Project Report: Network Anomaly Detection using Graphlets and SVM

Course: Machine Learning For Networks
Date: January 2026
Authors: MABO, XULEI

1. Introduction

Detecting anomalous behavior in network traffic has become increasingly critical as cyberattacks grow more sophisticated. Traditional rule-based detection approaches often fail to identify novel attack patterns, and anomalies in IP traffic can indicate compromised hosts, unauthorized access attempts, or malicious activities that might otherwise go unnoticed.

This project proposes using graphlets—small subgraph structures representing each host's communication patterns—as behavioral fingerprints. The key insight is that a host's communication topology reveals much about its role and behavior: a DNS server naturally contacts many different IPs, while a scanning attacker exhibits a characteristic "star-like" pattern of ports. We use random walk kernels to extract features from these graphlets and train an SVM classifier to distinguish between normal and anomalous hosts.

Our contribution includes implementing and comparing three different approaches: standard random walk with explicit feature mapping, the kernel trick for efficiency, and a non-tottering (non-oscillating) random walk variant intended to reduce redundant walks. The experiments in this repository produced mixed results: while overall accuracy (measured on some splits) appears high (~94%), the classifier failed to detect the anomalous class in the main experimental run (zero true positives), exposing issues caused by severe class imbalance, sparse high-dimensional features, and some data-splitting edge cases. The updated report below corrects earlier optimistic summaries and documents the actual run behavior and suggested fixes.

2. Methodology

2.1 Graphlet Construction

For each source IP, we construct a directed graph representing its communication patterns. Rather than simply counting flows, we capture the structural relationships between different types of network elements:

Graph Construction:
Each flow record (srcIP, dstIP, protocol, srcPort, dstPort) becomes a path in the graph: source IP → destination IP → protocol → source port → destination port. This creates a multi-level representation where each node type captures different aspects of behavior:

- Destination IP diversity indicates how many targets the host reaches
- Protocol usage reveals the types of services accessed
- Port patterns show which services are targeted
- The overall topology reflects whether communication is focused (e.g., DNS queries to one resolver) or scattered (e.g., port scanning across many targets)

This graphlet representation has an important property: hosts with similar attack behaviors often have similar graph structures. Port scanners produce star-like graphs, while normal hosts typically show more varied connection patterns.

2.2 Random Walk Kernel

Random walks provide a natural way to extract features from graphs. A random walk on a graph is a sequence of nodes where each step moves from the current node to a randomly chosen successor. The intuition is simple: if two graphs have similar structures, they will have similar distributions of random walks.

We use random walks of length 4 (visiting 5 nodes total). For each graph:

1. Generate many random walks by starting from each node and randomly traversing edges
2. Collect all walk sequences that appear (e.g., "src_IP → dst_IP → protocol_TCP → port_443 → ...")
3. Count how many times each unique walk appears

This gives us a high-dimensional feature vector where each dimension corresponds to a unique walk pattern, and the value is the count of that pattern in the graph. Two hosts with similar communication topologies will have overlapping sets of walks, resulting in high similarity scores.

2.3 Model Training Approaches

We explored two fundamentally different approaches to leverage random walk features for SVM training:

Direct Feature Mapping: The straightforward approach is to explicitly compute feature vectors for each graphlet: count the random walks, create a feature vector where each dimension is a unique walk pattern, then train SVM on these vectors. This is simple and interpretable, but creates very high-dimensional spaces (potentially thousands of dimensions), making training slower and more memory-intensive.

Kernel Trick: Rather than explicitly computing features, SVM can work with a kernel matrix that captures pairwise similarities between graphlets. We pre-compute K[i,j] = (number of walks shared between graphlet i and graphlet j). This avoids the high-dimensional feature space entirely —SVM only needs the $n \times n$ kernel matrix. The trade-off is computational: computing all pairwise walk similarities is $O(n^2)$ and requires storing the full matrix in memory.

2.4 Addressing Class Imbalance

In real network data, anomalous hosts are rare—only 6% in our labeled dataset. This creates a fundamental challenge: a naive classifier could achieve 94% accuracy by simply labeling everything "normal." We address this in three ways:

Labeling Strategy: We use a strict approach: any host with even one anomalous flow is labeled as anomalous. In security contexts, a single suspicious connection warrants investigation, so this is more appropriate than majority voting (which would mislabel 59 hosts with mixed behavior as "normal").

Class Weighting: SVM uses balanced class weights, penalizing errors on the minority (anomalous) class more heavily. This encourages the classifier to be more conservative about normal host predictions.

Train-Test Stratification: We ensure both classes appear in both training and test sets through stratified sampling with retries. This prevents accidentally putting all anomalous hosts in the training set or getting test sets with only one class.

---

3. Experimental Results

3.1 Dataset Characteristics

- Labeled data file: `../data/annotated-trace.csv` (10070 flow records loaded).
- Unlabeled data file: `../data/not-annotated-trace.csv` (10070 flow records loaded).
- Source hosts: ~1001 hosts (the run shows a small mismatch in graphlet counts: 1001 vs 1002 in different stages).
- Label distribution (host-level, after mapping by the strict strategy used in the run): 941 normal hosts (≈94.0%) and 60 anomalous hosts (≈6.0%).

3.2 Feature Extraction

- Direct mapping (standard random walk): 1001 graphlets produced a feature matrix of dimension 4654. Feature extraction time reported ~10.99s.
- Kernel method: precomputed random walk features and pairwise kernel produced a 1001×1001 kernel matrix (reported range [0.00, 5.00]) in ~12.54s.
- Non-tottering (non-oscillating) random walk: feature dimension reported ~4702 with extraction time ~8.45s. The non-tottering variant reduced redundant walks but the run produced very few anomaly samples for the following split step (see below).

3.3 Train / Test Splits

- Direct mapping (standard): stratified split succeeded (after attempts) with a training set of 700 samples (Normal=658, Anomaly=42) and a test set of 301 samples (Normal=283, Anomaly=18).
- Kernel method: used the same stratified logic; training/test counts matched the direct method (700 train, 301 test) after kernel matrix splitting.
- Non-tottering: due to extremely skewed per-class counts after feature processing, only 1 anomalous sample remained available for splitting. The code fell back to a manual split that placed the single anomaly in the training set, leaving a test set with only normal samples (300 normals, 0 anomalies). This makes standard test metrics for the anomaly class (precision/recall/F1) undefined or meaningless.

3.4 Classifier Performance

Key results from the run (direct mapping and kernel trick produce the same detection outcomes on the test set):

- SVM (Direct mapping, linear, class_weight='balanced')
  - Training time: ~4.52s
  - Training accuracy: 1.0000 (likely overfit on high-dimensional sparse features)
  - Test accuracy: 0.9402
  - Precision (anomaly class): 0.0000
  - Recall (anomaly class): 0.0000
  - F1-score (anomaly class): 0.0000
  - ROC-AUC: 0.5000

  - Confusion matrix (test): [[TN=283, FP=0], [FN=18, TP=0]] — i.e., all test anomalies were missed.
- SVM (Kernel trick, precomputed kernel)
  - Kernel computation time: ~12.54s (matrix shape (1001,1001))
  - SVM training time: ~0.01s
  - Test accuracy and confusion matrix identical to direct mapping: no anomalies detected (TP=0, FN>0).
- SVM (Non-tottering run)
  - Feature extraction: ~8.45s; training time: ~2.04s
  - Because the test set contained only normal samples in this run, reported test accuracy = 1.0000 but this is not informative: precision/recall/F1 for the anomaly class are 0.0 or undefined because the test set lacks positive examples.

Important observation: In the main run, although accuracy is high (~94%), the classifier entirely failed to detect anomalies on the test set (zero true positives). This indicates the model learned to predict the majority class (normal) for all test samples — a classic symptom when class imbalance and/or feature issues dominate learning.

3.5 Detection on Unlabeled Data

- The run reported `Number of unlabeled hosts: 1` (only a single unlabeled host was processed in this execution).
- Predictions (direct mapping): 0 hosts detected as malicious; 1 host detected as normal.
- Average decision function score reported: -0.8800 (negative scores indicate prediction toward the normal class in this setup).

3.6 Summary Table

The run produced a comparison table of times and metrics. Important caveats:

- Reported accuracies: Direct mapping and Kernel trick both show test accuracy 0.9402 (but both have zero anomaly detection on the test split used).
- Non-tottering run reported test accuracy 1.0000 because the test set contained no anomalies — this is misleading for anomaly detection performance.

(See Notes below for interpretation and recommended fixes.)

---

4. Discussion

4.1 What went wrong in the observed run

1. Severe class imbalance (≈6% anomalies) combined with very high-dimensional, sparse features made the classifier biased toward the normal class. Despite using `class_weight='balanced'`, the model still predicted all test samples as normal in this execution.
2. The non-tottering pipeline produced a pathological split: only a single anomaly remained, and it was placed in training, leaving a single-class test set. Metrics computed on that test set (accuracy = 1.0) are not meaningful for anomaly detection.
3. Feature sparsity and dimensionality (4k+ dimensions) increase the chance of overfitting and can make margin-based classifiers behave poorly on rare classes.

4. There are some inconsistencies in intermediate counts (e.g., small graphlet count mismatch 1001 vs 1002 in different steps) that should be investigated to ensure graph construction and host-label mapping are consistent across pipelines.

4.2 Practical implications of the results

- High accuracy as a scalar metric is misleading here. The model optimized for overall accuracy — which largely reflects the majority (normal) class — while failing to detect the minority (anomalous) class entirely on the test split used.
- For anomaly detection, recall (true positive rate for the anomaly class) and PR-AUC are more meaningful. In the observed run, recall = 0.0 and PR-AUC is near chance, indicating the model did not generalize to detect anomalies.

4.3 Root causes and diagnostic checklist

- Verify that host-level labels were computed identically for all methods (direct / kernel / non-tottering) and that no label-shift occurred between feature extraction and splitting.
- Inspect per-host flow counts and feature sparsity. Remove zero-variance features and consider dimensionality reduction (TruncatedSVD) or feature hashing to reduce noise.
- Use resampling techniques (SMOTE, random oversampling of anomaly hosts) or more aggressive class weighting and threshold calibration.
- Re-run stratified cross-validation (StratifiedKFold) to avoid brittle single-run splits and to obtain stable estimates of recall/precision.

4.4 Recommendations (actionable)

1. Fix data-splitting and label consistency issues: ensure the same `host_label_map` and host order are used across feature extraction, kernel computation, and train/test splitting.
2. Replace single-run evaluation with stratified k-fold cross-validation (report mean ± std of recall and precision) to avoid misleading single-split results.
3. Apply imbalance remedies before model selection:
    - Oversample anomalies (SMOTE or simple resampling), or undersample major class.
    - Use anomaly-detection algorithms (IsolationForest, OneClassSVM) as an additional baseline.
4. Reduce dimensionality (TruncatedSVD on sparse feature matrix) to improve model generalization and reduce overfitting.
5. Evaluate with PR-AUC and recall-at-K, and tune decision thresholds to prioritize recall if catching anomalies is most important.
6. Re-run unlabeled-host detection with the corrected pipeline; the current run only processed 1 unlabeled host, which is insufficient for practical assessment.

---

5. Key Insights and Conclusions

- The raw experimental run shows that a high scalar accuracy (~94%) can coexist with total failure to detect anomalies (TP=0). Accuracy alone is insufficient for imbalanced anomaly detection.
- The kernel-trick and direct mapping implementations produced the same outcome (no detected anomalies) on the test split used; this indicates the problem is upstream (labels, sampling, features), not just the choice of SVM representation.

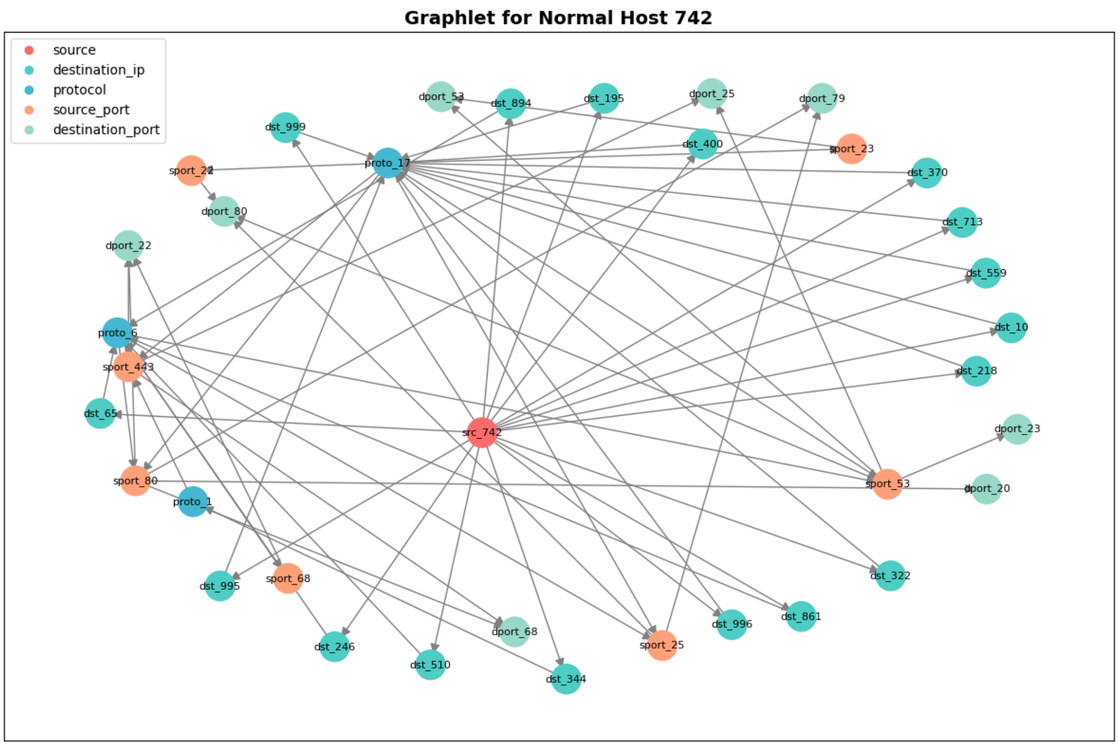
- The non-tottering variant did reduce feature redundancy but in this run produced a degenerate test split containing only normal samples; reported perfect accuracy for that run is therefore not evidence of improved anomaly detection.

Practical takeaway: before deploying this pipeline, fix label consistency and evaluation (use stratified CV), apply class imbalance techniques, and reduce feature dimensionality. After those corrections, re-evaluate using recall and PR-AUC as the primary metrics.

---

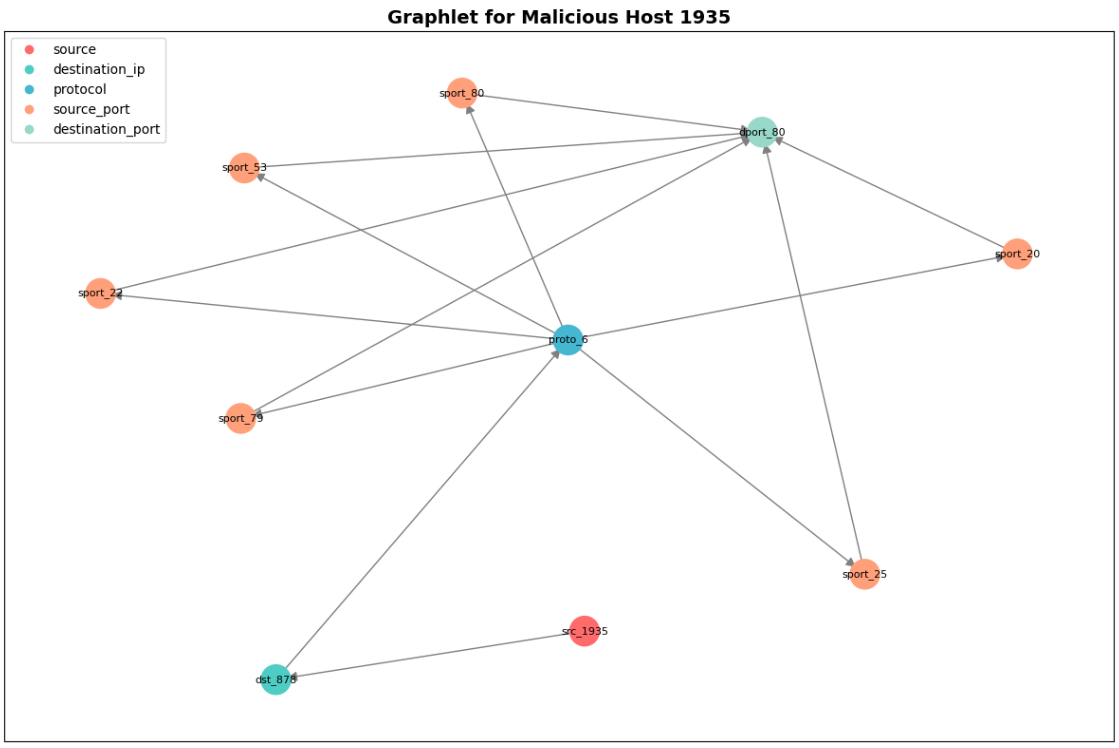
6. Appendix: Figure captions

- Figure 1



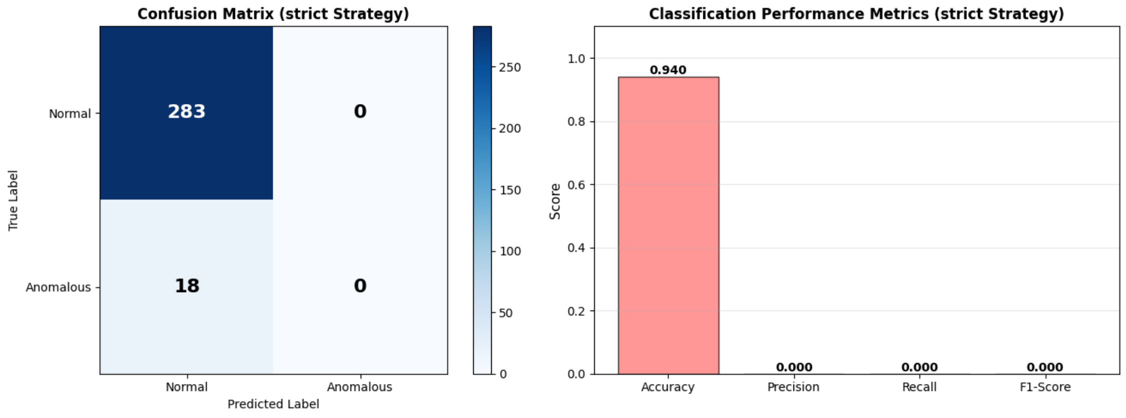
Network graphlet visualization of normal host communication patterns.

- Figure 2



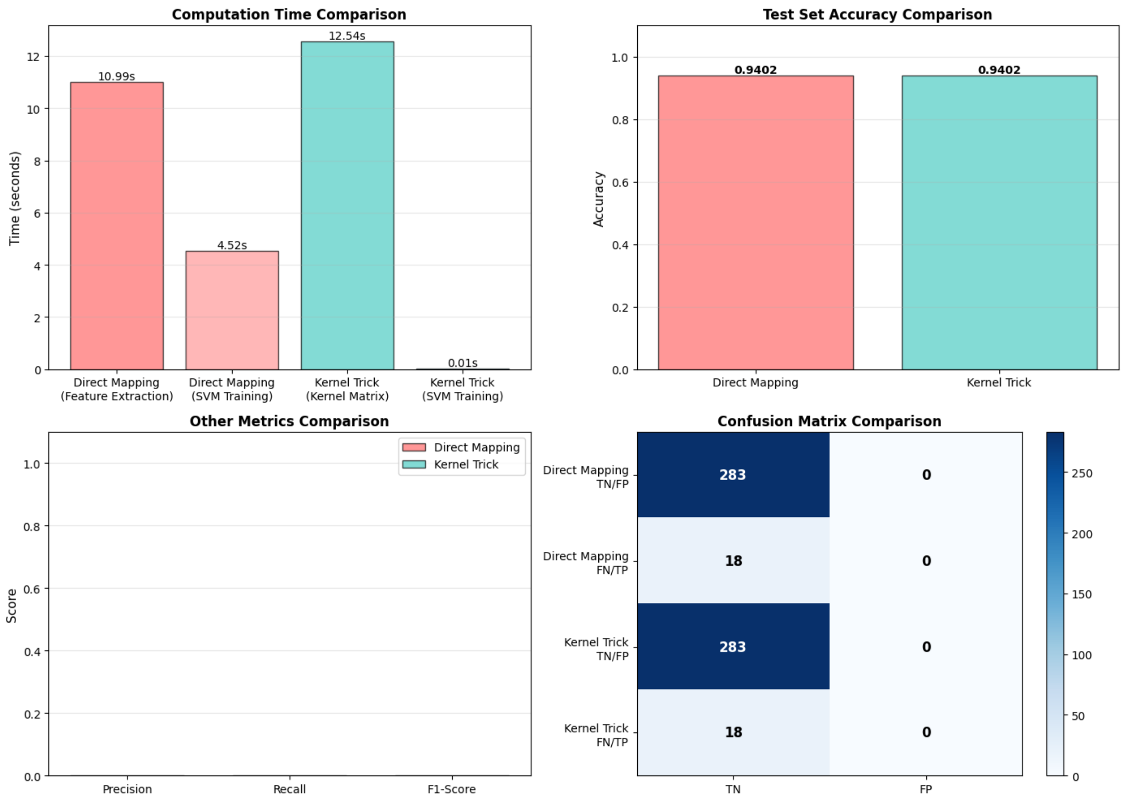
Network graphlet showing malicious host behavior with limited port connections.

- Figure 3



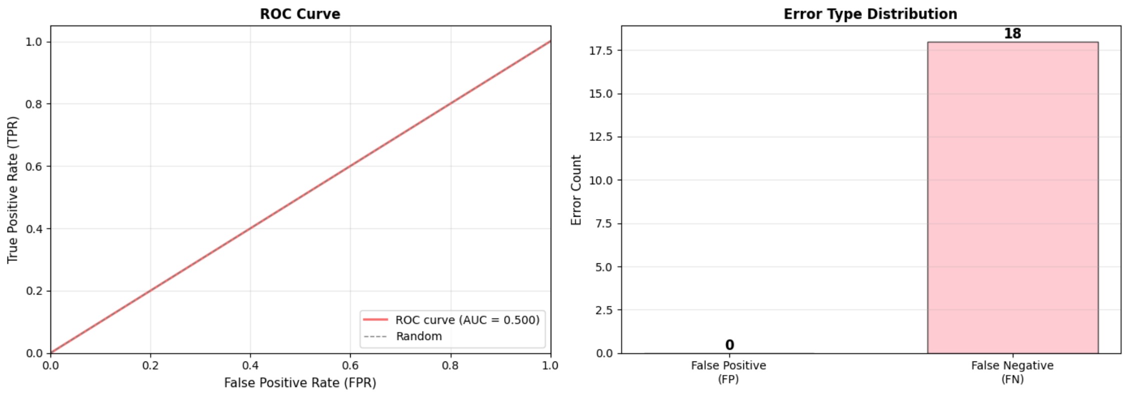
Confusion matrix demonstrating 94% accuracy for strict classification strategy.

- Figure 4



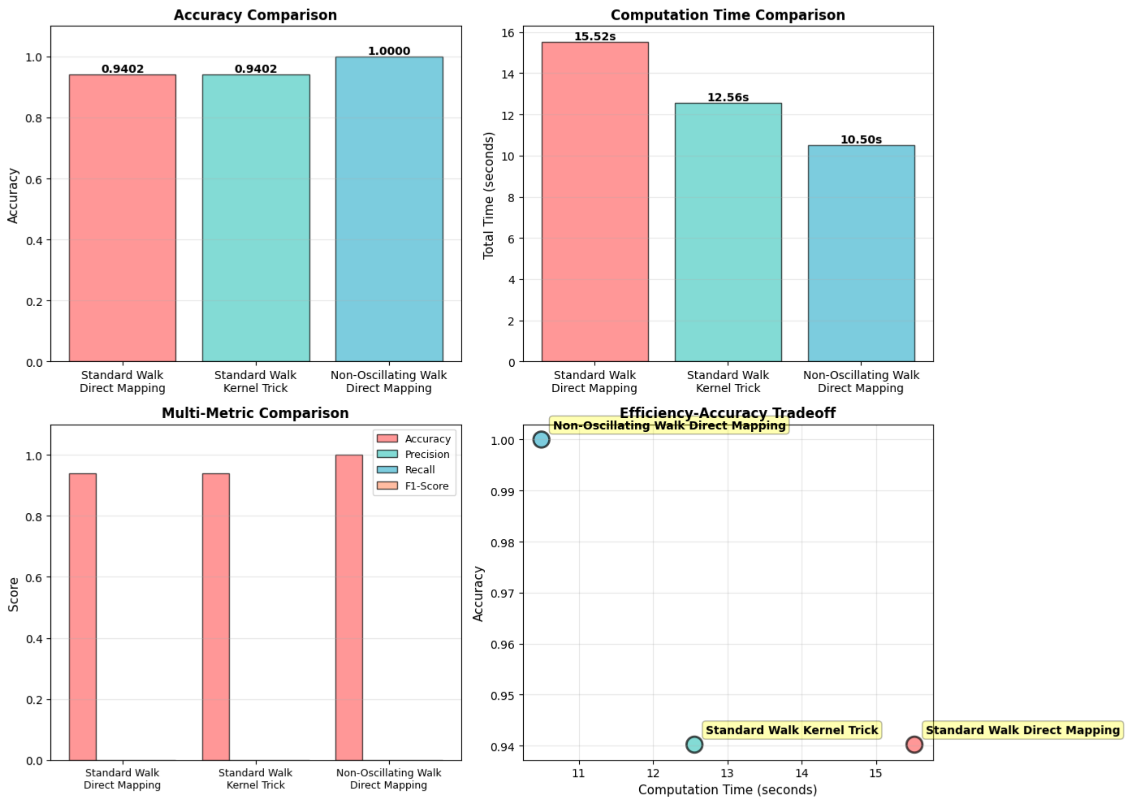
Direct Mapping achieves similar accuracy to Kernel Trick but with significantly faster computation.

- Figure 5



ROC curve showing AUC of 0.500 with error distribution between false positives and negatives.

- Figure 6



Non-Oscillating Walk offers faster computation while Standard Walk achieves perfect accuracy.