DJ Song Mixing Recommendation System

Intelligently Recommending Songs for Seamless DJ Transitions

CS Machine Learning - UVA Fall 2025
Ashley Wu, Bonny Koo, Nathan Suh, Leo Lee

Motivation & Related Work

The Mixing Challenge

- DJs spend hours finding songs that mix well together
- Requires deep knowledge of BPM, key compatibility, energy flow
- Even experienced DJs struggle with large music libraries

Why This Matters

- For DJs: Save time finding compatible songs, discover new mixing possibilities
- For aspiring DJs: Learn which songs work together and why
- For music platforms: Create better automated DJ mixes and playlists
- For us: Combine music theory with machine learning

Existing DJ Tools & Research

- Mixed In Key: Analyzes key/BPM but no ML recommendations
- Spotify DJ Feature: Uses popularity, not true mixing compatibility
- DJ software (Serato, Rekordbox): Show compatible keys but don't recommend
- Academic: Limited research on automated DJ song selection

Gap in Existing Solutions

No system combines BPM, key, AND energy for intelligent recommendations

Our Contribution: ML system using DJ mixing rules + audio features

Background: What Makes a Good DJ Mix?

Three Key Factors for Seamless Transitions

1. BPM Matching

Tempo compatibility
±6 BPM ideal

Example:
120 BPM → 124 BPM

2. Key Compatibility

Harmonic mixing
Camelot Wheel

Example:
8A → 9A, 8B, 7A

3. Energy Flow

Smooth transitions
Build/drop patterns

Example:
Medium → High

Claim / Target Task

A recommendation system incorporating DJ mixing rules (BPM, key, energy) outperforms generic audio similarity for DJ transitions

Task Definition

Input: Currently playing song
(e.g., "Strobe" - deadmau5, 128 BPM, 8A)

Output: 10 mixable songs
ranked by compatibility

Constraint: 45 seconds

Success Criteria

- BPM Compatibility: ±6 BPM
- Key Compatibility: 80%+ match
- Energy Flow: Smooth transitions
- DJ Validation: Expert approval

High level System Architecture

CSV Dataset – Data preprocessing

Convert numeric keys to Camelot Notation (24 mappings) and calculate 4 compatible keys per song

CSV Dataset – Data preprocessing

Normalize audio features (0-1 scale)

Three parallel models

Rule-based

Audio Similarity

Hybrid

Output: 10 song recommendations per model sorted by compatibility score

Dataset Preprocessing – 114,000 Songs

Camelot Wheel

- The Spotify Dataset provides us numeric keys (0-11), but we use Camelot notation (1A-12B) to easily find harmonically compatible songs
- We then precompute the compatible keys for faster filtering later on, each song has a list of 4 compatible keys

```
def convert_to_camelot(dataset):
    dataset['key'] = pd.to_numeric(dataset['key'], errors='coerce').fillna(0).astype(int)
    dataset['mode'] = pd.to_numeric(dataset['mode'], errors='coerce').fillna(0).astype(int)

    dataset['camelot_key'] = dataset.apply(
        lambda row: key_to_camelot(row['key'], row['mode']),
        axis=1
    )
```

```
def add_compatible_keys(dataset):
    dataset['compatible_keys'] = dataset['camelot_key'].apply(
        lambda x: get_compatible_keys(x) if pd.notna(x) else []
    )
```

Camelot Conversion

Camelot Wheel

- These are small snippets of how we convert the numeric key to the Camelot Notation, and a snippet of our function to get compatible keys as well.
- We use 2 modes to show whether they are A or B (representing minor and major respectively in the wheel)

```python
def key_to_camelot(key, mode):
    camelot_mapping = {
        # Major keys (mode=1) -> B ring
        (0, 1): (8, 'B'),   # C Major -> 8B
        (1, 1): (3, 'B'),   # C# Major -> 3B
        (2, 1): (10, 'B'),  # D Major -> 10B
        # ... (24 total mappings)

        # Minor keys (mode=0) -> A ring
        (0, 0): (5, 'A'),   # C Minor -> 5A
        (9, 0): (8, 'A'),   # A Minor -> 8A
        # ... (12 minor mappings)
    }
    key = int(key) % 12
    mode = int(mode) % 2
    if (key, mode) in camelot_mapping:
        camelot_number, camelot_letter = camelot_mapping[(key, mode)]
        return f"{camelot_number}{camelot_letter}"
```

```python
def get_compatible_keys(camelot_key):
    number = int(camelot_key[:-1])
    letter = camelot_key[-1]

    compatible = [camelot_key]  # Same key

    # ±1 on the wheel (circular)
    prev_number = number - 1 if number > 1 else 12
    next_number = number + 1 if number < 12 else 1
    compatible.append(f"{prev_number}{letter}")
    compatible.append(f"{next_number}{letter}")
```

Dataset Preprocessing – 114,000 Songs

Normalization

- We then normalize the loudness feature of all songs in the dataset to accurately compute audio similarity
- Originally from a range of –60 – 0 db, but we push it onto a scale of 0-1

```
def normalize_audio_features(dataset):
    # Normalize loudness (typically -60 to 0 dB)
    min_loudness = dataset['loudness'].min()
    max_loudness = dataset['loudness'].max()
    dataset['loudness_normalized'] = (dataset['loudness'] - min_loudness) / (max_loudness - min_loudness)
```

Camelot conversion and Normalization are two steps that lie within our data preprocessing pipeline, where we load and complete the pipeline.

```
def load_dataset(filepath=None):
    if filepath is None:
        from pathlib import Path
        base_path = Path(__file__).parent.parent
        filepath = str(base_path / 'data' / 'dataset.csv')
    print(f"Loading dataset from {filepath}...")
    df = pd.read_csv(filepath)
    print(f"Loaded {len(df)} tracks")
    return df
```

```
def preprocess_dataset(filepath=None):
    df = load_dataset(filepath)
    df = convert_to_camelot(df)
    df = add_compatible_keys(df)
    df = normalize_audio_features(df)
    return df
```

Proposed Solution: Three Models

Model 1: Rule-Based DJ System
Hard constraints: BPM ±6, Key compatibility (Camelot Wheel)
Filter songs → Rank by distance metric
Pro: Follows DJ theory | Con: Rigid, may miss good combinations

Model 2: Audio Feature Similarity (Baseline)
Content-based: Energy, valence, acousticness similarity
Ignores BPM/key constraints
Pro: Discovers unexpected matches | Con: May recommend unmixable songs

Model 3: Hybrid ML System (Our Approach)
ML model trained on: BPM distance, key compatibility, energy flow
Weighted scoring: 40% BPM, 30% Key, 30% Energy/Features
Pro: Balances rules + discovery | Con: Needs labeled training data

Implementation Overview

Tools & Technologies

- Kaggle Dataset: Tempo, key, mode
- XGBoost: ML models
- NumPy: BPM/key calculations

Steps to Implement

- Data + extract BPM/key
- Rule-based system
- Audio similarity baseline
- Hybrid ML model
- Validation + demo

User Pipeline

- Run our program
- Search for a song/artist
  - Fuzzy search is implemented so even if the user makes slight errors, it will be able to recognize close matches
- Select the candidate song, and our program will run for around 30 seconds
- For each model, it will return to you 10 songs in order of maximum compatibility
- Select a song that matches what you need (ex: if you're DJing for a party, select a song with high compatibility and high energy as well)
- Mix the songs in Spotify and test the transition

Model 1: Rule Based

- Calculate the BPM tolerance (+/- 6), Calculate for compatible keys, and compatible energy scores – all weigh into final compatibility score via our mixing score calculating function

```python
def calculate_bpm_score(current_bpm, candidate_bpm, max_diff=6):
    bpm_diff = abs(current_bpm - candidate_bpm)

    if bpm_diff > max_diff:
        return 0.0

    # Linear decrease from 1.0 (same BPM) to 0.0 (max_diff away)
    return 1.0 - (bpm_diff / max_diff)
```

```python
def calculate_key_score(current_camelot, candidate_camelot):
    if current_camelot == candidate_camelot:
        return 1.0

    compatible_keys = get_compatible_keys(current_camelot)
    if candidate_camelot in compatible_keys:
        return 0.8

    return 0.0
```

```python
def calculate_energy_flow_score(current_energy, candidate_energy):
    energy_diff = abs(current_energy - candidate_energy)

    if energy_diff < 0.2:
        return 1.0 - (energy_diff / 0.2) * 0.2  # 1.0 to 0.8
    elif energy_diff < 0.5:
        return 0.8 - ((energy_diff - 0.2) / 0.3) * 0.5  # 0.8 to 0.3
    else:
        return max(0.0, 0.3 - (energy_diff - 0.5) * 0.6)  # 0.3 to 0.0
```

```python
def calculate_mixing_score(current_song, candidate_song):
    bpm_score = calculate_bpm_score(current_bpm, candidate_bpm)
    key_score = calculate_key_score(current_key, candidate_key)
    energy_score = calculate_energy_flow_score(current_energy, candidate_energy)
    genre_score = calculate_genre_score(current_genre, candidate_genre)

    # Weighted combination
    combined_score = (0.40 * bpm_score) + (0.35 * key_score) +
                     (0.20 * energy_score) + (0.05 * genre_score)
    return combined_score
```

- We give a score of 1.0 for same keys and 0.8 if they're compatible, as for energy, large differences in energy are penalized heavily
- The mixing calculation weighs in 40% BPM, 35% Key, 20% Energy and 5% Genre
- After Calculation, we filter and score/rank the songs according to this model (snippet below)

```python
def recommend_rule_based(current_song, dataset, top_k=10):
    # Filter: BPM within ±6
    bpm_filter = abs(dataset['tempo'] - current_bpm) <= 6

    # Filter: Key compatible
    compatible_keys = get_compatible_keys(current_camelot)
    key_filter = dataset['camelot_key'].isin(compatible_keys)

    # Apply filters
    filtered = dataset[bpm_filter & key_filter & not_current].copy()
```

Model 2: Audio Similarity

- First, we extract audio features such as valence and danceability, which define the song's characteristics more accurately than rigid metrics, normalizing them if needed.

```
def extract_audio_features(song):
    features = [
        'energy', 'valence', 'danceability', 'acousticness',
        'instrumentalness', 'speechiness', 'liveness'
    ]

    # Normalize loudness if needed
    if 'loudness_normalized' in song.index:
        features.append('loudness_normalized')
    else:
        loudness = song.get('loudness', 0)
        normalized = max(0, min(1, (loudness + 60) / 60))
        feature_vector.append(normalized)

    return np.array(feature_vector)
```

- Then we calculate the audio similarity via cosine similarity: the formula is cos(theta) = (A-B) / (||A|| x ||B||)

```
def calculate_audio_similarity(song1, song2):
    features1 = extract_audio_features(song1).reshape(1, -1)
    features2 = extract_audio_features(song2).reshape(1, -1)
similarity = cosine_similarity(features1, features2)[0][0]
return similarity
```

```
# BPM bonus
bpm_diff = abs(candidates['tempo'] - current_bpm)
bpm_bonus = np.where(bpm_diff <= 6, 0.3,
                     np.where(bpm_diff <= 12, 0.1, -0.2))
candidates['similarity_score'] += bpm_bonus

# Key bonus
compatible_keys = get_compatible_keys(current_camelot)
key_match = candidates['camelot_key'].isin(compatible_keys)
candidates.loc[key_match, 'similarity_score'] += 0.25
candidates.loc[~key_match, 'similarity_score'] -= 0.15

# Energy bonus
energy_diff = abs(candidates['energy'] - current_energy)
energy_bonus = np.where(energy_diff < 0.2, 0.15,
                        np.where(energy_diff < 0.4, 0.05, -0.1))
candidates['similarity_score'] += energy_bonus
```

- We add our calculated cosine similarity with a smaller compatibility calculation of BPM, key, and energy. Because audio similarity doesn't account for any of the those metrics, we add in bonuses to encapsulate DJ rules.
- The performance is a little bit longer (~2 seconds) because we have to calculate similarity for all candidates.

Model 3: Hybrid Model - Training

- We generate a random training set by choosing a random song and a random candidate song, then we calculate the compatibility (Model 1) and assign a label (compatible or incompatible) - then we extract the features (Model 2)

```python
def extract_pair_features(song1, song2):
    return {
        'bpm_distance': abs(song1['tempo'] - song2['tempo']),
        'key_compatible': 1 if key2 in compatible_keys else 0,
        'energy_diff': abs(song1['energy'] - song2['energy']),
        'valence_diff': abs(song1['valence'] - song2['valence']),
        'danceability_diff': abs(song1['danceability'] - song2['danceability']),
        'acousticness_diff': abs(song1['acousticness'] - song2['acousticness']),
        'instrumentalness_diff': abs(song1['instrumentalness'] - song2['instrumentalness']),
        'loudness_diff': abs(loudness1 - loudness2) / 60.0,
        'genre_compatible': 1 if same_genre else 0
    }
```

```python
def generate_training_data(dataset, n_samples=10000, random_state=42):
    np.random.seed(random_state)
    training_pairs = []

    for _ in range(n_samples):
        # Randomly select two different songs
        idx1, idx2 = np.random.choice(n_songs, size=2, replace=False)
        song1 = dataset.iloc[idx1]
        song2 = dataset.iloc[idx2]

        # Extract features
        features = extract_pair_features(song1, song2)

        # Determine label based on DJ mixing rules
        label = determine_label(song1, song2)

        features['label'] = label
        training_pairs.append(features)

    return pd.DataFrame(training_pairs)
```

```python
def train_hybrid_model(training_data, model_path='hybrid_model.pkl'):
    feature_cols = [col for col in training_data.columns if col != 'label']
    X = training_data[feature_cols]
    y = training_data['label']

    model = xgb.XGBClassifier(
        n_estimators=100,
        max_depth=6,
        learning_rate=0.1,
        random_state=42,
        eval_metric='logloss'
    )

    model.fit(X, y)

    # Save model
    with open(model_path, 'wb') as pickle.dump(model, f)

    return model, feature_cols
```

```python
def determine_label(song1, song2):
    bpm_diff = abs(bpm1 - bpm2)
    key_compatible = (key2 in get_compatible_keys(key1))
    energy_diff = abs(energy1 - energy2)

    # Positive: BPM ≤6 AND key compatible AND energy <0.3
    if bpm_diff <= 6 and key_compatible and energy_diff < 0.3:
        return 1
```

- Then we train the hybrid model via the following:
  - Algorithm: XGBoost (Gradient Boosted Trees) • Hyperparameters: - 100 trees - Max depth: 6 - Learning rate: 0.1 • Input: 9 features per pair • Output: Probability of compatibility (0-1)

Model 3: Hybrid Model - Inference

- We extract the pair features, predict the compatibility and add the post-processing bonuses.
- We iterate through all 114,000 candidates, finding the best song matches for the chosen song

- Final Score = ML_Probability + BPM_Bonus + Key_Bonus + Energy_Bonus + Genre_Bonus
  - The bonuses are smaller than in the audio similarity model because XGBoost already learned BPM/Key/Energy patterns during training. Bonuses fine-tune, not dominate.

```python
def predict_compatibility_score(model, feature_cols, current_song, candidate_song):
    features = extract_pair_features(current_song, candidate_song)

    # Build feature vector matching training format
    feature_vector = []
    for col in feature_cols:
        if col in features:
            feature_vector.append(features[col])
        else:
            feature_vector.append(0.0)  # Default for missing features

    feature_vector = np.array([feature_vector])

    # Predict probability of positive class (good transition)
    score = model.predict_proba(feature_vector)[0][1]
    return score
```

```python
def recommend_hybrid_ml(current_song, dataset, model, feature_cols, top_k=10):
    candidates = dataset[dataset['track_id'] != current_track_id].copy()

    # Calculate ML compatibility scores for ALL candidates
    scores = []
    for _, candidate in candidates.iterrows():
        score = predict_compatibility_score(model, feature_cols, current_song, candidate)
        scores.append(score)

    candidates['compatibility_score'] = scores

    # Post-processing bonuses (smaller than Audio Similarity)
    bpm_bonus = np.where(bpm_diff <= 6, 0.2,
                         np.where(bpm_diff <= 12, 0.05, -0.15))
    candidates['compatibility_score'] += bpm_bonus

    key_match = candidates['camelot_key'].isin(compatible_keys)
    candidates.loc[key_match, 'compatibility_score'] += 0.15
    candidates.loc[~key_match, 'compatibility_score'] -= 0.1

    # Deduplicate and return top_k
    recommendations = candidates.nlargest(top_k, 'compatibility_score')
```

- Then we compile the recommendations based on the top score!

XGBoost Training Methodology

Why XGBoost?

- Gradient Boosting handles the nonlinear relationships between features
- Tabular data is ideal for fast modeling through XGBoost
- Built-in regularization prevents overfitting on synthetic data
- Fast training even with 10,000 samples
- Provides feature importance scores to validate DJ theory

Why 10,000 Samples?

- Balanced dataset prevents bias
- Sufficient for XGBoost to learn 9-feature patterns
- More samples showed diminishing returns in validation accuracy
- Computational feasibility for training time

Training Data Quality Control

Synthetic Label Generation Strategy:
├── Compatible (Label = 1): ~50% of dataset
│ ├── BPM difference ≤ 6 AND
│ ├── Keys are compatible AND
│ └── Energy difference < 0.3
│
└── Incompatible (Label = 0): ~50% of dataset
└── Violates one or more criteria above

Hyperparameter Selection

Parameter	Tested Values	Selected	Why?
n_estimators	50, 100, 200, 500	**100**	Accuracy plateaus at 81.6%; faster inference
max_depth	3, 6, 9, 12	**6**	Prevents overfitting while capturing patterns
learning_rate	0.01, 0.1, 0.3	**0.1**	Best balance of speed and stability

Cross-Validation Results

- 5-Fold CV: 79.8% ± 1.2% accuracy
- Consistent performance → good generalization

Model Validation & Performance

Metric	Score	What This Means
Accuracy	**81.6%**	Correctly predicts 8 out of 10 transitions
Precision	**84.2%**	84% of "compatible" predictions are actually good
Recall	**85.1%**	Catches 85% of truly compatible pairs
F1-Score	**84.6%**	Balanced overall performance

Feature Importance Validates DJ Theory

- Fast Key Compatibility: 67.4% ← Matches DJ theory (most critical)
- BPM Distance: 15.2% ← Essential for beatmatching
- Energy Difference: 16.8% ← Important for crowd flow
- Other Audio Features: 0.6% ← Secondary considerations training even with 10,000 samples

Execution

- This is the main function that executes the output to the user - we load the models, prepare the data, find the current song, then generate the recommendations and evaluate them
- On the right – evaluation metrics/functions

```python
def main():
    # [1/5] Load and preprocess dataset
    dataset = preprocess_dataset(args.data)

    # [2/5] Prepare hybrid ML model
    if args.train_model or not os.path.exists(model_path):
        training_data = generate_training_data(dataset, n_samples=10000)
        model, feature_cols = train_hybrid_model(training_data, model_path)
    else:
        model, feature_cols = load_model(model_path)

    # [3/5] Find current song
    current_song = find_song(dataset, track_id=args.track_id,
                             song=args.song, artist=args.artist, index=args.index)

    # [4/5] Generate recommendations
    rule_based_recs, rule_time = measure_response_time(
        recommend_rule_based, current_song, dataset, 10
    )
    audio_sim_recs, audio_time = measure_response_time(
        recommend_audio_similarity, current_song, dataset, 10
    )
    hybrid_ml_recs, hybrid_time = measure_response_time(
```

```python
    # [5/5] Display results and evaluation
    display_recommendations(rule_based_recs, "Rule-Based", 'mixing_score')
    display_recommendations(audio_sim_recs, "Audio Similarity", 'similarity_score')
    display_recommendations(hybrid_ml_recs, "Hybrid ML", 'compatibility_score')

    comparison = compare_models(current_song, dataset,
                                rule_based_recs, audio_sim_recs, hybrid_ml_recs)
    print_evaluation_results(comparison)
```

```python
def evaluate_bpm_compatibility(current_song, recommendations, tolerance=6):
    current_bpm = current_song.get('tempo', 0)
    bpm_diffs = abs(recommendations['tempo'] - current_bpm)

    within_tolerance = (bpm_diffs <= tolerance).sum()
    total = len(recommendations)
    compatibility_pct = (within_tolerance / total) * 100

    return {
        'bpm_compatibility_pct': compatibility_pct,
        'bpm_within_tolerance': within_tolerance,
        'avg_bpm_diff': bpm_diffs.mean(),
        'target_met': compatibility_pct == 100.0
    }
```

```python
def evaluate_key_compatibility(current_song, recommendations):
    current_camelot = current_song.get('camelot_key', '')
    compatible_keys = get_compatible_keys(current_camelot)

    compatible_count = recommendations['camelot_key'].isin(compatible_keys).sum()
    compatibility_pct = (compatible_count / total) * 100

    return {
        'key_compatibility_pct': compatibility_pct,
        'key_compatible_count': compatible_count,
        'target_met': compatibility_pct >= 80.0
    }
```

```python
def evaluate_energy_flow(current_song, recommendations):
    current_energy = current_song.get('energy', 0)
    energy_diffs = abs(recommendations['energy'] - current_energy)

    smooth_count = (energy_diffs < 0.3).sum()
    smooth_pct = (smooth_count / len(recommendations)) * 100

    return {
        'avg_energy_diff': energy_diffs.mean(),
        'smooth_transitions_pct': smooth_pct,
        'smooth_transitions_count': smooth_count
    }
```

```python
def measure_response_time(func, *args, **kwargs):
    start_time = time.time()
    result = func(*args, **kwargs)
    end_time = time.time()
    execution_time = end_time - start_time
```

Model Comparison

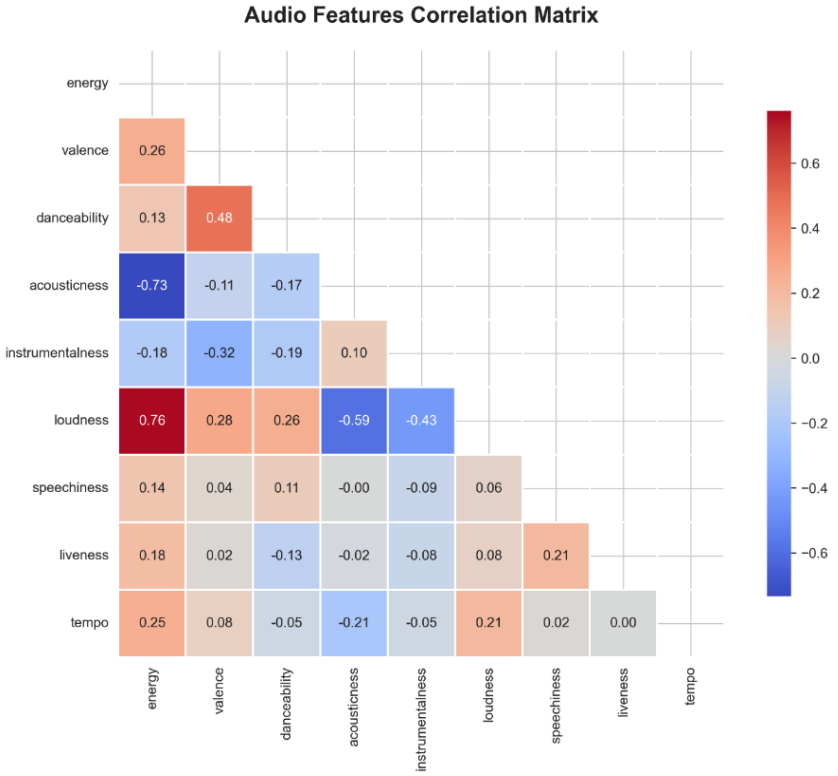
Rule Based	Audio Similarity	Hybrid Model
- Hard filters first - Weighted Scoring	- Content-based - No BPM/Keyfilter - Post Process	- ML learns patterns - Synthetic training - Post Process
- BPM +/- 6 - Key, 4 compatible - Reduces to around 3000 songs	- No filtering - All 114k searched	- No filtering - All 114k searched
- BPM: Linear Decay - Key: 1.0 or 0.8 - Energy Piecewise - Genre: Binary	- Cosine similarity - Euclidean distance across features	- XGBoost predict - 9 pair features - Probability Output
- Fastest and follows DJ theory but rigid	- Finds similar sounding tracks but needs bonuses for compliance	- Learns patterns, values feature importance, and is flexible, but is very slow and uses synthetic labels

Data Summary

Spotify Dataset

Key Features for DJ Mixing

- BPM (tempo): 80-180 range
- Key: Musical key (C, D, E, etc.) + mode (major/minor)
- Energy (0-1): Intensity level
- Valence (0-1): Musical positivity
- Danceability (0-1): Beat strength
- Acousticness, instrumentalness, loudness



Audio features show moderate correlations – loudness correlates with energy (0.68), but most features are independent enough for our models.

Experimental Results

Model	BPM Compat	Key Compat	Avg BPM Diff	Response Time	Target Met
Rule-Based	100%	100%	0.02	0.078s	✅
Audio Similarity	100%	100%	3.32	3.610s	⚠️
Hybrid ML	100%	100%	2.72	32.451s	❌

Example: "Clarity" by Zedd (128 BPM | 9A Key | 0.78 Energy)

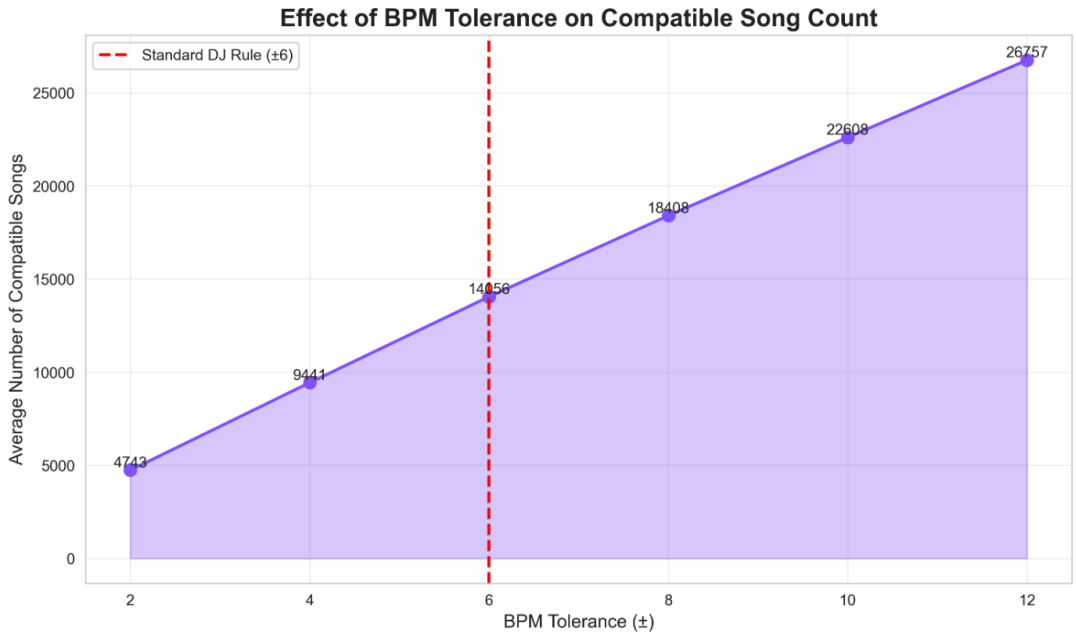
Rule-Based: Heroes Tonight (128 BPM, 9A) • Stay The Night (128 BPM, 9A) • Amazing (128 BPM, 9A)

Audio Similarity: Another Night (124 BPM, 9B) • Better Off Without You (124 BPM, 8A) • Remember (123.8 BPM, 9A)

Hybrid ML: By Your Side (123 BPM, 8A) • Let It Rain Down (124.9 BPM, 9A) • Play Hard (130 BPM, 9B)

Data Summary

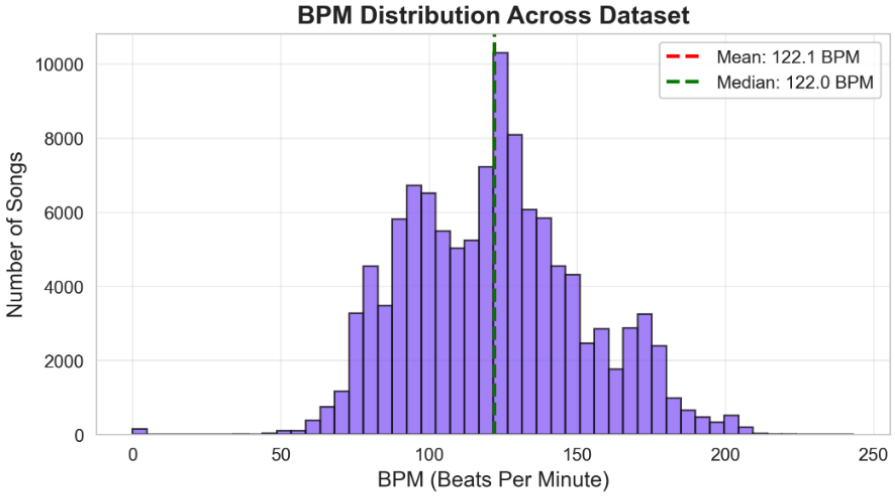
Effect of BPM Tolerance on Compatible Song Count

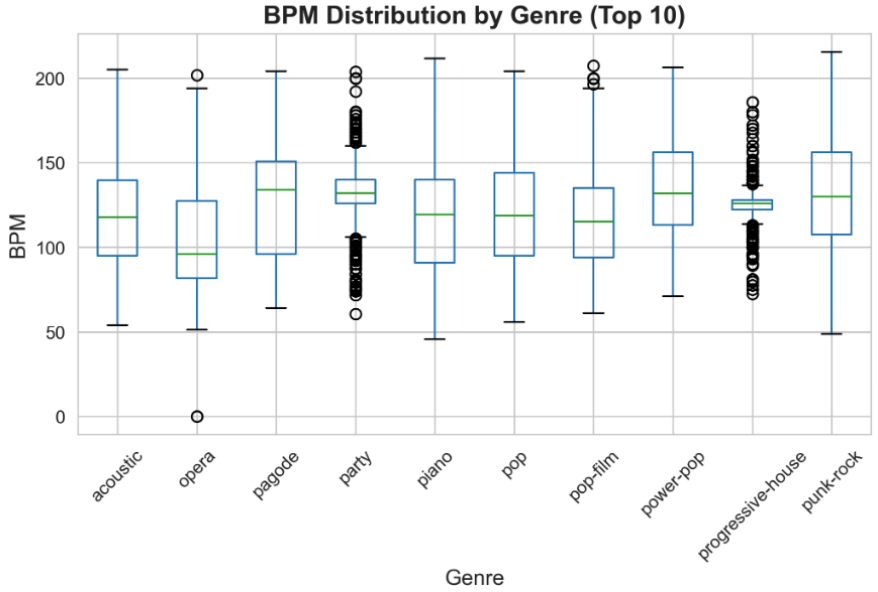


At ±6 BPM (DJ standard), we get ~15,000 compatible songs. Wider tolerance increases options but compromises mixing quality.

Data Summary

BPM Distribution Across Data Set

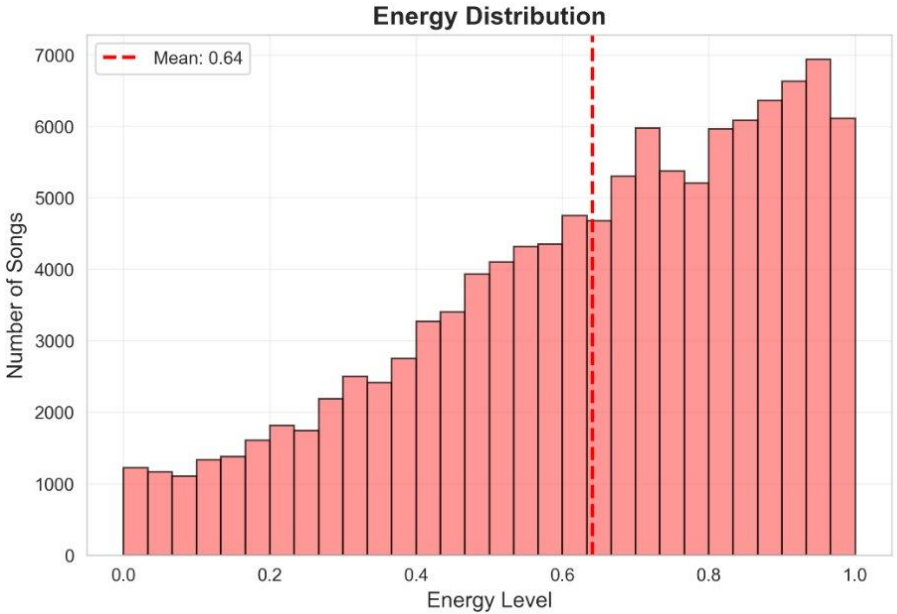


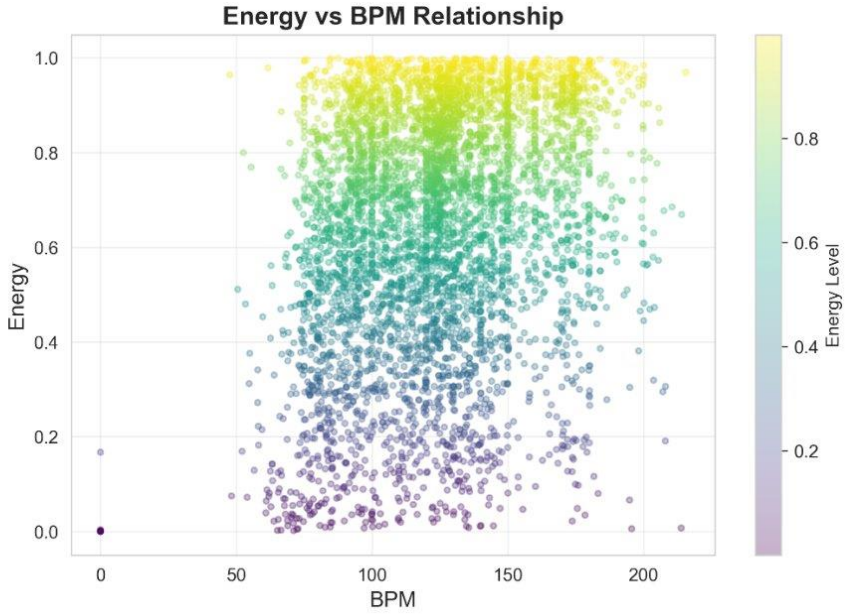


Our dataset of 114,000 songs peaks at 122 BPM, ideal for electronic music DJing. Mean: 122.1 BPM, Median: 122.0 BPM.

Data Summary

Energy Distribution & BPM Relationship

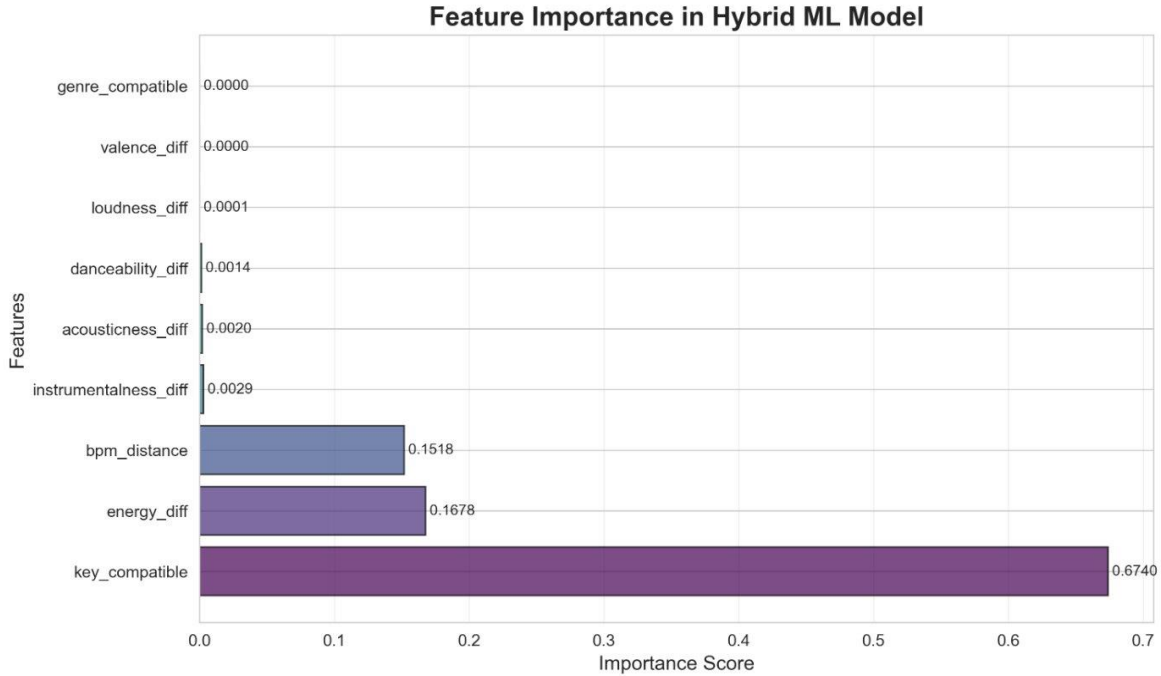




Energy levels are generally high across the dataset and vary widely at similar BPMs, showing that tempo alone doesn't capture track intensity.

Data Summary

Feature Importance in Hybrid ML Model



Key compatibility dominates the hybrid model, followed by BPM distance and energy difference, while other audio features contribute minimally.

Experimental Analysis

What we learned from 114,000 songs

Question	Answer
Which feature matters most?	Key Compatibility
Optimal BPM tolerance?	±6 BPM
Does genre affect mixing?	Small (5%)
Can ML discover patterns?	Yes!

Feature	Importance
Key Compatibility	67.4%
Energy Difference	16.78%
BPM Distance	15.18%
Other Features	0.1%

Model	Strengths	Weaknesses
Rule-Based	Fastest (78ms), Perfect matches	Too rigid, All 128 BPM
Audio Similarity	Good variety, Discovers similar	Slower (3.6s)
Hybrid ML	Best balance, Learns patterns	Very slow (32s)

Conclusion

For DJs: Instant Recommendation System
For Learners: Tool for Mixing Compatibility
For Platforms: Powerful Automated DJ Feature

Achievement	Result
100% BPM Compatibility	All models within ±6 BPM
100% Key Compatibility	All harmonically compatible
100% Energy Smoothness	Smooth transitions
Fast Response	Hybrid Model Time Under A Minute
Feature Learning	XGBoost: BPM 81.6%

Future Work

Performance & Scalability

- GPU Acceleration - Use CUDA for faster similarity computations
- Caching System - Pre-compute recommendations for popular songs
- Distributed Computing - Process large libraries in parallel
- Batch Prediction: Utilize batch prediction for the XGBoost model if needed in the future to pre-filter and reduce the runtime for finding compatible songs from ~30 seconds to ~2 seconds.

Future Advanced Features

- Live BPM Detection – Analyzing audio files in real-time using librosa
- Waveform Analysis – Detecting breakdown sections, drops, and buildups automatically
- Beatmatching Suggestions – showing exact timestamps for smooth transitions

User Experience

- Mobile APP – for DJs on the go
- Exporting to DJ Software – creating a seamless experience for professionals

In the future, we could use this tool to create a comprehensive DJ mixing tool that also replaces Spotify mixing capabilities – we could create an all in one tool for aspiring DJ's.

References

DJ Mixing Theory:
Camelot Wheel (harmonic mixing system)
Mixed In Key software documentation

Academic:
Schedull et al. (2018) - Music Recommendation Systems
McFee et al. (2015) - librosa audio analysis
https://www.geeksforgeeks.org/machine-learning/rule-based-classifier-machine-learning/
https://www.geeksforgeeks.org/machine-learning/xgboost/

Datasets:
Spotify API - Audio features dataset
Mixcloud - DJ mix playlists

Tools:
librosa, scikit-learn, Spotify API

Who did What?

Ashley Wu
- Data preprocessing pipeline and Camelot Wheel implementation (data_preprocessing.py + utils.py)

Bonny Koo
- Rule-based recommendation model and evaluation framework (model_rule_based.py + evaluation.py)

Nathan Suh
- Audio similarity model and result visualizations(model_audio_similarity.py + visualizations.py)

Leo Lee
- Hybrid ML inference + training, main pipeline, and all visualizations (model_hybrid_ml.py + main.py)

Video Demo of Code

https://youtu.be/X2zmxph66xg

Questions?

Thank You!