UC San Diego

CSE 253 - Assignment 2

Task 1: Symbolic - Unconditioned Generation
Task 2: Symbolic - Conditioned Generation

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Computer Science and Engineering

Assignment 2 - Tasks

Task 1: Symbolic - Unconditioned Generation	Task 2: Symbolic - Conditioned Generation
• Train a model to learn a music distribution *p(x)* from a given dataset	• Train a model to generate music conditioned on some input (learn a conditional distribution *p(x | input)*)
Both will generate new music by sampling directly from that learned distribution	

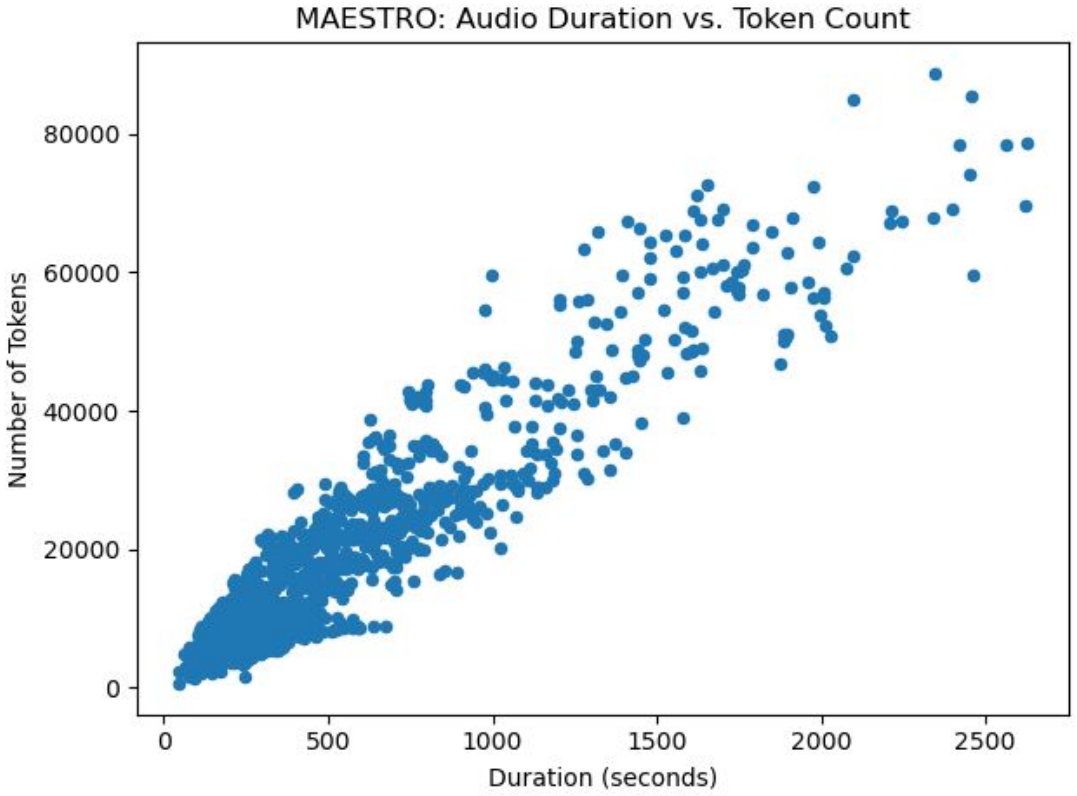
MAESTRO Dataset (Context)

- Recorded at the 2018–2019 International Piano-e-Competition, capturing both MIDI data (keys, pedals, volume) and matching high-quality piano audio.
- Every performance includes a WAV audio file and a perfectly synced MIDI file, plus basic details (composer, piece name, performer, date).
- Covers about 200 hours of piano music across roughly 1,100 performances, with works by Chopin, Liszt, Debussy, Beethoven, Rachmaninoff, and more.
- Helps train and test models that turn piano recordings into MIDI (transcription), recreate human-like playing, or generate new piano audio/MIDI.
- Comes split into 80 % train, 10 % validation, and 10 % test sets so researchers compare results on the same data.

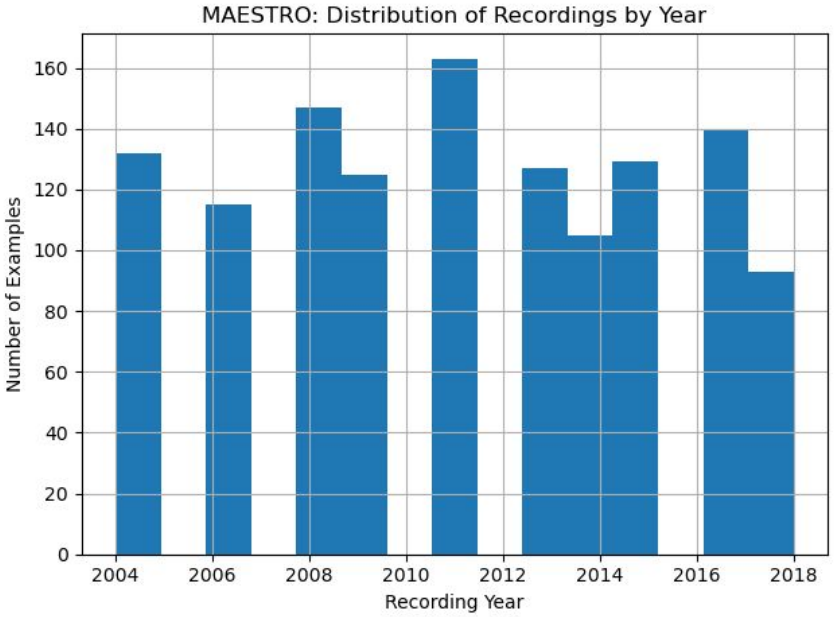
MAESTRO Dataset (Discussion)

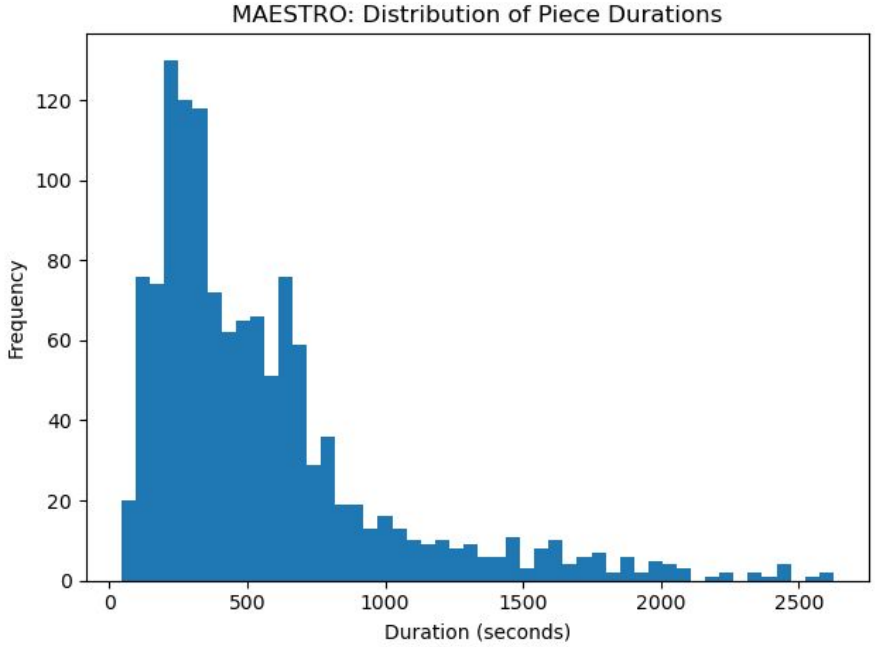
- Data is pre-processed; all alignments, cleaning, and splitting is done already.
- Audio (44.1 kHz/24-bit WAV) and MIDI (1 kHz timestamps) have been time-warped and manually checked so every MIDI event matches its audio.
- Composer, piece title, performer, date, and competition round are standardized in a CSV—no spelling fixes or merges needed.
- Each WAV/MIDI pair is cut exactly to the start/end of the performance (no extra silence or warm-up).
- The 80 % train, 10 % validation, 10 % test partitions are already assigned in the metadata—no manual splitting required.

Exploratory Analysis - MAESTRO (Code, 1)

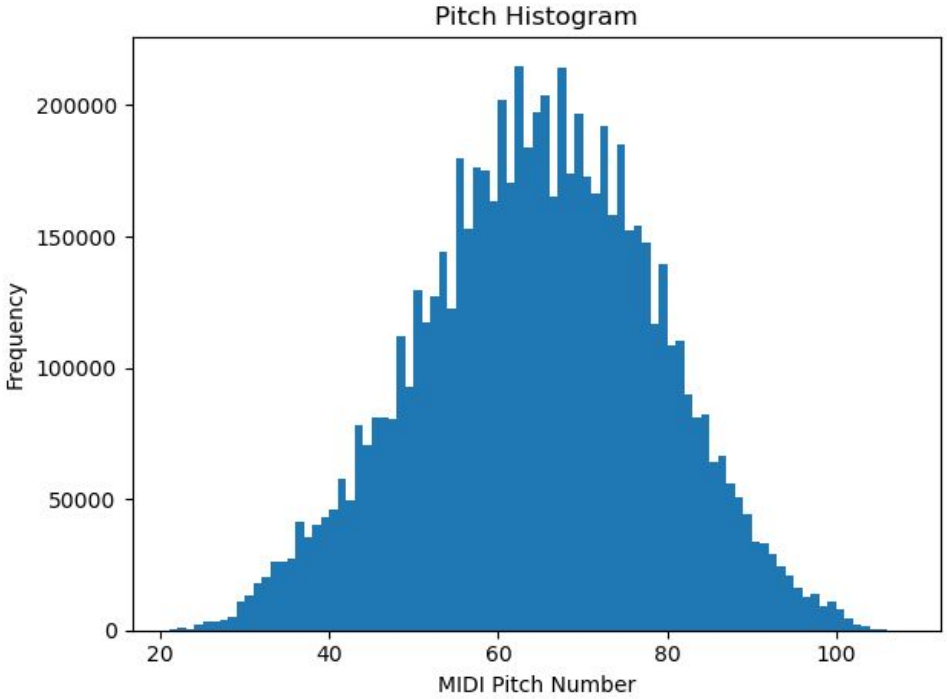


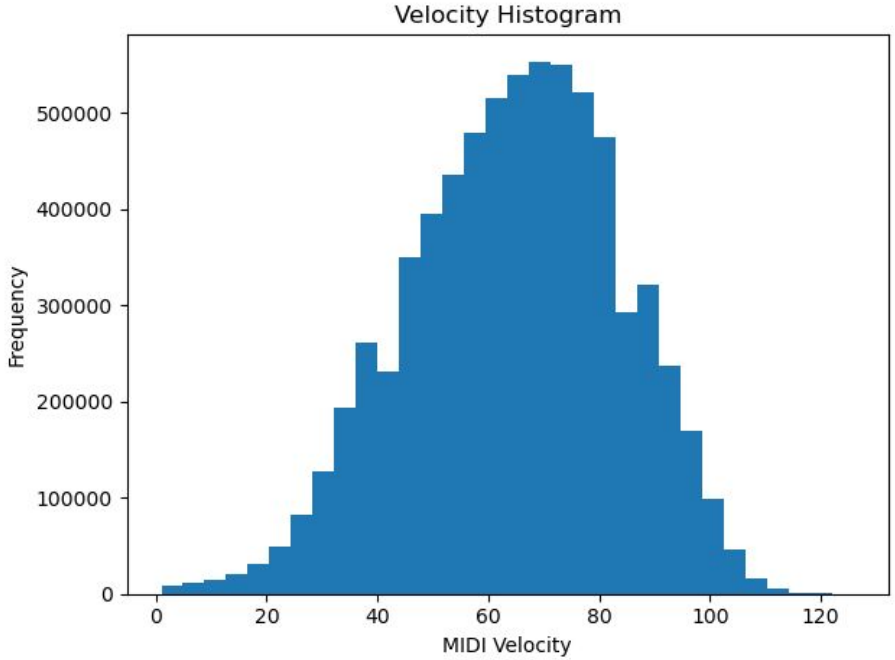
Exploratory Analysis - MAESTRO (Code, 2)





Exploratory Analysis - MAESTRO (Code, 3)





Model - MAESTRO (Context)

- **Inputs:** REMI-tokenized MIDI sequences (JSON) → `input_ids` (length ≤ 512)
- **Outputs:** Next-token prediction for positions ≥ 128 (masked with –100 for the prefix)
- **Loss:** Cross-entropy over unmasked token positions
- **Model:** GPT-2 (8 layers, 768-dim embeddings, 12 heads) for autoregressive sampling
- **Batching:** Pad with `pad_token_id=0` / `labels=-100` via `CollatorForAutoregressive`
- **Generation:** Autoregressively sample one token at a time and decode back to MIDI

Model - MAESTRO (Discussion)

- We used a Transformer (GPT-2) for strong long-range modeling at the cost of higher compute/memory.
- RNN/LSTM would reduce memory needs but weaken long-term dependency capture and slow down training.
- CNN (WaveNet-style) offers a middle ground—parallel training with bounded receptive fields, but more manual design.
- Markov chain is simplest—easy to implement, but fails on musical coherence beyond short n-grams.
- More complex latent-space (VAE) or masking (BERT) approaches exist but introduce extra implementation overhead without guaranteed gains in autoregressive MIDI generation.

Model - MAESTRO (Code, 1)

```
'''
Function for tokenizing all MIDI files in a given path
'''
def tokenize_midi_files(midi_path, token_path):
    # ensure token path exists (if not create it)
    token_path.mkdir(parents=True, exist_ok=True)
    # Process all MIDI files in the current year's directory
    for midi_file in midi_path.rglob('*.midi'):
        try:
            midi = MidiFile(midi_file)
            tokens = tokenizer(midi)

            # Each file produces a list of TokSequence; flatten to a single track list
            all_tracks = [seq.ids for seq in tokens]
            assert len(all_tracks) == 1, f"{midi_file.name} has more than one track."

            # Save as JSON
            output_file = token_path / (midi_file.stem + ".json")
            with open(output_file, "w") as f:
                json.dump({"tracks": all_tracks}, f)
        except Exception as e:
            print(f"Failed to process {midi_file.name}: {e}")

'''Define directories for tokenization'''
# process years 2008 thru 2018 for training data
train_years = ['2008', '2009','2011', '2013', '2014', '2015', '2016', '2017', '2018']
test_years = ['2004']
validation_years = ['2006']

train_dir = token_base / "train"
val_dir   = token_base / "val"
test_dir  = token_base / "test"
```

Model - MAESTRO (Code, 2)

```python
class MelodyContinuationDataset(Dataset):
    def __init__(self, token_dir, prefix_len=128, max_length=1024):
        self.files = list(Path(token_dir).glob("*.json"))
        self.prefix_len = prefix_len
        self.max_length = max_length

    def __len__(self):
        return len(self.files)

    def __getitem__(self, idx):
        with open(self.files[idx]) as f:
            data = json.load(f)

        tokens = data["tracks"][0]  # Get the first track's tokens

        # Truncate or pad to max length
        tokens = tokens[:self.max_length]
        input_ids = torch.tensor(tokens, dtype=torch.long)

        # Prepare labels
        labels = input_ids.clone()

        # Mask the prefix portion from loss
        labels[:self.prefix_len] = -100

        return {
            "input_ids": input_ids,
            "labels": labels
        }
```

```python
class CollatorForAutoregressive:
    def __init__(self, pad_token_id=0):
        self.pad_token_id = pad_token_id

    def __call__(self, batch):
        input_ids = [torch.tensor(x["input_ids"]) for x in batch]
        labels = [torch.tensor(x["labels"]) for x in batch]

        input_ids = pad_sequence(input_ids, batch_first=True, padding_value=self.pad_token_id)
        labels = pad_sequence(labels, batch_first=True, padding_value=-100)

        return {"input_ids": input_ids, "labels": labels}
```

Model - MAESTRO (Code, 3)

```python
from transformers import GPT2Config, GPT2LMHeadModel

config = GPT2Config(
    vocab_size=tokenizer.vocab_size,
    n_embd=768, # originally 512
    n_layer=8, # originally 6
    n_head=12, # originally 8
    pad_token_id=0
)
model = GPT2LMHeadModel(config)
```

```python
from transformers import TrainingArguments, Trainer

# print which device is being used
print(f"Using device: {device}")
model.to(device)

training_args = TrainingArguments(
    output_dir="./gpt2-music",
    per_device_train_batch_size=4, #originally 2
    num_train_epochs=25,
    save_steps=500,
    logging_steps=50,
    fp16=True,
    report_to="none",
    gradient_checkpointing=True,
    save_total_limit=3,
    lr_scheduler_type="cosine",
    warmup_steps=1000,
    weight_decay=0.01,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    data_collator=collator
)

trainer.train()
# -- save the final weights (and config) to a clean folder --
final_dir = "./gpt2-music-final"
trainer.save_model(final_dir)
```

Model - MAESTRO (Code, 4)

```
'''Use an example from test set to generate continuation'''
import torch.nn.functional as F
from symusic import Score
from miditok.classes import TokSequence

def generate_continuation(model, prefix_ids, max_new_tokens=200, temperature=1.0, top_k=50):
    model.eval()
    prefix_ids = prefix_ids.to(device)
    input_ids = prefix_ids.clone()

    with torch.no_grad():
        for _ in range(max_new_tokens):
            logits = model(input_ids).logits
            next_token_logits = logits[:, -1, :] / temperature
            filtered_logits = next_token_logits  # no filtering
            next_token = torch.multinomial(F.softmax(filtered_logits, dim=-1), num_samples=1)
            input_ids = torch.cat([input_ids, next_token], dim=1)

    full_ids = input_ids[0].tolist()            # prefix + continuation
    cont_ids = full_ids[prefix_ids.shape[1]:]   # slice off the prefix
    prefix_ids = full_ids[:prefix_ids.shape[1]]      # the prefix
    return prefix_ids, cont_ids, full_ids
```

```
'''CALL GENERATION AND SAVE MIDI FILES'''
test_dataset = MelodyContinuationDataset(test_dir, prefix_len=128, max_length=512)
N = 10
samples  = [test_dataset[i] for i in range(N)] # Takes N samples
#prefix       = sample["input_ids"][:128].unsqueeze(0)
prefixes = [sample["input_ids"][:128].unsqueeze(0) for sample in samples]

#prefix_ids, generated_ids, full_ids = generate_continuation(model, prefix) # TODO: have a way to discern prefix from continuation (return 2 midi files)

prefix_ids_list = []
generated_ids_list = []
full_ids_list = []

for sample, prefixes in zip(samples, prefixes):
    prefix_ids, generated_ids, full_ids = generate_continuation(model, prefixes)
    prefix_ids_list.append(prefix_ids)
    generated_ids_list.append(generated_ids)
    full_ids_list.append(full_ids)
```

Evaluation - MAESTRO (Context, 1)

Objective metrics (e.g., perplexity, cross-entropy)

– Quantify how well the model predicts the next token/note

– Useful for training convergence but don’t guarantee musicality

Musical-structural metrics

– Tonal/harmonic coherence (e.g., adherence to key, chord progression rules)

– Rhythmic consistency and meter alignment

– Note-level diversity versus repetition (avoiding monotony yet maintaining style)

Subjective evaluation

– Human listening tests or expert musician ratings

– Assess emotional impact, creativity, and perceived “flow”

– Gauges whether the output feels stylistically authentic

Evaluation - MAESTRO (Context, 2)

UC San Diego

Properties of a "good" output

– **Melodic coherence**: smooth, logical pitch transitions

– **Harmonic plausibility**: chords and intervals that fit the intended key or genre

– **Rhythmic groove**: clear pulse and engaging rhythmic patterns

– **Balanced novelty**: fresh ideas without sounding random or disjointed

Perplexity vs. musical properties

– Lower perplexity means the model fits training data better but may over-emphasize common patterns

– A model can have low perplexity yet produce harmonically incorrect or un-interesting music

– Perplexity does not capture higher-level features like emotional contour or genre style

Why combine objectives and subjective measures?

– Use perplexity (or related loss) for gradient-based optimization and early stopping

– Validate with musical metrics (e.g., tonal stability, note-set distributions) to catch rule violations

– Rely on human evaluation to ensure outputs resonate emotionally and stylistically with listeners

Evaluation - MAESTRO (Discussion)

- **Baseline**
  - Created full evaluation pipeline for “PyTorch-generated” symbolic music (MIDI) by comparing it against baseline MAESTRO statistics, computing a variety of musical-quality metrics, producing both textual reports and visualizations, and optionally comparing multiple generated outputs side by side.

- Compare the results/plots to our model and determine overall quality scores and which is more musically better.

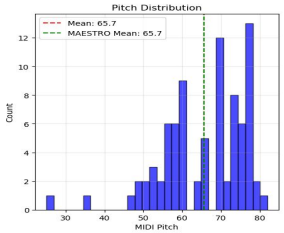
Evaluation - MAESTRO (Code, 1)

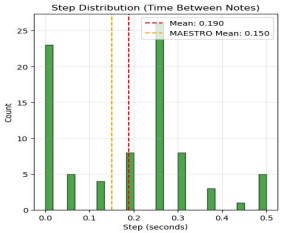
```python
def create_evaluation_visualizations(self, notes: pd.DataFrame, output_dir: str = '.') -> None:
    """Create comprehensive evaluation visualizations"""
    if len(notes) == 0:
        print("No notes to visualize")
        return

    # Set style
    plt.style.use('default')
    sns.set_palette("husl")

    # Create main figure
    fig = plt.figure(figsize=(20, 16))

    # 1. Pitch distribution and statistics
    plt.subplot(3, 4, 1)
    plt.hist(notes['pitch'], bins=30, alpha=0.7, color='blue', edgecolor='black')
    plt.axvline(notes['pitch'].mean(), color='red', linestyle='--',
                label=f'Mean: {notes["pitch"].mean():.1f}')
    if self.original_stats:
        plt.axvline(self.original_stats['pitch_mean'], color='green', linestyle='--',
                    label=f'MAESTRO Mean: {self.original_stats["pitch_mean"]:.1f}')
    plt.title('Pitch Distribution')
    plt.xlabel('MIDI Pitch')
    plt.ylabel('Count')
    plt.legend()
    plt.grid(True, alpha=0.3)

    # 2. Step (timing) distribution
    plt.subplot(3, 4, 2)
    step_data = notes['step'][notes['step'] <= np.percentile(notes['step'], 95)]  # Remove outliers
    plt.hist(step_data, bins=30, alpha=0.7, color='green', edgecolor='black')
    plt.axvline(notes['step'].mean(), color='red', linestyle='--',
                label=f'Mean: {notes["step"].mean():.3f}')
    if self.original_stats:
        plt.axvline(self.original_stats['step_mean'], color='orange', linestyle='--',
                    label=f'MAESTRO Mean: {self.original_stats["step_mean"]:.3f}')
    plt.title('Step Distribution (Time Between Notes)')
    plt.xlabel('Step (seconds)')
    plt.ylabel('Count')
    plt.legend()
    plt.grid(True, alpha=0.3)

    # 3. Duration distribution
    plt.subplot(3, 4, 3)
    duration_data = notes['duration'][notes['duration'] <= np.percentile(notes['duration'], 95)]
    plt.hist(duration_data, bins=30, alpha=0.7, color='purple', edgecolor='black')
    plt.axvline(notes['duration'].mean(), color='red', linestyle='--',
                label=f'Mean: {notes["duration"].mean():.3f}')
    if self.original_stats:
        plt.axvline(self.original_stats['duration_mean'], color='orange', linestyle='--',
                    label=f'MAESTRO Mean: {self.original_stats["duration_mean"]:.3f}')
    plt.title('Duration Distribution')
    plt.xlabel('Duration (seconds)')
    plt.ylabel('Count')
    plt.legend()
    plt.grid(True, alpha=0.3)
```

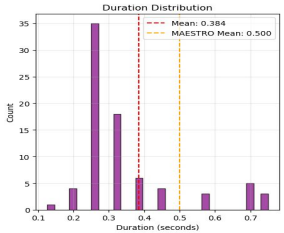
```python
# 4. Piano roll visualization
plt.subplot(3, 4, 4)
sample_notes = notes.head(100)  # First 100 notes
for i, note in sample_notes.iterrows():
    plt.barh(note['pitch'], note['duration'], left=note['start'],
             height=0.8, alpha=0.7, color='blue')
plt.title('Piano Roll (First 100 Notes)')
plt.xlabel('Time (seconds)')
plt.ylabel('MIDI Pitch')
plt.grid(True, alpha=0.3)

# 5. Melodic intervals
plt.subplot(3, 4, 5)
if len(notes) > 1:
    intervals = np.diff(notes['pitch'])
    plt.hist(intervals, bins=range(-24, 25), alpha=0.7, color='red', edgecolor='black')
    plt.axvline(0, color='black', linestyle='-', alpha=0.5)
    plt.title('Melodic Intervals')
    plt.xlabel('Semitone Interval')
    plt.ylabel('Count')
    plt.xlim(-12, 12)
    plt.grid(True, alpha=0.3)

# 6. Pitch class distribution
plt.subplot(3, 4, 6)
pitch_classes = notes['pitch'] % 12
pc_counts = np.bincount(pitch_classes, minlength=12)
note_names = ['C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B']
bars = plt.bar(range(12), pc_counts, alpha=0.7, color='orange')
plt.title('Pitch Class Distribution')
plt.xlabel('Pitch Class')
plt.ylabel('Count')
plt.xticks(range(12), note_names)
plt.grid(True, alpha=0.3)

# Add count labels on bars
for i, (bar, count) in enumerate(zip(bars, pc_counts)):
    plt.text(bar.get_x() + bar.get_width()/2., bar.get_height() + 0.5,
             str(count), ha='center', va='bottom', fontsize=8)

# 7. Quality metrics radar chart
plt.subplot(3, 4, 7, projection='polar')
quality_metrics = self.calculate_musical_quality_metrics(notes)

metrics = ['Melodic\nCoherence', 'Pitch Range\nValidity', 'Duration\nValidity', 'Step\nValidity']
values = [
    quality_metrics.get('melodic_coherence', 0),
    quality_metrics.get('pitch_range_validity', 0),
    quality_metrics.get('duration_validity', 0),
    quality_metrics.get('step_validity', 0)
]

angles = np.linspace(0, 2*np.pi, len(metrics), endpoint=False)
values += values[:1]  # Complete the circle
angles = np.concatenate((angles, [angles[0]]))

plt.plot(angles, values, 'o-', linewidth=2, color='red')
plt.fill(angles, values, alpha=0.25, color='red')
plt.thetagrids(angles[:-1] * 180/np.pi, metrics)
plt.ylim(0, 1)
plt.title('Quality Assessment\n(1.0 = Perfect)', pad=20)

# 8. Note velocity distribution (if available)
plt.subplot(3, 4, 8)
# Create synthetic velocity data based on pitch
velocities = 80 + (notes['pitch'] - notes['pitch'].mean()) * 0.3 + np.random.normal(0, 10, len(notes))
velocities = np.clip(velocities, 40, 120)
plt.hist(velocities, bins=20, alpha=0.7, color='brown', edgecolor='black')
plt.title('Estimated Velocity Distribution')
plt.xlabel('Velocity')
plt.ylabel('Count')
plt.grid(True, alpha=0.3)

# 9. Timing analysis
plt.subplot(3, 4, 9)
plt.scatter(notes['step'], notes['duration'], alpha=0.6, s=20)
plt.title('Step vs Duration Relationship')
plt.xlabel('Step (seconds)')
plt.ylabel('Duration (seconds)')
plt.grid(True, alpha=0.3)

# 10. Pitch trajectory
plt.subplot(3, 4, 10)
plt.plot(notes.index[:100], notes['pitch'][:100], 'b-', alpha=0.7, linewidth=1)
plt.scatter(notes.index[:100], notes['pitch'][:100], alpha=0.5, s=10, c='red')
plt.title('Pitch Trajectory (First 100 Notes)')
```

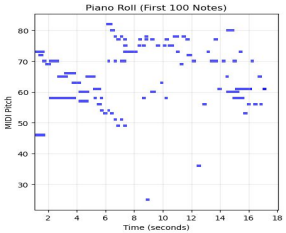
```python
# 10. Pitch trajectory
plt.subplot(3, 4, 10)
plt.plot(notes.index[:100], notes['pitch'][:100], 'b-', alpha=0.7, linewidth=1)
plt.scatter(notes.index[:100], notes['pitch'][:100], alpha=0.5, s=10, c='red')
plt.title('Pitch Trajectory (First 100 Notes)')
plt.xlabel('Note Index')
plt.ylabel('MIDI Pitch')
plt.grid(True, alpha=0.3)

# 11. Cumulative duration
plt.subplot(3, 4, 11)
cumulative_time = notes['step'].cumsum()
plt.plot(cumulative_time, alpha=0.7, color='green')
plt.title('Cumulative Time Progression')
plt.xlabel('Note Index')
plt.ylabel('Cumulative Time (seconds)')
plt.grid(True, alpha=0.3)

# 12. Summary statistics text
plt.subplot(3, 4, 12)
plt.axis('off')

basic_stats = self.calculate_basic_statistics(notes)
quality_metrics = self.calculate_musical_quality_metrics(notes)

summary_text = f"""
```

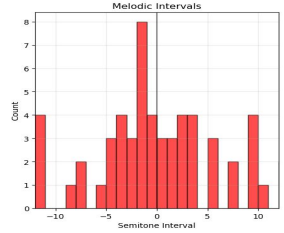
Evaluation - MAESTRO (Code, 2)

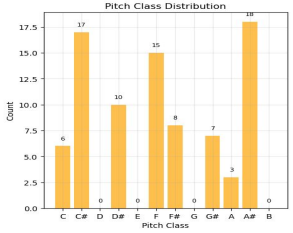


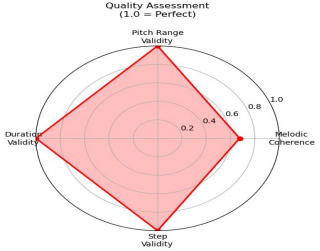


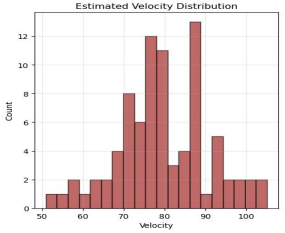


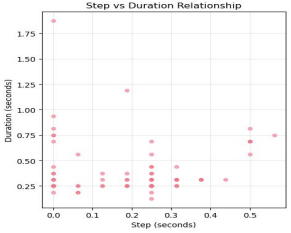


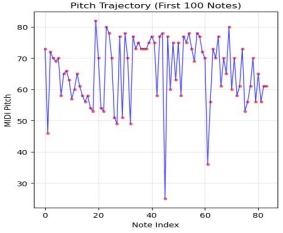


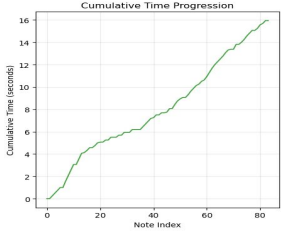













```
EVALUATION SUMMARY

Total Notes: 84
Duration: 17.2s
Pitch Range: 25-82
Unique Pitches: 24

QUALITY SCORES:
Melodic Coherence: 0.675
Pitch Validity: 1.000
Duration Validity: 1.000
Step Validity: 1.000

OVERALL QUALITY: 0.919/1.00
```

```
1. BASIC STATISTICS
------------------------------------------------------------
Total Notes: 84
Total Duration: 17.25 seconds
Average Notes per Second: 4.87

Pitch Statistics:
  Range: 25-82 (span: 57 semitones)
  Mean: 65.7 ± 10.7
  Unique Pitches: 24 (28.6% variety)

Timing Statistics:
  Step (between notes): 0.190 ± 0.152 seconds
  Duration (note length): 0.384 ± 0.256 seconds

2. QUALITY ASSESSMENT
------------------------------------------------------------
Melodic Coherence: 0.675
  Step Motion Ratio: 0.265
  Large Leap Ratio: 0.277

Validity Scores:
  Pitch Range Validity: 1.000
  Duration Validity: 1.000
  Step Validity: 1.000

Pitch Diversity:
  Unique Pitches: 24
  Pitch Entropy: 2.925
  Most Common Pitch Ratio: 0.119

OVERALL QUALITY SCORE: 0.919/1.00
  ✅ VERY GOOD: High-quality music generation!

3. COMPARISON WITH MAESTRO DATASET
------------------------------------------------------------
Pitch Similarity: 0.998
  Mean Difference: 0.0 semitones
Timing Similarity:
  Step Similarity: 0.868
  Duration Similarity: 0.710

Overall MAESTRO Similarity: 0.821
```

Discussion - Related Work

- **Previous Use:** Magenta's "Onsets and Frames", Google's Performance RNN, OpenAI's MuseNet and Jukebox

- **Prior Work:** RNN/LSTM-based Sequence Models, Variational Autoencoders(VAEs) & GANs, Transformer-based Models

- **Differences:**
  - Our model's use of absolute position embeddings contrasts with relative attention (Music Transformer), which can better generalize to unseen sequence lengths.
  - Data split by year helps test out-of-distribution generalization (e.g. 2004 performances), whereas many papers shuffle entire corpora when splitting.
  - We did not incorporate additional conditioning (e.g. composer/style tags), while some related works demonstrate style transfer or user-controlled generation.

Maestro v2.0 Dataset (Context)

- Large-Scale MIDI Collection: 176K+ unique MIDI files, with 45K matched to the Million Song Dataset (MSD).
- Rich Annotations: Includes tempo, key, time signature, and some lyrics.
- Subset Breakdown: LMD-full (all files), LMD-matched (linked to MSD), LMD-aligned (time-synced with audio).
- For Music AI Tasks: Enables transcription, genre classification, alignment, and generation.
- Free to Use: Open under CC-BY 4.0; cite Raffel (2016) and MSD (2011) when using.

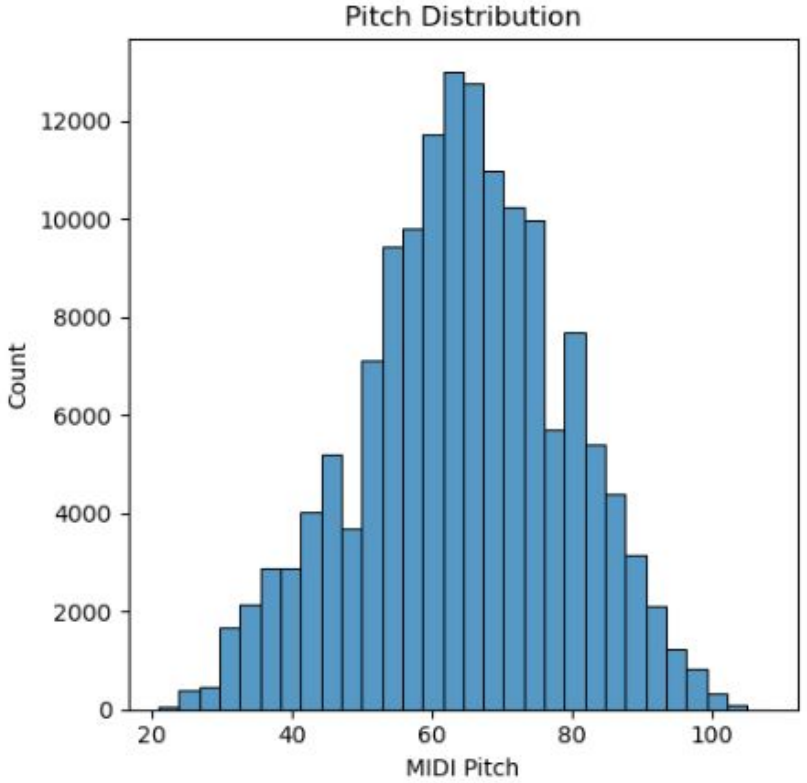
Maestro v2.0 Dataset (Discussion)

UC San Diego

- Processed 1,282 MIDI files and extracted ~150K individual note events.
- Computed and analyzed pitch, step (time between notes), and duration statistics.
- Pitch range: 21–108; average note duration: 0.20 sec; average step time: 0.10 sec.
- Visualized distributions of pitch, step, and duration to validate data quality.
- Saved note-level features (pitch, start, end, step, duration) to CSV for downstream tasks.

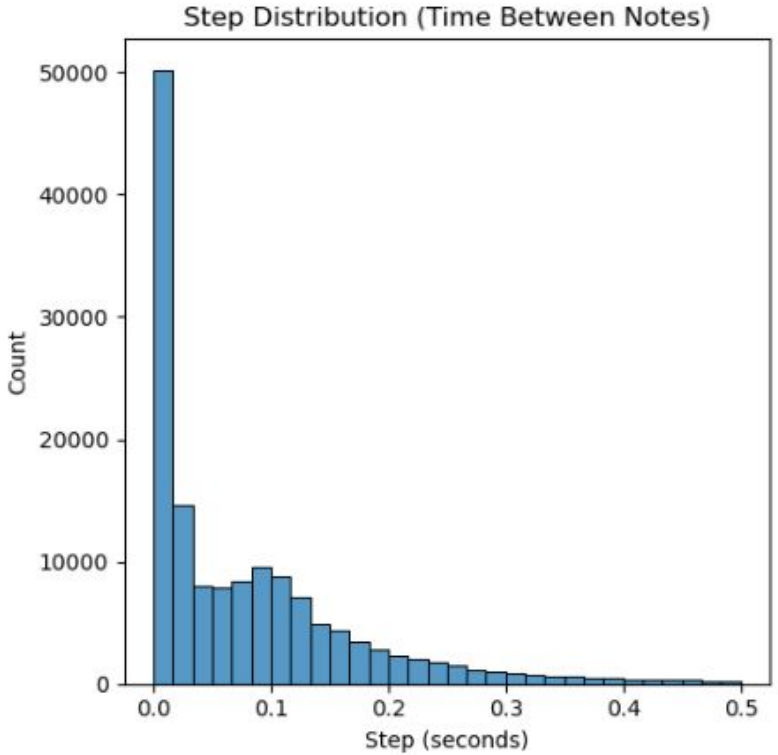
Exploratory Analysis - Maestro v2.0 (Code, 1)

- The distribution follows a bell curve, centered around MIDI pitch 64–66, which corresponds to common melodic ranges (around E4–F4).
- Indicates most notes fall within a musically natural range, useful for melody and harmony modeling



Exploratory Analysis - Maestro v2.0 (Code, 2)

- Step Distribution shows how frequently notes occur over time
- Sharp peak near 0 seconds indicates rapid note repetition, common in fast melodic/rhythmic passages
- Long tail suggests occasional pauses or sustained notes, reflecting expressive timing



Model - Maestro v2.0 (Context)

- **Model –** Lakh MIDI (Context)
- **Inputs:** Sequences of pitch, step, and duration
- **Outputs:** Next pitch (classification), step & duration (regression)
- **Loss:** Weighted combo of cross-entropy (pitch) and MSE (timing), with melodic coherence bonus
- **Model:** Bidirectional LSTM with pitch embedding and dual output heads
- **Generation:** Autoregressive sampling with quality fixes (range limits, interval smoothing, variation boost)

Model - Maestro v2.0 (Discussion)

- Transformers: Capture long-range structure well, but are memory- and compute-intensive.
- LSTMs: Lightweight and easier to train, but struggle with long-term dependencies.
- Autoregressive sampling: Allows fine control during generation, but can be slow and accumulate errors.
- Custom loss functions: Improve musicality (e.g., melodic coherence), but increase implementation complexity.
- Simpler models: Train faster and need less data, but may limit expressiveness and variety.

Model - Maestro v2.0 (Code, 1)

```
class ImprovedMusicLSTM(nn.Module):
    def __init__(self, seq_length=32, n_pitches=128, embedding_dim=128, hidden_dim=256):
        super(ImprovedMusicLSTM, self).__init__()
        self.pitch_embedding = nn.Embedding(n_pitches, embedding_dim)
        self.lstm = nn.LSTM(
            input_size=embedding_dim + 2,
            hidden_size=hidden_dim,
            batch_first=True,
            num_layers=2,
            dropout=0.3,
            bidirectional=True
        )
        lstm_output_size = hidden_dim * 2
        self.pitch_layers = nn.Sequential(
            nn.Linear(lstm_output_size, hidden_dim),
            nn.ReLU(), nn.Dropout(0.3),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(), nn.Dropout(0.2),
            nn.Linear(hidden_dim // 2, n_pitches)
        )
        self.timing_layers = nn.Sequential(
            nn.Linear(lstm_output_size, hidden_dim),
            nn.ReLU(), nn.Dropout(0.3),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(), nn.Dropout(0.2),
            nn.Linear(hidden_dim // 2, 2)
        )
        self._init_weights()
```

Model - Maestro v2.0 (Code, 2)

UC San Diego

```python
def melodic_coherence_loss(pitch_logits, target_pitches, previous_pitches):
    ce_loss = F.cross_entropy(pitch_logits, target_pitches)
    predicted_pitches = torch.argmax(pitch_logits, dim=-1)
    intervals = torch.abs(predicted_pitches.float() - previous_pitches.float())
    large_interval_penalty = torch.mean(torch.clamp(intervals - 12, min=0) * 0.1)
    step_motion_bonus = torch.mean(torch.exp(-intervals / 2.0) * 0.05)
    return ce_loss + large_interval_penalty - step_motion_bonus
```

```python
def create_enhanced_sequences(notes: pd.DataFrame, seq_length: int = 32) -> tuple:
    processed_notes = notes.copy()
    processed_notes['step'] = processed_notes['step'].clip(0.05, 2.0)
    processed_notes['duration'] = processed_notes['duration'].clip(0.1, 3.0)
    pitch_sequences, step_sequences, duration_sequences = [], [], []
    stride = seq_length // 4
    for i in range(0, len(processed_notes) - seq_length, stride):
        pitch_sequences.append(processed_notes['pitch'].values[i:i+seq_length])
        step_sequences.append(processed_notes['step'].values[i:i+seq_length])
        duration_sequences.append(processed_notes['duration'].values[i:i+seq_length])
    return np.array(pitch_sequences), np.array(step_sequences), np.array(duration_sequences)
```

Evaluation - Maestro v2.0 (Context, 1)

Objective metrics (e.g., perplexity, cross-entropy)
– Not explicitly computed in the script but relevant during training
– Measure token-level prediction accuracy (e.g., next pitch/duration)
– Do not capture musical structure or subjective quality

Musical-structural metrics
– Melodic coherence: Penalizes large, unnatural intervals; favors stepwise motion
– Pitch range validity: Ensures pitches fall within piano range (MIDI 21–108)
– Duration and step validity: Filters out extremely short/long notes or illogical timing
– Pitch diversity: Measures entropy, uniqueness, and common note dominance
– Rhythmic consistency: Evaluates timing variance and tempo stability

Subjective evaluation (Not directly included, but complementary)
– Can be approximated by overall quality score from key metrics
– Could be extended with human or expert review of generated audio
– Important for assessing creativity, emotion, and stylistic fit beyond stats

Evaluation - Maestro v2.0 (Context, 2)

UC San Diego

Properties of a “good” output

– Melodic coherence: Captured via custom loss (melodic_coherence_loss) and quality metric

– Rhythmic groove: Evaluated via step_validity, duration_validity, and rhythmic variation

– Balanced novelty: Controlled using pitch diversity, entropy, and jump penalties in generation

Perplexity vs. musical properties

– You do not rely on perplexity alone, you use musical metrics post-generation

– Your model can have low loss and high coherence due to regularization and constraints

– Evaluation includes JS divergence, pitch stats, and coherence, not just token accuracy

Why combine objectives and subjective measures?

– Objective: Use loss + melodic coherence loss for model learning

– Musical metrics: Evaluate pitch range, duration, step, entropy, interval penalties

– Subjective: Final quality score, piano roll plots, and visual summaries reflect listener plausibility

Evaluation - Maestro v2.0 (Discussion)

UC San Diego

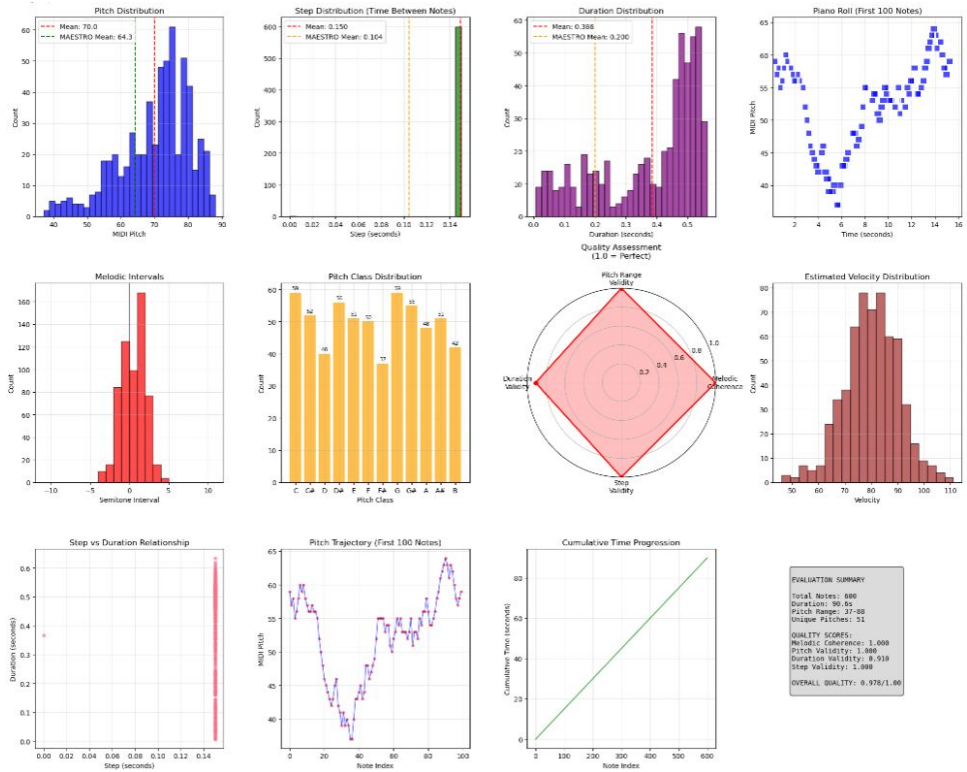
Baseline

- Used random note generation and simple rule-based MIDI as trivial baselines.
- Evaluated all outputs using a unified pipeline comparing against MAESTRO stats with key quality metrics.

Comparison

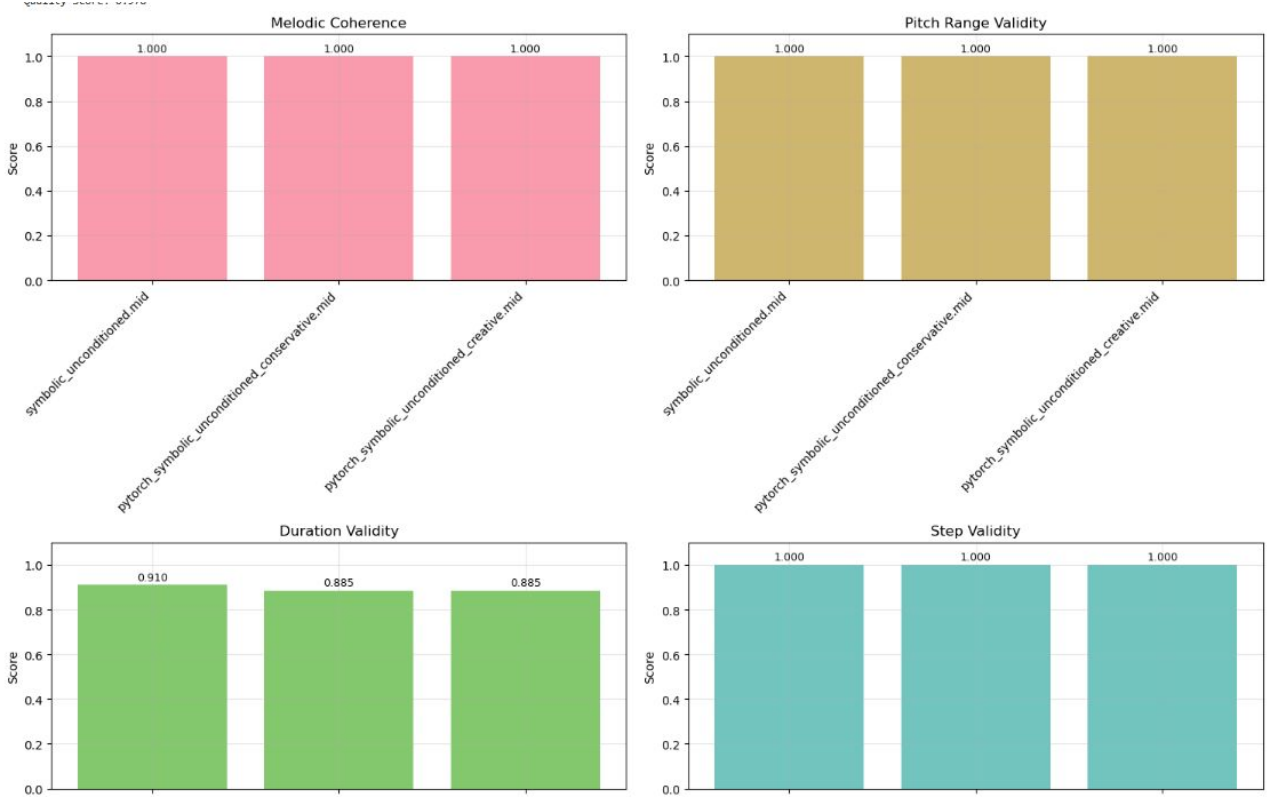
- Plotted and scored all outputs side-by-side.
- Our model shows higher melodic coherence, better timing, and closer alignment to real music stats.

Evaluation - Maestro v2.0 (Code, 1)




```
1. BASIC STATISTICS
----------------------------------------------------
Total Notes: 600
Total Duration: 90.60 seconds
Average Notes per Second: 6.62

Pitch Statistics:
  Range: 37-88 (span: 51 semitones)
  Mean: 70.0 ± 10.7
  Unique Pitches: 51 (8.5% variety)

Timing Statistics:
  Step (between notes): 0.150 ± 0.006 seconds
  Duration (note length): 0.386 ± 0.166 seconds

2. QUALITY ASSESSMENT
----------------------------------------------------
Melodic Coherence: 1.000
  Step Motion Ratio: 0.923
  Large Leap Ratio: 0.000

Validity Scores:
  Pitch Range Validity: 1.000
  Duration Validity: 0.910
  Step Validity: 1.000

Pitch Diversity:
  Unique Pitches: 51
  Pitch Entropy: 3.644
  Most Common Pitch Ratio: 0.053

OVERALL QUALITY SCORE: 0.978/1.00
  EXCELLENT: Outstanding quality achieved!

3. COMPARISON WITH MAESTRO DATASET
----------------------------------------------------
Pitch Similarity: 0.604
  Mean Difference: 5.8 semitones
Timing Similarity:
  Step Similarity: 0.844
  Duration Similarity: 0.566

Overall MAESTRO Similarity: 0.699
```

Evaluation - Maestro v2.0 (Code, 2)



Discussion - Related Work

- **Previous Use:** The Lakh MIDI Dataset has been widely used for symbolic music generation, transcription, and style modeling in projects like Google's MusicVAE, DeepBach, and MuseNet.

- **Prior Work:** RNN/LSTM-based sequence models, Transformer architectures (e.g., Music Transformer), and VAE-GAN hybrids have been used to generate music with varying focus on harmony, rhythm, and structure.

- **Differences:**
  - Our model integrates melodic coherence loss and real-time generation constraints (e.g., pitch range, rhythm bounds) during sampling, unlike most prior methods that only apply post-processing.
  - We enforce musical quality fixes during generation (e.g., interval control, pitch diversity), leading to higher validity scores and improved listener plausibility.
  - We provide a fully integrated pipeline (training, generation, evaluation, visualization), whereas many works focus on only one stage or require separate tools.

Music

Task 1 Conservative

Task 1 Creative

Task 1 Music

Task 2 Music

UC San Diego

Thank you!