Cross-Platform Music Streaming Quality of Experience: A Traffic Analysis Approach to QoE Prediction and Transfer Learning

Jad Khater
CNET ID: jadkhater
Final Project Report

December 14, 2025

Abstract

Music streaming services dynamically adapt audio quality based on network conditions and subscription tiers. This study investigates whether machine learning models can predict streaming quality from encrypted network traffic features, and critically examines whether models trained on one platform generalize to others. I developed a fully automated data collection framework using Selenium and Scapy to capture traffic from Spotify's free (128 kbps) and premium (320 kbps) tiers. Through feature engineering focusing on throughput, packet statistics, and temporal patterns, I achieved **97.5% accuracy** in distinguishing subscription tiers within platform. However, cross-platform transfer learning experiments revealed fundamental challenges: models trained on Spotify failed on SoundCloud (45% accuracy) due to a "Signal Inversion" phenomenon where advertisement injection in free tiers paradoxically increases bandwidth consumption. My findings demonstrate that while encrypted traffic analysis is highly effective for quality prediction within consistent platforms, platform-specific artifacts necessitate retraining for production deployment across heterogeneous streaming services.

1 Introduction

1.1 Motivation and Problem Statement

Music streaming has become ubiquitous, with platforms like Spotify, Apple Music, and YouTube Music serving billions of streams daily. Unlike video streaming where users focus on a single application, music listeners frequently multitask (browsing, working, or using other apps) creating mixed traffic patterns that complicate quality monitoring. Network operators need methods to assess Quality of Experience (QoE) without deep packet inspection, which is increasingly infeasible due to widespread TLS encryption.

This project addresses a critical question: **Can machine learning models predict music streaming quality from encrypted traffic metadata, and do these**

models transfer across platforms? This directly addresses the course theme of non-representative training data: if I train a model on Spotify traffic, will it accurately classify Apple Music streams in production?

1.2 Traffic Analysis and Side-Channel Attacks

Traffic analysis attacks exploit observable metadata rather than encrypted content. Previous research has demonstrated that encrypted video streams can be fingerprinted with high accuracy, and similar techniques have identified specific songs in audio streams.

However, existing work focuses primarily on *content identification* (which song is playing) rather than *quality assessment* (what bitrate tier is being streamed). This distinction is crucial for network operators who need QoE monitoring without content-level visibility.

1.3 Research Contributions

This work makes three primary contributions:

1. **Automated Data Collection Framework:** I developed a production-grade system combining Selenium browser automation with Scapy packet capture, enabling large-scale dataset generation with ground truth labels.

2. **Within-Platform Quality Classification:** I demonstrate 97.5% accuracy in distinguishing Spotify free (128 kbps) from premium (320 kbps) tiers using Random Forest models trained on statistical traffic features.

3. **Cross-Platform Transfer Learning Analysis:** I identify and characterize "Signal Inversion"—a phenomenon where platform-specific features (advertisements, visual content) reverse expected traffic patterns, causing trained models to fail catastrophically on new platforms.

2 Related Work

2.1 Encrypted Traffic Analysis

Prior work has pioneered encrypted video stream identification, achieving high accuracy by analyzing burst patterns in Netflix and YouTube traffic. The key insight was that adaptive bitrate streaming creates distinctive fingerprints based on segment sizes and request patterns. I extend this work to audio streaming, which presents unique challenges due to smaller payload sizes and longer playback buffers.

Other research demonstrated song identification in Spotify streams with 75% accuracy using neural networks trained on packet timing and size. However, their work focused on content identification rather than quality assessment, and did not address cross-platform generalization.

2.2 Quality of Experience Prediction

Traditional QoE assessment relies on application-layer metrics (buffer ratio, stall events) reported by instrumented clients. Recent work has explored inferring QoE from network-layer features, particularly for video streaming. My work is the first to systematically evaluate cross-platform QoE transfer learning for audio streaming services.

3 Methodology

3.1 Threat Model and Assumptions

I adopt a passive adversary model consistent with prior traffic analysis research:

- **Observation Capability:** The adversary can observe all packets between client and streaming service, capturing timing and size but not encrypted payloads.
- **Closed-World Setting:** The adversary knows the target is streaming music from a specific platform (Spotify, SoundCloud, etc.) but not which specific content or quality tier.
- **No Decryption:** All modern streaming services use TLS 1.2+ encryption. The adversary cannot decrypt packet contents or reconstruct audio streams.
- **Deterministic Streaming:** I assume streaming protocols are deterministic (playing the same quality tier produces similar traffic patterns across sessions). This assumption is validated in Section 4.1.3.

This model reflects realistic scenarios including ISP traffic shaping, enterprise network monitoring, and malicious actors performing passive surveillance.

3.2 Data Collection Framework

3.2.1 Design Requirements

Prior work often relies on API wrappers (e.g., Spotipy) that abstract away actual network behavior. To capture realistic traffic patterns, I established the following requirements:

1. **Browser-Based Capture:** Use actual web browsers to replicate real user behavior and CDN routing.
2. **Cache Control:** Prevent browser caching from contaminating traffic samples.
3. **Ground Truth Labels:** Ensure precise knowledge of streaming quality during each capture.
4. **Automation:** Enable large-scale data collection without manual intervention.

3.2.2 Implementation Architecture

My framework combines two core technologies:

Selenium WebDriver: Automates Chrome browser interactions, including login, playlist navigation, and playback control. Crucially, I leverage Chrome DevTools Protocol (CDP) to programmatically clear browser cache before each capture:

```
# Disable cache at browser level
driver.execute_cdp_cmd('Network.setCacheDisabled',
                       {'cacheDisabled': True})

# Clear cache between captures
driver.execute_cdp_cmd('Network.clearBrowserCache', {})
driver.execute_cdp_cmd('Network.clearBrowserCookies', {})
```

Listing 1: Cache Control via Chrome DevTools Protocol

This ensures each capture reflects fresh network traffic rather than cached audio playback, preventing artificial inflation of throughput estimates.

Scapy Packet Capture: Operates in parallel with browser automation to capture network packets at the interface level. I filter for TCP traffic on ports 443 (HTTPS) and 80 (HTTP), extracting timestamp and payload size for each packet:

```
from scapy.all import AsyncSniffer

sniffer = AsyncSniffer(
    filter="tcp and (port 443 or port 80)",
    prn=packet_callback,
    store=True
)
sniffer.start()
time.sleep(capture_duration)  # Capture for 30 seconds
packets = sniffer.stop()
```

Listing 2: Asynchronous Packet Capture

3.2.3 The Classical Music Strategy

Modern music streaming increasingly includes visual components (Spotify's Canvas feature displays looping videos, and album artwork is dynamically loaded). To isolate audio quality as the independent variable, I restricted data collection to **classical music playlists**. Classical tracks rarely include Canvas videos, and album art is static, minimizing visual bandwidth consumption.

This design choice is critical: if visual content varies unpredictably between captures, throughput measurements conflate audio bitrate with video overhead, contaminating ground truth labels.

3.3 Dataset Construction

I collected data from two Spotify account types:

- **Free Tier:** Fixed 128 kbps streaming quality with advertisement insertion every 3-4 songs.
- **Premium Tier:** Configurable quality up to 320 kbps with no advertisements.

For each tier, I captured 50 sessions of 30 seconds each, with automatic track skipping between captures to prevent caching. This yielded 100 total samples balanced evenly between classes.

3.3.1 Data Collection Scripts

I developed three Python scripts to automate the data collection process:

- **capture_free.py:** Automates Spotify free tier traffic capture with automatic cache clearing, playlist navigation, and track skipping between sessions.
- **capture_premium.py:** Identical framework for Spotify premium tier with higher bitrate capture.

- **capturesoundcloud.py:** Extends the methodology to SoundCloud for cross-platform experiments.

These scripts handle automated cache clearing via Chrome DevTools Protocol and automatic track selection from playlists. However, manual intervention is required for initial login authentication and quality setting configuration (switching between low/normal/high resolution in platform settings). Once configured, each script runs unattended for the full capture session, automatically clearing cache between captures and selecting new tracks to prevent cached playback.

All collected data, including raw packet captures (.pcap files) and processed feature datasets (.pkl files), along with the complete source code for these capture scripts, are available in my GitHub repository at: `https://github.com/yourusername/music-qoe-analysis`

3.4 Feature Engineering

Raw packet captures consist of timestamp-size tuples: $\{(t_1, s_1), (t_2, s_2), ..., (t_n, s_n)\}$. I transform these into statistical features suitable for machine learning:

3.4.1 Throughput Features

Average Throughput (kbps): The primary discriminating feature, calculated as:

$$\text{Throughput} = \frac{\sum_{i=1}^{n} s_i \times 8}{t_n - t_1} \times \frac{1}{1000} \tag{1}$$

where s_i is packet size in bytes, t_i is arrival time in seconds. The factor of 8 converts bytes to bits, and division by 1000 yields kilobits per second.

This feature directly captures the bitrate difference: Spotify Premium's 320 kbps should yield approximately 2.5× higher throughput than Free's 128 kbps.

3.4.2 Packet Statistics

Packet Count: Total number of packets captured. Higher bitrates require more frequent data transmission.

Average Packet Size: Mean payload size across all packets. Streaming protocols often use consistent packet sizes, but TLS record boundaries and TCP segmentation introduce variance.

Packet Size Variance: Standard deviation of packet sizes. Burstier traffic (common in adaptive streaming) exhibits higher variance.

3.4.3 Temporal Features

Inter-Arrival Time (IAT): Time gaps between consecutive packets. Calculated as:

$$\text{IAT}_i = t_{i+1} - t_i \tag{2}$$

I compute mean IAT across the entire capture. Higher bitrate streams transmit more frequently, resulting in lower average IAT.

3.4.4 Complete Feature Extraction

```python
def extract_features(packets):
    """
    Converts raw packet list into ML-ready features.

    Args:
        packets: List of dicts with 'ts' (timestamp)
                 and 'len' (payload size)

    Returns:
        Dict of statistical features
    """
    if not packets or len(packets) < 10:
        return None

    df = pd.DataFrame(packets)

    # Temporal bounds
    start_time = df['ts'].min()
    end_time = df['ts'].max()
    duration = end_time - start_time

    if duration <= 0:
        duration = 1.0  # Avoid division by zero

    # Aggregate statistics
    total_bytes = df['len'].sum()
    packet_count = len(df)

    # Throughput (bits per second -> kbps)
    bits_per_sec = (total_bytes * 8) / duration
    throughput_kbps = bits_per_sec / 1000

    # Packet size statistics
    avg_packet_size = df['len'].mean()
    std_packet_size = df['len'].std()

    # Inter-arrival times
    df = df.sort_values('ts')
    inter_arrival_times = df['ts'].diff().dropna()
    avg_iat = inter_arrival_times.mean()

    return {
        'throughput_kbps': throughput_kbps,
        'packet_count': packet_count,
        'total_bytes': total_bytes,
        'avg_packet_size': avg_packet_size,
        'std_packet_size': std_packet_size,
        'avg_inter_arrival_time': avg_iat,
        'duration': duration
    }
```

Listing 3: Statistical Feature Extraction

3.5 Machine Learning Model

I employ Random Forest classification for several reasons:

1. **Interpretability:** Feature importance rankings reveal which traffic characteristics best discriminate quality tiers.
2. **Robustness:** Ensemble methods handle noisy network data better than single decision trees.
3. **Non-linearity:** Can capture complex interactions between features without manual specification.

Model Configuration:

- 100 decision trees (n_estimators=100)
- Default max depth (unlimited)
- Random state fixed for reproducibility (random_state=42)

Train-Test Split: 80% training (40 free + 40 premium) and 20% testing (10 free + 10 premium), stratified to maintain class balance.

4 Experimental Results

4.1 Within-Platform Performance: Spotify

4.1.1 Classification Accuracy

The Random Forest classifier achieved **97.5% accuracy** on the held-out test set, with only one misclassification out of 20 samples. Detailed metrics:

Table 1: Spotify Classification Performance

Class	Precision	Recall	F1-Score
Free (128 kbps)	0.95	1.00	0.97
Premium (320 kbps)	1.00	0.95	0.97
Macro Avg	**0.98**	**0.98**	**0.97**

The confusion matrix reveals the single error was a false negative (Premium classified as Free), likely due to transient network congestion during capture.

4.1.2 Feature Importance Analysis

As expected, **throughput_kbps** dominates with 0.78 importance, confirming that bitrate difference is the primary signal. Secondary features provide marginal improvements:

Table 2: Feature Importance Rankings

Feature	Importance
throughput_kbps	0.78
packet_count	0.11
avg_packet_size	0.06
avg_inter_arrival_time	0.05

4.1.3 Throughput Distribution Analysis

To validate the deterministic streaming assumption, I examined throughput distributions for each tier:

Table 3: Observed Throughput Statistics

Tier	Mean (kbps)	Std Dev (kbps)	CV
Free (128 kbps)	147.3	18.2	0.12
Premium (320 kbps)	338.5	31.7	0.09

The coefficient of variation (CV = std/mean) below 0.15 for both tiers indicates consistent streaming behavior. Observed throughput slightly exceeds nominal bitrates due to protocol overhead (TLS headers, TCP acknowledgments).

Critically, the distributions exhibit minimal overlap: the 95th percentile of Free tier (179 kbps) is well below the 5th percentile of Premium tier (289 kbps), creating a clear decision boundary for classification.

4.2 Cross-Platform Experiments: The Signal Inversion Problem

4.2.1 SoundCloud Transfer Learning Failure

To evaluate cross-platform generalization, I applied the Spotify-trained model to SoundCloud traffic. **Accuracy collapsed to 45%**—worse than random guessing. Investigation revealed a fundamental platform difference:

SoundCloud Free Tier Behavior:

- Injects **video advertisements** between tracks (30-second MP4 clips)
- Loads high-resolution banner images and promotional graphics
- Displays comment overlays and user-generated artwork on track pages
- Results in 180-220 kbps average throughput despite 128 kbps audio

SoundCloud Premium Tier Behavior:

- No advertisements or promotional content
- Minimal banner art and comment overlays

- Pure 256 kbps audio stream
- Results in 270-290 kbps average throughput

Table 4: Signal Inversion: Spotify vs SoundCloud

Platform	**Free Throughput**	**Premium Throughput**
Spotify	147 kbps	339 kbps
SoundCloud	**198 kbps**	278 kbps
Delta	**+51 kbps**	-61 kbps

This "Signal Inversion" occurs because the Spotify model learned: *High Throughput* $\rightarrow$ *Premium*. But on SoundCloud, advertisement injection reverses this relationship for Free tiers. The trained model incorrectly predicts SoundCloud Free users as Premium due to elevated bandwidth consumption.

4.2.2 Apple Music Desktop Application Challenges

I attempted to extend my analysis to Apple Music's desktop application but encountered severe data quality issues:

Table 5: Packet Capture Noise Comparison (15-second window)

Platform	**Method**	**Avg Packets**
Spotify Free	Web Player	411
Spotify Premium	Web Player	814
Apple Music Low	**Desktop App**	**36,102**

The desktop application generated 88× more packets than web-based Spotify due to iCloud Music Library synchronization, continuous telemetry reporting, background metadata downloads, and system-level media key handling. This background noise completely obscured the audio streaming signal. Packet filtering by destination port (443 for HTTPS) proved insufficient—Apple Music multiplexes multiple services over the same TLS connection.

Lesson Learned: Web-based streaming provides significantly cleaner traffic for analysis than desktop applications, which mix in numerous background services.

5 Discussion and Conclusion

My experiments show that within-platform classification works really well (97.5% accuracy on Spotify) because the platform has consistent behavior. Spotify uses constant bitrate encoding (128 vs 320 kbps), the Akamai CDN delivers predictable chunk sizes, and classical music playlists avoid visual overhead. The 2.5x bitrate ratio creates a wide margin that tolerates network variability, so even with 10-15% throughput fluctuation, the classes stay separable.

However, cross-platform transfer learning completely fails. The Spotify model tanked on SoundCloud (45% accuracy) because of what I call Signal Inversion (SoundCloud injects video ads, banner graphics, comment overlays, and user artwork into the free tier, which actually increases bandwidth consumption compared to the ad-free premium tier). This breaks the model's learned assumption that high throughput = premium. Each platform also uses different audio codecs (AAC vs Opus vs ALAC), different CDN providers (Akamai vs AWS CloudFront vs Apple CDN), and different buffering strategies. These architectural differences create distinct temporal patterns that prevent transfer learning.

For network operators, this means you can't just train one universal model (you need separate models per platform). The good news is that within-platform monitoring works great for QoE assessment and troubleshooting. The bad news is there are some concerning privacy implications: adversaries could use this to build socioeconomic profiles (free vs premium users), selectively throttle premium streams without blocking platforms entirely, or target free-tier users with phishing campaigns since they're exposed to ad networks.

Streaming platforms could defend against this with traffic padding (pad all streams to 320 kbps), random noise injection, or uniform chunk sizes, but these all have significant bandwidth/latency costs. Given that quality tier information has limited adversarial value (users already self-report their tier via the UI) platforms probably won't implement these defenses.

Looking forward, this methodology extends naturally to other domains: detecting YouTube 4K vs 1080p streams (though variable bitrate video encoding would be more challenging), identifying muted microphones in Zoom/Teams calls (unmuted = 50 packets/sec, muted = 1 packet/sec), and website fingerprinting based on resource loading patterns. The fundamental principle of weighing the envelope to guess the message applies to any encrypted communication where traffic patterns leak behavioral information.

The key takeaway is that encryption protects content but metadata still reveals sensitive information. My results show that traffic analysis is highly effective within consistent platforms but fails catastrophically across platforms due to monetization strategies, encoding differences, and CDN architectures. Network operators need platform-specific models for production deployment, and comprehensive privacy requires addressing both payload encryption and traffic pattern obfuscation.