Detecting Network Failures with Machine Learning: A Comparative Analysis of Supervised, Unsupervised, and Semi-Supervised Techniques

Ben Towers and Taylor Hall

April 16th 2025

Abstract

This project explores the use of machine learning to detect anomalies in network traffic using the SOFI CoreSwitch-II dataset. The data represents a realistic enterprise network environment with a strong class imbalance between normal and failure events. We began by establishing a baseline using a Random Forest classifier, then explored unsupervised methods like Isolation Forest and Autoencoder to identify failures without labeled data. Supervised models, including a feedforward neural network and an SVM, were trained using class weighting and SMOTE to address imbalance. We also implemented a semi-supervised approach by combining anomaly scores from the autoencoder with a supervised classifier, which resulted in the best overall performance. Finally, we attempted an LSTM-based sequence model, but found limited value due to weak temporal structure in the dataset. Overall, the project gave us practical insight into the strengths and trade-offs of different modeling approaches for anomaly detection.

Introduction

Anomaly detection plays a key role in maintaining the reliability and security of networked systems. Failures in these systems can lead to downtime, data loss, or degraded performance, so identifying them early is important. In many cases, failures are rare and may appear in ways that are difficult to predict, making traditional rule-based monitoring unreliable.

Machine learning offers a more adaptive approach by learning from past behavior and recognizing unusual patterns in network traffic. However, the effectiveness of these models depends on a number of factors, including data quality, class imbalance, model choice, and how well the models are tuned.

In this project, we work with the SOFI CoreSwitch-II dataset, which simulates network behavior in a large enterprise environment. Our goal is to detect failure events using a range of machine learning

models. We explore supervised, unsupervised, and semi-supervised methods, compare their strengths and weaknesses, and evaluate their performance in the context of detecting rare but important anomalies.

Dataset: SOFI CoreSwitch-II

The dataset used in this project is the SOFI CoreSwitch-II dataset, which was generated using GNS3 and Cisco devices to simulate a large enterprise network. Data was collected through SNMP polling every three minutes, capturing various metrics from edge-layer switches under both normal and fault-induced conditions.

Each row in the dataset represents a snapshot of network activity and includes features such as ICMP response time, packet error counts, discarded packets, bits sent and received, interface speeds, and device uptime. These features are grouped across internal, external, and peripheral interfaces to provide a broad view of network behavior.

Each instance is labeled as either “NE” (normal event) or “F” (failure). The class distribution is highly imbalanced, with failures accounting for less than 2% of the total records.

The dataset contains 12,971 labeled entries and over 30 numerical features. These characteristics made it a good candidate for testing different machine learning approaches to anomaly detection, especially under conditions where failures are rare and difficult to model.

Data Preprocessing

Before training any models, we applied several preprocessing steps to clean and prepare the dataset. We started by removing rows that contained placeholder values such as -1, which typically indicated missing or invalid data. We also dropped columns with low variance, as they provided little useful information for classification.

To improve model performance, we engineered several new features. These included per-second traffic rates, error rate ratios, and log-transformed versions of skewed metrics. The goal was to create features that captured more meaningful patterns in the data while maintaining interpretability.

All numerical features were standardized using StandardScaler to ensure consistent scaling across the dataset. We then split the data into training, validation, and testing sets to evaluate models fairly.

Since failure events made up a very small portion of the data, we addressed class imbalance in multiple ways. We used class weighting in models that supported it, applied SMOTE to generate synthetic failure samples, and performed threshold tuning to adjust sensitivity during prediction. These steps were critical to improving recall on the minority class without sacrificing overall model stability.

Semi-Standardized Training Procedure

To keep our comparisons fair and reduce variability between models, we applied a semi-standardized training process to all neural network-based models. This allowed us to better evaluate each model's performance based on its structure rather than inconsistencies in training setup.

We used the Adam optimizer across all applicable models due to its adaptability and strong performance in a wide range of tasks. Each model was trained for up to 50 epochs, with early stopping enabled based on validation loss to prevent overfitting.

For binary classification tasks, we used binary cross-entropy loss, with optional pos_weight adjustments to emphasize the minority failure class. We also used a consistent batch size of 32 and applied the same feature scaling to all models using StandardScaler.

To monitor generalization, we used a fixed validation set for all models during training. Where possible, we also set random seeds to reduce variance between runs and improve reproducibility.

This approach helped maintain a controlled training environment while giving each model a fair chance to perform under similar conditions.

Baseline Model: Random Forest

To establish a starting point for our comparisons, we selected Random Forest as our baseline model. Random Forest is an ensemble method that builds multiple decision trees on random subsets of the data and aggregates their predictions. It is well-suited for structured datasets, handles a mix of feature types, and tends to perform well with minimal tuning.

Because Random Forest is relatively robust to overfitting and interpretable, it provided a reliable foundation for testing our data preprocessing steps and evaluating feature importance. It also allowed us to explore early strategies for handling class imbalance, including class weighting and custom decision thresholds.

This model served as a benchmark for the rest of our experiments, helping us assess how much value more complex or specialized methods were adding beyond a strong traditional approach.

Implementation

The Random Forest model was implemented using scikit-learn with class weighting enabled to account for imbalance. We trained the model on the preprocessed feature set and extracted probability scores to allow for threshold tuning. The following code snippet summarizes the core implementation:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

# Train Random Forest with class weighting to handle imbalance
rf = RandomForestClassifier(n_estimators=100, class_weight='balanced',
random_state=42)
rf.fit(X_resampled, y_resampled)

# Predict class probabilities for test set
y_proba = rf.predict_proba(X_test)[:, 1]  # Probability of class 'F' (failure)

# Apply custom threshold to improve recall
threshold = 0.4
y_pred = (y_proba > threshold).astype(int)

# Evaluate model performance
print(classification_report(y_test, y_pred, target_names=["NE", "F"]))
```

This implementation captures the key elements: using class_weight='balanced' to address label imbalance, extracting predicted probabilities for threshold tuning, and evaluating the output using precision, recall, and F1-score. Similar structures were followed for other supervised models to keep evaluation consistent.

Results

The Random Forest model achieved an overall accuracy of 95% on the test set. While this seems high, it is important to note that this metric is heavily influenced by the model's performance on the majority class (normal events, or NE), which makes up over 97% of the dataset.

	precision	recall	f1-score	support
NE	0.99	0.96	0.97	1088
F	0.30	0.68	0.41	28
accuracy			0.95	1116
macro avg	0.64	0.82	0.69	1116
weighted avg	0.97	0.95	0.96	1116

As expected, the model performed well on these normal cases, with a precision of 0.99 and recall of 0.96. However, these metrics are less meaningful in the context of failure detection, since correctly predicting

normal behavior in an imbalanced dataset is relatively easy and contributes disproportionately to the accuracy score.

What matters more in this case is the model's ability to identify failure events. The classifier achieved a recall of 0.68 on the minority class (F), which means it successfully detected 68% of actual failures. This is a promising result, especially considering the scarcity of failure examples in the dataset. The precision for failures was 0.30, indicating a moderate number of false positives, but this was an acceptable trade-off given our goal of maximizing recall.

The macro-averaged F1-score was 0.69, offering a more balanced view of model performance across both classes. These results highlight how class imbalance can skew traditional metrics and reinforce the importance of using precision, recall, and F1-score—particularly for the failure class—when evaluating models for anomaly detection.

Unsupervised Models

We explored two unsupervised approaches for anomaly detection: Isolation Forest and Autoencoder. These models were trained without using failure labels, making them well-suited for detecting novel or unexpected anomalies in situations where labeled data is limited or unreliable. Both models rely on learning the structure of normal behavior and identifying deviations, but they do so in different ways—Isolation Forest isolates outliers through random partitioning, while the Autoencoder reconstructs input data and flags samples with high reconstruction error. These models provided a useful foundation for evaluating failure detection without supervision and later informed the design of our semi-supervised approach.

Isolation Forest

Isolation Forest is an unsupervised anomaly detection method that identifies outliers by randomly partitioning data points and measuring how quickly they become isolated. Anomalies tend to be easier to isolate because they differ more from the general data distribution, resulting in shorter paths through the trees. This makes the algorithm effective for detecting rare or unusual behavior in high-dimensional data without needing labeled training examples.

Implementation

The model was trained using the full set of unlabeled features, without reference to class labels. We used the default settings with contamination='auto', allowing the model to estimate the expected proportion of anomalies. During inference, each sample received an anomaly score, and predictions were generated based on whether a point was classified as an outlier (−1) or not (+1). The predictions were then compared against the known failure labels to evaluate performance.

```
# Train Isolation Forest on the full (unlabeled) feature set
iso = IsolationForest(contamination='auto', random_state=42)
iso.fit(X_train)

# Predict anomaly scores on test set (lower scores = more anomalous)
anomaly_scores = iso.decision_function(X_test)

# Classify anomalies using threshold (default: anything < 0 is an outlier)
y_pred = iso.predict(X_test)

# Convert prediction output to binary labels: 1 = failure, 0 = normal
# IsolationForest returns -1 for anomaly, 1 for normal
y_pred = (y_pred == -1).astype(int)

# Evaluate using ground truth failure labels
print(classification_report(y_test, y_pred, target_names=["NE", "F"]))
```

Results

The Isolation Forest model achieved an overall accuracy of 91%, which, like the Random Forest, is largely influenced by the majority class (NE). The model correctly classified 93% of normal events but only reached a precision of 0.25 and recall of 0.54 on failure cases. This means it was able to detect over half of the actual failures, but flagged a significant number of false positives in the process.

The macro-average F1-score of 0.65 provides a more balanced view and reflects moderate success in identifying both classes. These results are consistent with what's expected from unsupervised models: without access to labeled failures during training, they tend to be more sensitive to anomalies, resulting in higher recall but lower precision. Despite its limitations, Isolation Forest proved useful for detecting unseen or novel failure patterns and offered a starting point for combining anomaly scores in our semi-supervised model.

	precision	recall	f1-score	support
NE	0.98	0.93	0.95	1088
F	0.25	0.54	0.34	28
accuracy			0.91	1116
macro avg	0.62	0.73	0.65	1116
weighted avg	0.96	0.91	0.93	1116

Autoencoder

Autoencoders are unsupervised neural networks designed to learn a compressed representation of input data and reconstruct it as accurately as possible. When trained only on normal data, the model becomes good at reconstructing expected patterns and performs poorly on anomalous inputs. This makes reconstruction error a useful metric for detecting failures, which are likely to differ from the normal distribution the model has learned.

Implementation

We trained a simple feedforward autoencoder using only samples labeled as normal. The model architecture consisted of two encoding layers and two decoding layers with ReLU activations. After training, we calculated the mean squared reconstruction error for each test sample and set a threshold using the 95th percentile of training error. Samples with error above this threshold were classified as failures.

```
# Autoencoder architecture
class Autoencoder(nn.Module):
  def __init__(self, input_dim):
    super().__init__()
    self.encoder = nn.Sequential(
      nn.Linear(input_dim, 16),
      nn.ReLU(),
      nn.Linear(16, 8)
    )
    self.decoder = nn.Sequential(
      nn.Linear(8, 16),
      nn.ReLU(),
      nn.Linear(16, input_dim)
    )
```

```
    def forward(self, x):
        return self.decoder(self.encoder(x))

# Calculate reconstruction error on test set
model.eval()
with torch.no_grad():
    reconstructed = model(X_test_tensor)
    mse = torch.mean((X_test_tensor - reconstructed) ** 2, dim=1).numpy()

# Set threshold (e.g., 95th percentile of training errors or tuned on validation)
threshold = np.percentile(mse, 95)
y_pred = (mse > threshold).astype(int)  # 1 = failure, 0 = normal

# Evaluate predictions against ground truth
print(classification_report(y_test, y_pred, target_names=["NE", "F"]))
```

Results

The autoencoder achieved an overall accuracy of 89%, again driven primarily by strong performance on the majority class. It maintained a recall of 0.90 and F1-score of 0.93 for normal events, but the more relevant result is the model's performance on failure detection.

The model correctly identified 61% of failures, indicating that it successfully flagged a significant portion of anomalous samples using only normal training data. Precision on failure cases was lower at 0.22, reflecting a relatively high number of false positives, which is a common trade-off in unsupervised settings.

The macro-average F1-score of 0.63 shows balanced improvement compared to Isolation Forest. These results suggest that the autoencoder was able to generalize well on known normal behavior and identify outliers with reasonable sensitivity, making it a useful component in hybrid or semi-supervised models later in the pipeline.

	precision	recall	f1-score	support
NE	0.97	0.90	0.93	1088
F	0.22	0.61	0.32	28
accuracy			0.89	1116
macro avg	0.59	0.76	0.63	1116
weighted avg	0.95	0.89	0.91	1116

Supervised Models

After evaluating unsupervised methods, we shifted focus to supervised learning, where models are trained directly on labeled data to classify normal and failure events. These models have the advantage of being more targeted, but their performance depends heavily on the quality and balance of the training data. We experimented with a simple feedforward neural network and a Support Vector Machine (SVM), using SMOTE and class weighting to address the strong class imbalance. Both models were trained under the same standardized settings to allow for fair comparison.

Feedforward Neural Network

To evaluate the potential of deep learning on this task, we implemented a basic feedforward neural network. These models learn non-linear decision boundaries by stacking layers of interconnected neurons with activation functions. While relatively simple compared to more advanced architectures, feedforward networks can perform well on structured data when properly tuned.

Implementation

The network consisted of two hidden layers with ReLU activations and dropout layers for regularization. We used a sigmoid activation in the final layer to output probabilities for binary classification. The model was trained using binary cross-entropy loss and the Adam optimizer under our standardized training setup. Predictions were made on the test set by applying a probability threshold of 0.4.

```
# Define feedforward neural network
class SimpleNN(nn.Module):
  def __init__(self, input_dim):
    super().__init__()
    self.network = nn.Sequential(
```

```
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.network(x)
# Predict class probabilities
model.eval()
with torch.no_grad():
    y_proba = model(X_test_tensor).squeeze().numpy()

# Apply threshold
threshold = 0.4
y_pred = (y_proba > threshold).astype(int)

# Evaluate predictions
print(classification_report(y_test, y_pred, target_names=["NE", "F"]))
```

Results

The model achieved an overall accuracy of 95%, again mostly driven by strong performance on the majority class. However, it also performed well on the minority class, with a recall of 0.64 and an F1-score of 0.48 for failure detection. This suggests the model was able to capture meaningful patterns in the data and correctly identified the majority of failure events.

The precision for failures was 0.38, which reflects a reasonable trade-off between false positives and recall. The macro-average F1-score of 0.72 was among the highest of all models tested, indicating strong balanced performance. These results show that with proper tuning and class balancing, even a simple neural network can be effective for detecting rare failures in network data.

	precision	recall	f1-score	support
NE	0.98	0.95	0.96	1088
F	0.38	0.64	0.48	28
accuracy			0.95	1116
macro avg	0.68	0.80	0.72	1116
weighted avg	0.96	0.95	0.95	1116

Support Vector Machine (SVM)

Support Vector Machines are powerful classifiers that aim to find the optimal boundary (or hyperplane) that separates data points from different classes. Using a kernel function, SVMs can project inputs into higher-dimensional spaces where non-linear boundaries become linearly separable. This makes them a good fit for complex datasets with overlapping patterns. However, they are sensitive to class imbalance and require careful tuning of hyperparameters like C and gamma.

Implementation

We used an SVM with an RBF (radial basis function) kernel and enabled class weighting to help balance the influence of the minority class. Input features were standardized before training. Since SVMs do not provide probabilities by default, we enabled the probability=True setting to allow for threshold tuning. A classification threshold of 0.5 was applied to the predicted probabilities for the failure class.

```
# Initialize SVM with RBF kernel and class weighting
svm = SVC(kernel='rbf', class_weight='balanced', probability=True,
random_state=42)
svm.fit(X_train_scaled, y_train)

# Predict class probabilities
y_proba = svm.predict_proba(X_test_scaled)[:, 1]

# Apply threshold
threshold = 0.5
y_pred = (y_proba > threshold).astype(int)

# Evaluate predictions
print(classification_report(y_test, y_pred, target_names=["NE", "F"]))
```

Results

The SVM achieved an overall accuracy of 95%, which again reflects the strong presence of normal events in the dataset. More importantly, the model reached a precision of 0.44 and a recall of 0.39 for the failure class. While slightly lower in recall compared to the neural network, the higher precision suggests that when the SVM predicts a failure, it is more likely to be correct.

The macro-average F1-score was 0.69, showing that the SVM handled both classes relatively well despite the imbalance. These results indicate that with class weighting and proper scaling, SVMs can be a reliable option for structured classification tasks, especially when interpretability and precision are priorities.

	precision	recall	f1-score	support
NE	0.98	0.96	0.97	1088
F	0.44	0.39	0.41	28
accuracy			0.95	1116
macro avg	0.71	0.67	0.69	1116
weighted avg	0.96	0.95	0.96	1116

Semi-Supervised Model

To improve failure detection without sacrificing overall performance, we developed a semi-supervised model that combines insights from unsupervised learning with the structure of supervised classification. Specifically, we used the reconstruction error from a pre-trained Autoencoder—trained only on normal samples—as an additional feature for a Random Forest classifier. The goal was to capture subtle deviations that indicate failure, even if they don't align cleanly with the original feature space.

Implementation

The Autoencoder was first used to compute the reconstruction error for all samples. This error was then appended as a new feature to the existing scaled dataset. We retrained a Random Forest classifier on the new augmented feature set using the same class weighting strategy as before. A classification threshold of 0.4 was applied to the predicted probabilities for failure detection.

```
# --- Step 1: Get reconstruction error from a pre-trained Autoencoder ---
model.eval()
```

```
with torch.no_grad():
    X_all_tensor = torch.tensor(X_all_scaled, dtype=torch.float32)
    recon = model(X_all_tensor)
    reconstruction_error = torch.mean((X_all_tensor - recon) ** 2, dim=1).numpy()

# --- Step 2: Combine reconstruction error with scaled feature data ---
# Append reconstruction error as a new feature
X_augmented = np.hstack((X_all_scaled, reconstruction_error.reshape(-1, 1)))

# --- Step 3: Train supervised classifier on augmented data ---
X_train_aug, X_test_aug = X_augmented[train_indices], X_augmented[test_indices]
y_train, y_test = y_all[train_indices], y_all[test_indices]

rf = RandomForestClassifier(n_estimators=100, class_weight='balanced', random_state=42)
rf.fit(X_train_aug, y_train)

# --- Step 4: Predict and evaluate ---
y_proba = rf.predict_proba(X_test_aug)[:, 1]
threshold = 0.4
y_pred = (y_proba > threshold).astype(int)

print(classification_report(y_test, y_pred, target_names=["NE", "F"]))
```

Results

This hybrid model achieved our best overall performance. It reached 96% accuracy, but more importantly, it improved failure detection with a recall of 0.71 and precision of 0.50 on the failure class. The resulting F1-score of 0.59 for failures reflects a much stronger balance between identifying true failures and limiting false positives compared to other models.

The macro-average F1-score was 0.78, and macro recall was 0.84, which are the highest among all approaches tested. These results show that combining an unsupervised anomaly signal with a traditional classifier provides complementary value, enabling better generalization across both known and unknown failure types. This approach proved especially effective for handling rare events without overfitting to noise.

	precision	recall	f1-score	support
NE	0.98	0.96	0.97	1088
F	0.50	0.71	0.59	28
accuracy			0.96	1116
macro avg	0.74	0.84	0.78	1116
weighted avg	0.96	0.96	0.96	1116

LSTM Attempt

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network designed to handle sequential data. They are well-suited for tasks where patterns over time may influence the outcome, making them a logical candidate for modeling temporal behavior in network traffic. In this case, the goal was to see if short sequences of network data could help signal upcoming failures more effectively than treating each record independently.

Implementation

We structured the data into overlapping sequences using a sliding window of five time steps. Each sequence was labeled based on the final time step to simulate predicting failure slightly ahead of time. The LSTM consisted of one layer with 64 hidden units and a fully connected output layer followed by a sigmoid activation to generate class probabilities. A threshold of 0.4 was applied to the predicted probabilities to generate binary predictions.

```
# --- Step 1: Define LSTM model ---
class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim=64, num_layers=1):
        super().__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        _, (hn, _) = self.lstm(x)
        out = self.fc(hn[-1])
        return self.sigmoid(out)

# --- Step 2: Prepare sequential data using sliding windows ---
def create_sequences(X, y, window_size):
    X_seq, y_seq = [], []
```

```
    for i in range(len(X) - window_size):
        X_seq.append(X[i:i+window_size])
        y_seq.append(y[i+window_size - 1])  # label of the last time step
    return np.array(X_seq), np.array(y_seq)

window_size = 5
X_seq, y_seq = create_sequences(X_all_scaled, y_all)

# Split into train/test
X_train_seq = torch.tensor(X_seq[train_indices], dtype=torch.float32)
y_train_seq = torch.tensor(y_seq[train_indices], dtype=torch.float32)

X_test_seq = torch.tensor(X_seq[test_indices], dtype=torch.float32)
y_test_seq = y_seq[test_indices]  # keep as numpy array for evaluation

# --- Step 3: Initialize and evaluate model ---
model = LSTMModel(input_dim=X_train_seq.shape[2])
model.eval()

with torch.no_grad():
    y_proba = model(X_test_seq).squeeze().numpy()

# --- Step 4: Threshold and evaluate ---
threshold = 0.4
y_pred = (y_proba > threshold).astype(int)

print(classification_report(y_test_seq, y_pred, target_names=["NE", "F"]))
```

Results

The LSTM achieved a 97% accuracy, but this number is misleading due to the strong class imbalance. The model performed well on normal samples, with precision and recall of 0.98, but it struggled significantly with failure detection. It only identified 4% of failure cases, with an F1-score of 0.05, indicating that it rarely recognized true failures.

The macro-average F1-score was 0.52, showing minimal contribution from the minority class. These results suggest that, despite its theoretical advantages, the LSTM was not well suited to this dataset. The lack of strong temporal patterns between time steps likely made it difficult for the model to learn meaningful trends. With more structured time-series data or longer sequence windows, this approach may have been more effective.

	precision	recall	f1-score	support
NE	0.98	0.98	0.98	1088
F	0.09	0.04	0.05	28
accuracy			0.97	1116
macro avg	0.53	0.51	0.52	1116
weighted avg	0.96	0.97	0.97	1116

Conclusions

This project explored a range of machine learning methods for detecting network failures using the SOFI CoreSwitch-II dataset. We tested unsupervised models like Isolation Forest and Autoencoder, supervised models including a neural network and SVM, and a hybrid semi-supervised approach that combined both. Each model came with its own strengths and limitations, especially in the context of highly imbalanced data.

The Random Forest baseline performed well and set a solid benchmark. Unsupervised methods were helpful in detecting unknown patterns, but struggled with precision. Supervised models performed better overall, especially after applying SMOTE, class weighting, and threshold tuning. The best results came from our semi-supervised model, which successfully combined anomaly detection with classification to improve failure recall without heavily sacrificing precision.

While the LSTM-based sequence model did not perform well, it highlighted the importance of choosing models that match the structure of the data. Overall, this project gave us hands-on experience with a range of approaches and showed how tuning, preprocessing, and model design all contribute to building effective systems for failure detection.

Key Takeaways

Throughout this project, we gained practical insight into building and evaluating models for rare event detection. Below are the key lessons and observations that shaped our understanding and will guide our approach in future machine learning work:

- Addressing class imbalance was essential. Without it, most models defaulted to predicting only the majority class.

- Simpler models performed surprisingly well. With the right tuning, models like Random Forest matched or outperformed more complex architectures.

- Hybrid models showed the most promise. Augmenting the feature set with anomaly scores led to stronger recall and more balanced results.

- Model choice must match the data. The LSTM underperformed due to limited temporal structure in the dataset, highlighting the importance of model-data alignment.

- Preprocessing and threshold tuning made a significant difference. Cleaning the data, engineering new features, and adjusting decision thresholds often had more impact than model architecture itself.

- Standardized training helped ensure fair comparisons. Using consistent optimizers, loss functions, and validation strategies made it easier to isolate what each model contributed.

These takeaways not only reflect what worked in this project but also provide a solid foundation for future work in applied machine learning—especially in settings where data is messy, imbalanced, or hard to label.

References

Shiksha. (n.d.). *Anomaly detection: Definition, types, and use cases*. Shiksha Online. https://www.shiksha.com/online-courses/articles/anomaly-detection/

Vargas-Arcila, A., Corrales, J. C., Sanchis, A., & Rendón, Á. (2022). *SOFI dataset: Symptom-fault relationship for IP-network*. *Computer Networks, 216*, 109233. https://doi.org/10.1016/j.comnet.2022.109233

Wang, S., Balarezo, J. F., Kandeepan, S., Al-Hourani, A., Chavez, K. G., & Rubinstein, B. (2021). Machine learning in network anomaly detection: A survey. *IEEE Access, 9*, 152379–152396. https://doi.org/10.1109/ACCESS.2021.3126834

Yakovyna, V. S. (n.d.). *Software failures prediction using RBF neural network*. https://old-pratsi.op.edu.ua/app/webroot/articles/1324635817.pdf

Contributions

This project was a collaborative effort between Ben Towers and Taylor Hall. We worked closely throughout the process and split the major components evenly to ensure shared learning and balanced responsibility. Below is a breakdown of our individual contributions:

Ben Towers

- Led the implementation of supervised learning models (Random Forest, SVM, Neural Network)
- Built and tuned the semi-supervised model using autoencoder reconstruction error
- Designed and maintained the training pipeline, including threshold tuning and SMOTE integration
- Wrote the majority of the report sections on supervised, semi-supervised, and training procedures

Taylor Hall

- Focused on the implementation and evaluation of unsupervised models (Isolation Forest, Autoencoder)
- Built the LSTM sequence model and handled sequence data preprocessing
- Contributed to data cleaning, feature engineering, and initial exploratory analysis
- Wrote the majority of the report sections on unsupervised learning, LSTM, and dataset structure

We collaborated equally on planning, testing model combinations, discussing results, and editing the final report.