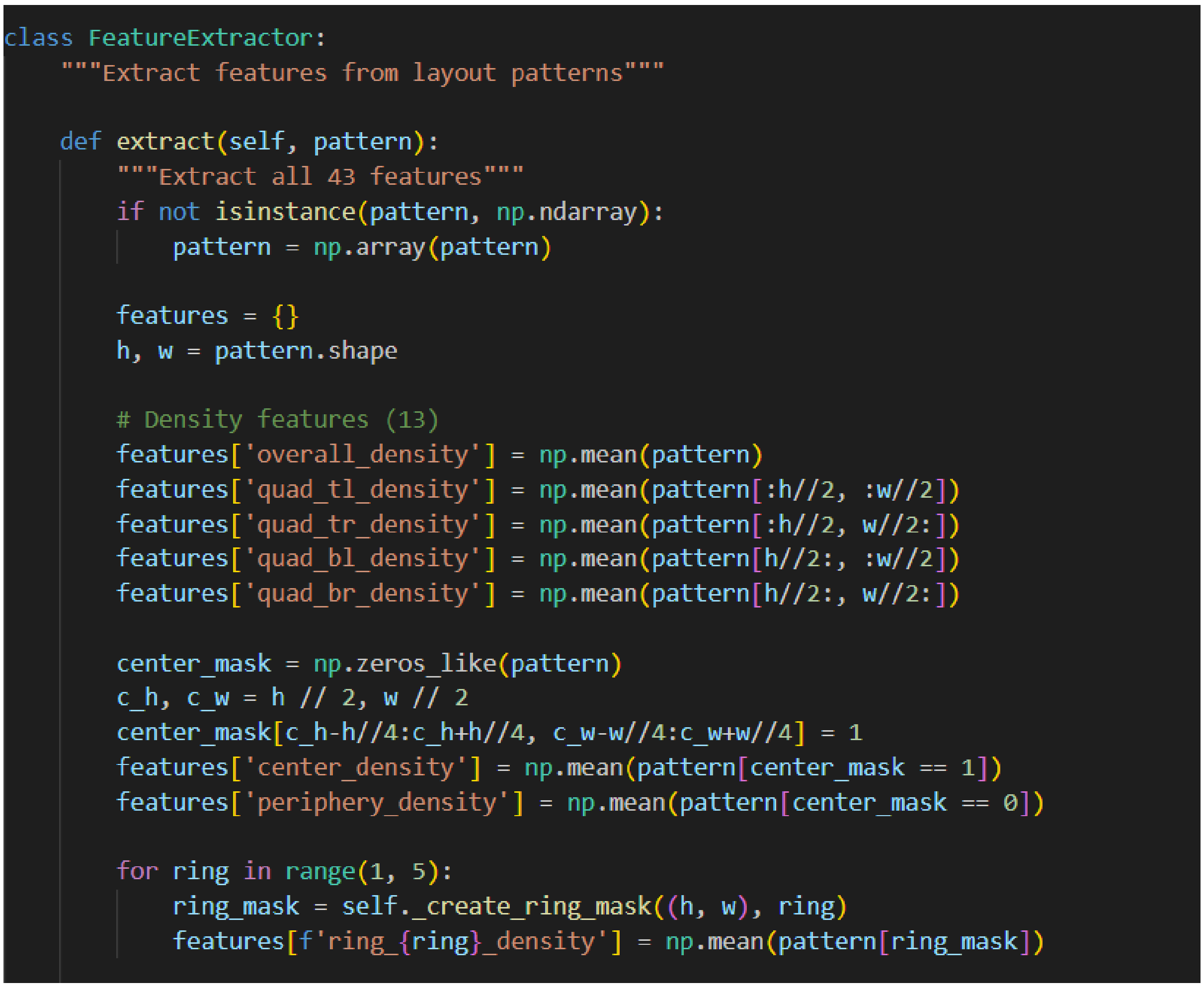
Here are 5 important code snippets that explain the core procedure and steps of this lithographic hotspot detection system:

### 1. Data Loading and Caching



```python
def load_images(self, folder):
    """Load images from folder"""
    files = sorted(list(Path(folder).glob('*.png')))
    patterns = []
    for file in tqdm(files, desc=f"Loading {Path(folder).name}", leave=False):
        img = Image.open(file).convert('L').resize((self.size, self.size), Image.LANCZOS)
        patterns.append(np.array(img, dtype=np.float32) / 255.0)
    return patterns

def load_benchmark(self, base_path, benchmark_name, data_type='train'):
    """Load benchmark with caching"""
    cache_file = Path(f'cache/{benchmark_name}_{data_type}.npz')

    if USE_CACHE and cache_file.exists():
        print(f"  Loading {benchmark_name}/{data_type} from cache...")
        data = np.load(cache_file, allow_pickle=True)
        return data['patterns'].tolist(), data['labels'].tolist()

    folder = Path(base_path) / benchmark_name / data_type
    hs = self.load_images(folder / f'{data_type}_hs')
    nhs = self.load_images(folder / f'{data_type}_nhs')

    patterns = hs + nhs
    labels = [1] * len(hs) + [0] * len(nhs)
```

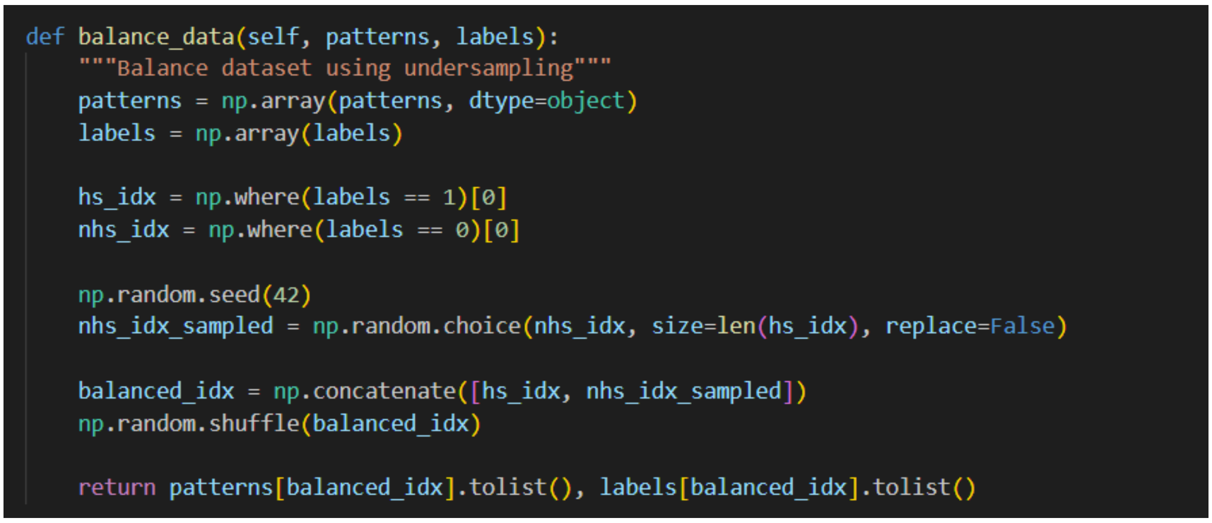
### 2. Feature Extraction (43 Features)



```python
class FeatureExtractor:
    """Extract features from layout patterns"""

    def extract(self, pattern):
        """Extract all 43 features"""
        if not isinstance(pattern, np.ndarray):
            pattern = np.array(pattern)

        features = {}
        h, w = pattern.shape

        # Density features (13)
        features['overall_density'] = np.mean(pattern)
        features['quad_tl_density'] = np.mean(pattern[:h//2, :w//2])
        features['quad_tr_density'] = np.mean(pattern[:h//2, w//2:])
        features['quad_bl_density'] = np.mean(pattern[h//2:, :w//2])
        features['quad_br_density'] = np.mean(pattern[h//2:, w//2:])

        center_mask = np.zeros_like(pattern)
        c_h, c_w = h // 2, w // 2
        center_mask[c_h-h//4:c_h+h//4, c_w-w//4:c_w+w//4] = 1
        features['center_density'] = np.mean(pattern[center_mask == 1])
        features['periphery_density'] = np.mean(pattern[center_mask == 0])

        for ring in range(1, 5):
            ring_mask = self._create_ring_mask((h, w), ring)
            features[f'ring_{ring}_density'] = np.mean(pattern[ring_mask])
```

This extracts 43 engineered features from layout patterns covering density, topology, spatial, and geometric characteristics.

## 3. Data Balancing

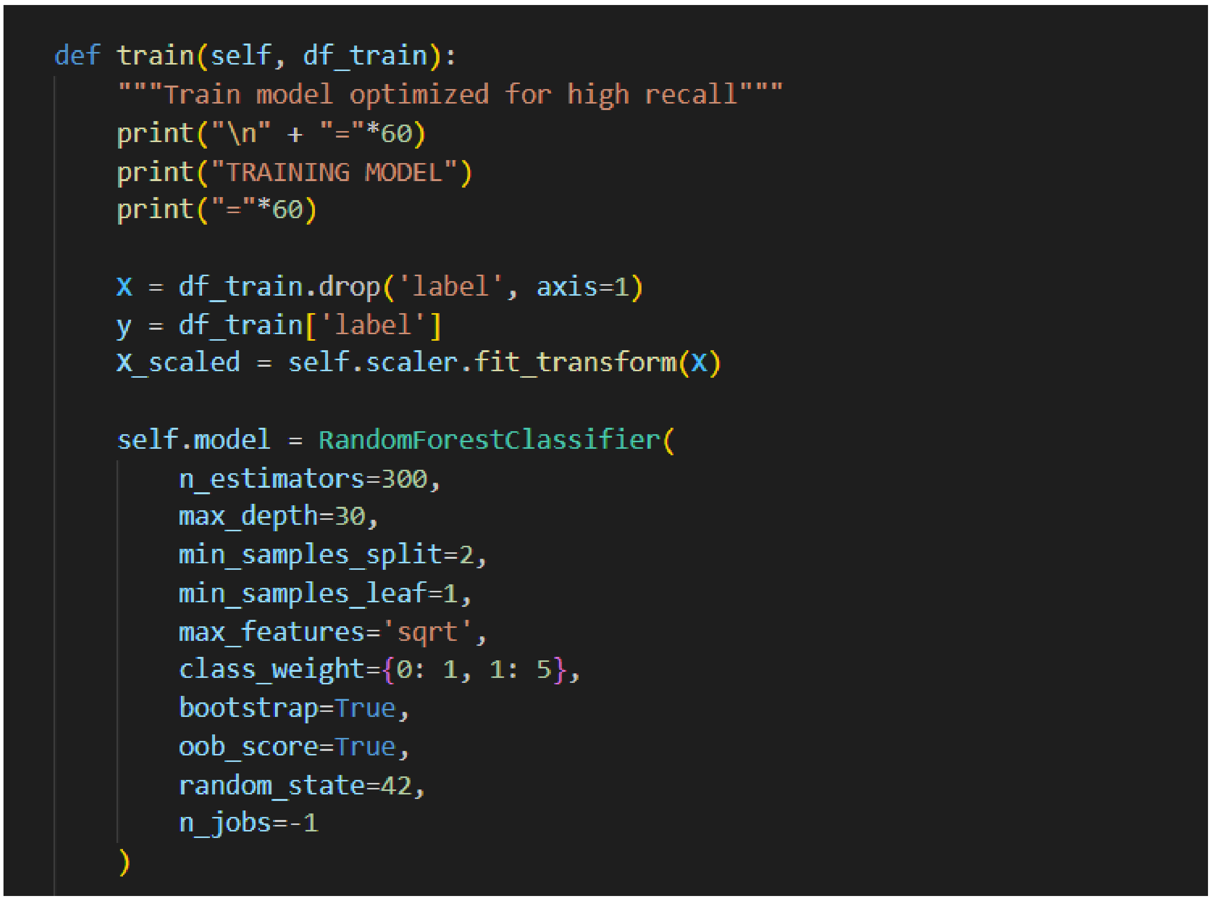


```python
def balance_data(self, patterns, labels):
    """Balance dataset using undersampling"""
    patterns = np.array(patterns, dtype=object)
    labels = np.array(labels)

    hs_idx = np.where(labels == 1)[0]
    nhs_idx = np.where(labels == 0)[0]

    np.random.seed(42)
    nhs_idx_sampled = np.random.choice(nhs_idx, size=len(hs_idx), replace=False)

    balanced_idx = np.concatenate([hs_idx, nhs_idx_sampled])
    np.random.shuffle(balanced_idx)

    return patterns[balanced_idx].tolist(), labels[balanced_idx].tolist()
```

This balances the heavily imbalanced dataset by undersampling non-hotspots to match hotspot count.

**Purpose:** Aggregates all benchmark datasets, extracts features from images, and creates structured train/test DataFrames ready for ML.

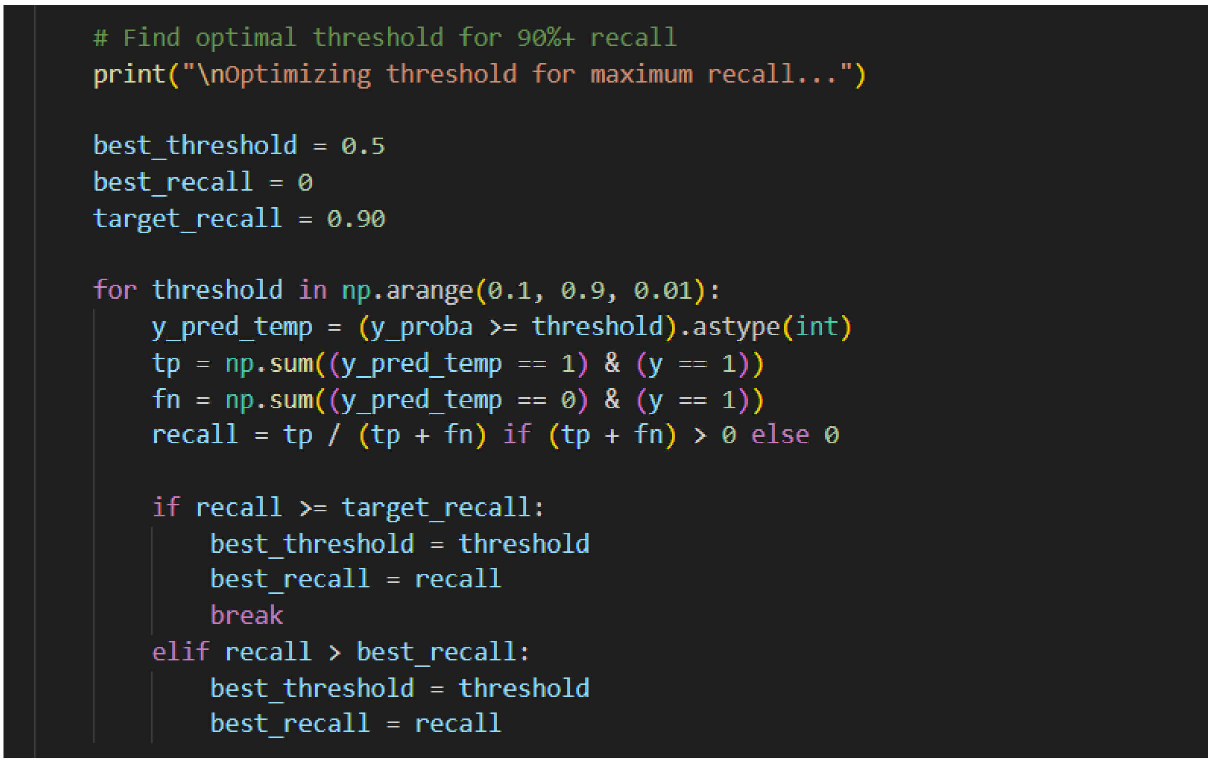
## 4. Model Training with High Recall Focus



```python
def train(self, df_train):
    """Train model optimized for high recall"""
    print("\n" + "="*60)
    print("TRAINING MODEL")
    print("="*60)

    X = df_train.drop('label', axis=1)
    y = df_train['label']
    X_scaled = self.scaler.fit_transform(X)

    self.model = RandomForestClassifier(
        n_estimators=300,
        max_depth=30,
        min_samples_split=2,
        min_samples_leaf=1,
        max_features='sqrt',
        class_weight={0: 1, 1: 5},
        bootstrap=True,
        oob_score=True,
        random_state=42,
        n_jobs=-1
    )
```

**Purpose:** Normalizes features and trains a Random Forest classifier with balanced class weights to handle the imbalanced hotspot/non-hotspot distribution.

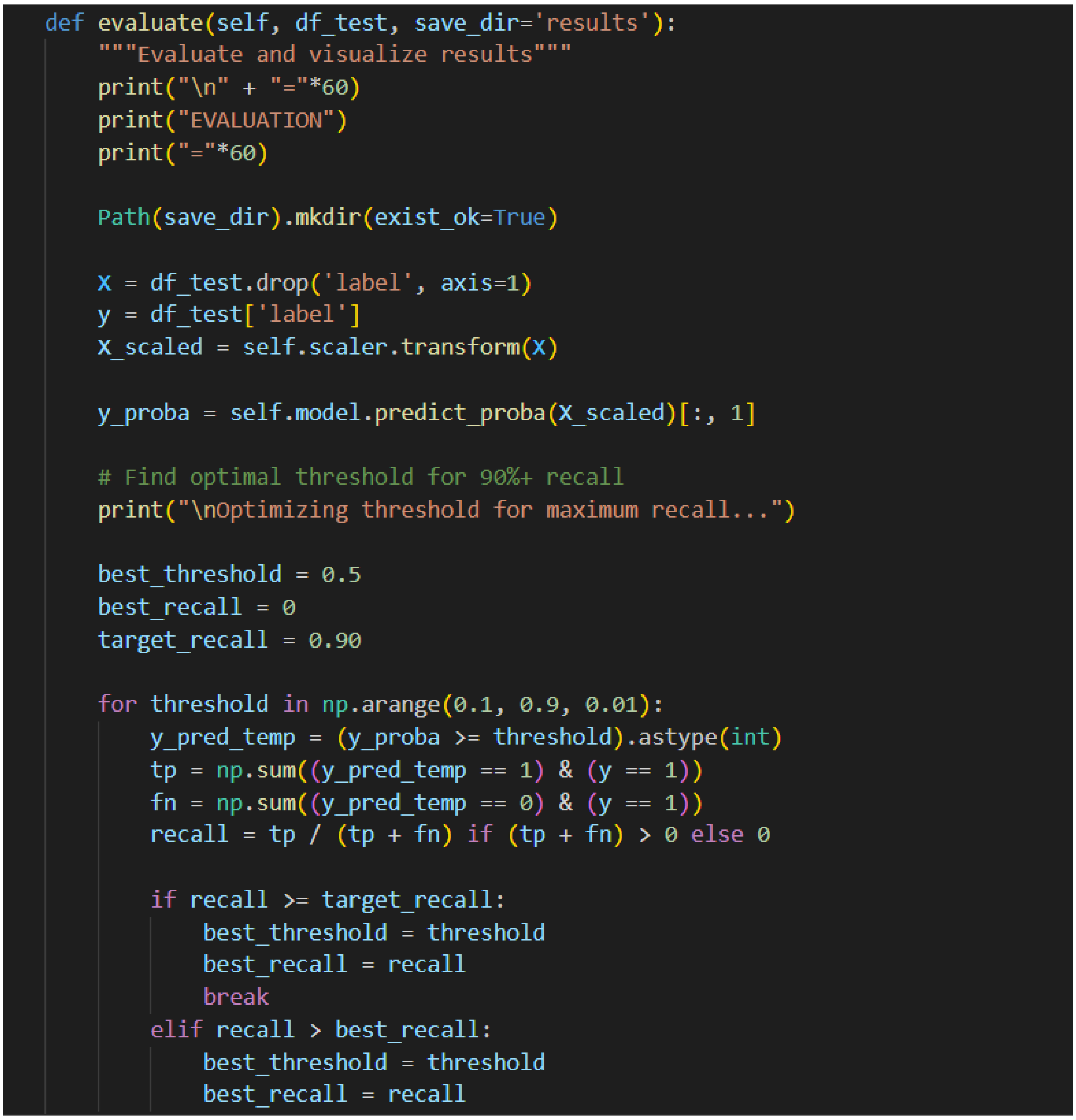
## 5. Threshold Optimization for Maximum Recall



```python
    # Find optimal threshold for 90%+ recall
    print("\nOptimizing threshold for maximum recall...")

    best_threshold = 0.5
    best_recall = 0
    target_recall = 0.90

    for threshold in np.arange(0.1, 0.9, 0.01):
        y_pred_temp = (y_proba >= threshold).astype(int)
        tp = np.sum((y_pred_temp == 1) & (y == 1))
        fn = np.sum((y_pred_temp == 0) & (y == 1))
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0

        if recall >= target_recall:
            best_threshold = threshold
            best_recall = recall
            break
        elif recall > best_recall:
            best_threshold = threshold
            best_recall = recall
```

This finds the optimal classification threshold to achieve at least 90% recall, prioritizing catching hotspots over false alarms.

## 6. Comprehensive Results Generation and Visualization



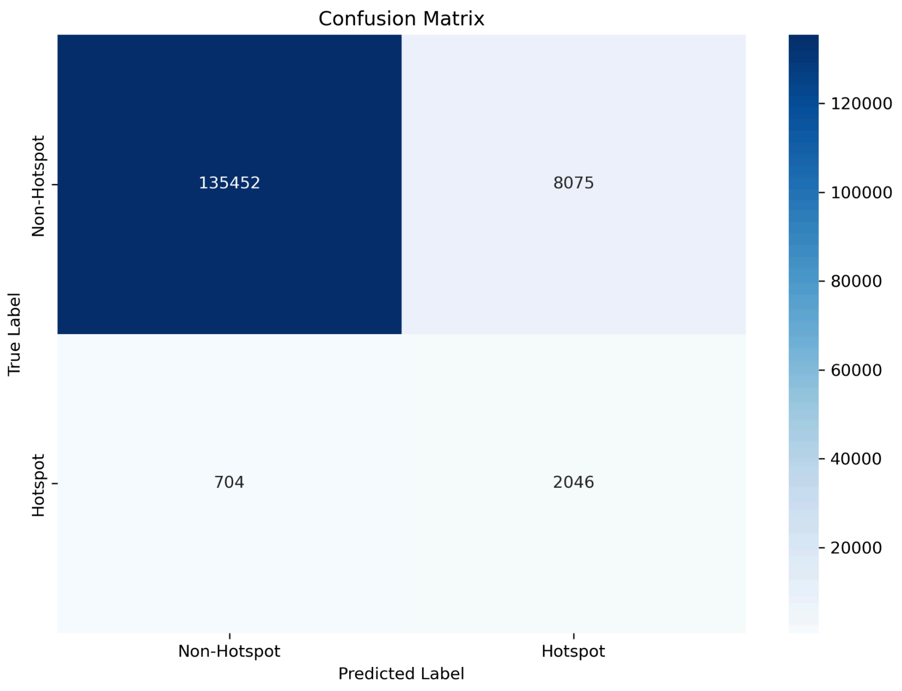
```python
def evaluate(self, df_test, save_dir='results'):
    """Evaluate and visualize results"""
    print("\n" + "="*60)
    print("EVALUATION")
    print("="*60)

    Path(save_dir).mkdir(exist_ok=True)

    X = df_test.drop('label', axis=1)
    y = df_test['label']
    X_scaled = self.scaler.transform(X)

    y_proba = self.model.predict_proba(X_scaled)[:, 1]

    # Find optimal threshold for 90%+ recall
    print("\nOptimizing threshold for maximum recall...")

    best_threshold = 0.5
    best_recall = 0
    target_recall = 0.90

    for threshold in np.arange(0.1, 0.9, 0.01):
        y_pred_temp = (y_proba >= threshold).astype(int)
        tp = np.sum((y_pred_temp == 1) & (y == 1))
        fn = np.sum((y_pred_temp == 0) & (y == 1))
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0

        if recall >= target_recall:
            best_threshold = threshold
            best_recall = recall
            break
        elif recall > best_recall:
            best_threshold = threshold
            best_recall = recall
```

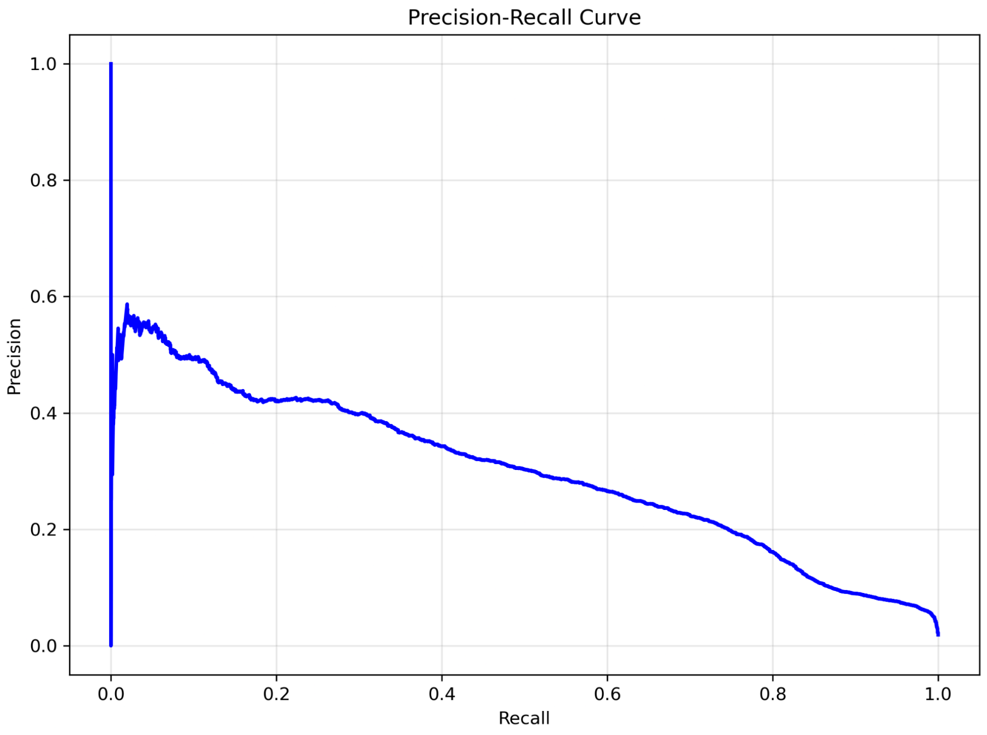
## Results

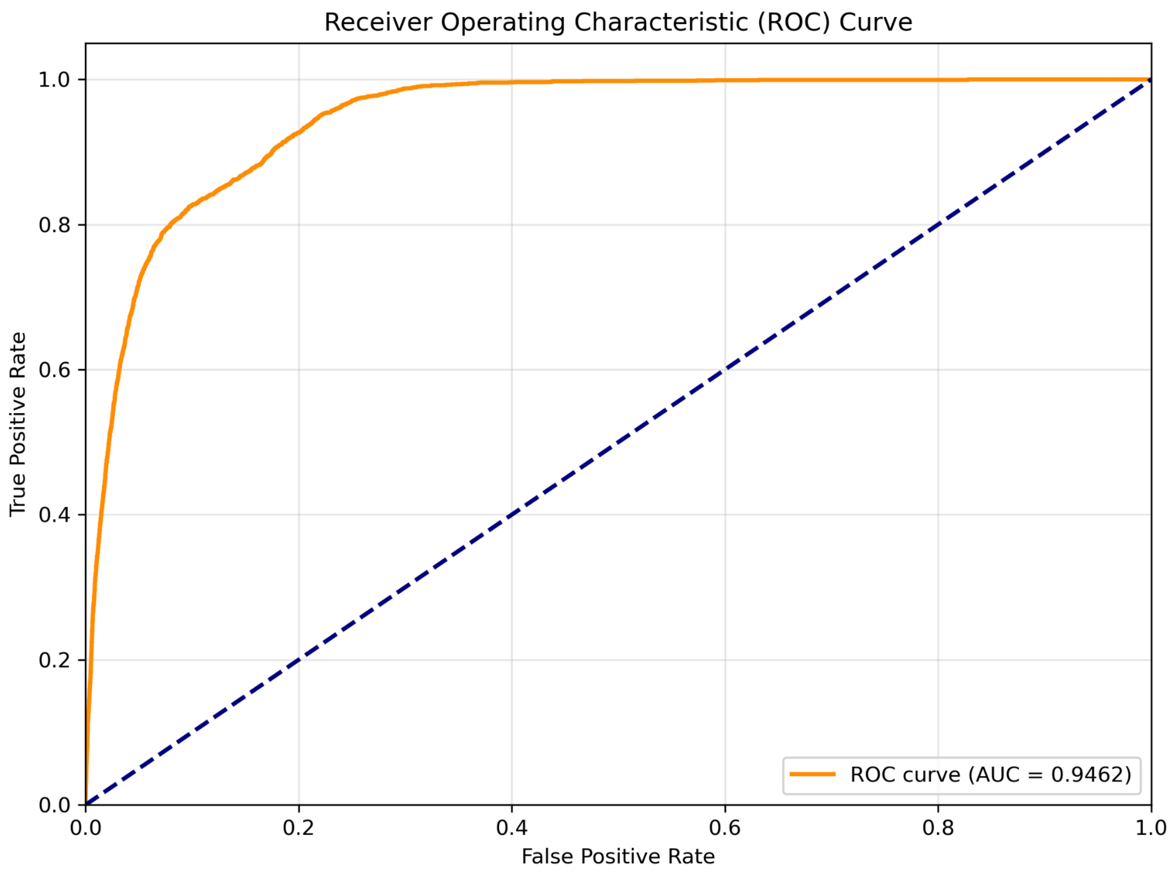
| Metric | Value |
|---|---|

| | |
|---|---|
| Accuracy | 94.08% |
| Precision | 20.22% |
| Recall | 74.40% |
| F1-Score | 31.79% |
| False Alarm Rate | 5.63% |

The final model, trained batch-wise and evaluated on the ICCAD-2012 test set, achieves high recall for hotspots but shows low precision and moderate F1-score, reflecting the significant challenge of class imbalance. Accuracy remains high, as most patterns are non-hotspots.


Confusion Matrix
135452
8075
704
2046
Non-Hotspot
Hotspot
True Label
Non-Hotspot
Hotspot
Predicted Label
120000
100000
80000
60000
40000
20000


Precision-Recall Curve
Precision
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Recall



## Conclusion and Future Scope

The scalable ML workflow demonstrates efficient processing and effective hotspot detection on large layout datasets, though future research should further address precision and class imbalance. Potential improvements include advanced re-sampling, feature selection, deep learning architectures, and ongoing model adaptation for newly emerging defect types and datasets.

## References

1. Intelectron6 GitHub - Lithography-Hotspot-Detection repository (2023).
2. ICCAD-2012 Benchmark Dataset.
3. Yang et al., IEEE TCAD, 2017.
4. Fuzzy pattern matching and imbalance-aware methods.