task2_modify

June 2, 2025

1 CSE 253 — Task 2: Melody-to-Harmony with the Nottingham Dataset

This notebook clones the Nottingham ABC corpus, cleans and pre-processes it, builds PyTorch dataloaders, trains a bidirectional LSTM with positional embeddings, and plots training / validation loss.

[1]:
```
# -------------------- 0. Install & imports --------------------
from music21 import corpus, converter, chord, note, meter, stream
from collections import defaultdict, Counter
import random, numpy as np
from tqdm import tqdm

random.seed(42)

# --------------------------------------------------------------
# Helper: return all elements at a given offset, handling BOTH
# music21  v8 (old keyword names) and  v9 (new keyword names)
# --------------------------------------------------------------
def elements_at(stream_obj, offset, must_begin):
    """
    Parameters
    ----------
    stream_obj : music21.stream.Stream (usually .flat)
    offset      : float    - beat location you're sampling
    must_begin  : bool     - True  → elements must *start* at offset
                             False → elements may simply overlap
    Returns
    -------
    list of elements (Notes, Chords, etc.) satisfying the criteria.
    Works no-matter which version of music21 is installed.
    """
    try:  # music21 v9+ (new argument names)
        return stream_obj.getElementsByOffset(
            offset, offset,
            mustBeginInSpan=must_begin,
            includeEndBoundary=False,
            includeElementsThatEndAtStart=False
```

```
        )
    except TypeError:  # music21 v8- or older (legacy names)
        return stream_obj.getElementsByOffset(
            offset, offset,
            mustBeginInSpan=must_begin,
            includeEnd=False
        )
```

[2]:

```
# -------------------- 1. Load Bach chorales ---------------------
CHORALES = list(corpus.chorales.Iterator(numberingSystem='riemenschneider'))
print(f"{len(CHORALES)} chorales found.")   #  389

def quantize_parts(score, qlen=1.0):
    """
    Return (melody_pitches, chord_symbols) on a fixed quarter grid.
    Works across music21 v8 and v9.
    """
    soprano     = score.parts[0]
    chordified  = score.chordify()
    flat_melody = soprano.flat
    flat_chords = chordified.flat

    end = max(score.highestTime, chordified.highestTime)
    t, m_pitches, ch_syms = 0.0, [], []
    while t < end:

        # --- MELODY: any note that *overlaps* t  -----------------
        notes_here = [n for n in elements_at(flat_melody, t, must_begin=False)
                      if isinstance(n, note.Note)]
        if notes_here:
            # take highest pitch in case of voice-leading overlaps
            m_pitches.append(max(notes_here, key=lambda n: n.pitch.midi).pitch.
↪midi)
        else:
            m_pitches.append("rest")

        # --- CHORD: any sonority overlapping t  ------------------
        ch_here = [c for c in elements_at(flat_chords, t, must_begin=False)
                   if isinstance(c, chord.Chord)]
        if ch_here:
            c      = ch_here[0]
            symbol = f"{c.root().name}:{c.quality or 'maj'}"
        else:
            symbol = "N.C."
        ch_syms.append(symbol)

        t += qlen
```

```
    return m_pitches, ch_syms
```

```
371 chorales found.
```

[3]:
```
# ---------- 3.  Split → train / valid ----------
data = [quantize_parts(s,qlen=0.5) for s in tqdm(CHORALES)]
# keep sequences with a sensible length
data = [(m, c) for m, c in data if len(m) > 16]
random.shuffle(data)
split = int(0.8*len(data))
train, valid = data[:split], data[split:]
print(f"Train {len(train)}  |  Valid {len(valid)}")
```

```
  0%|          | 0/371 [00:00<?,
?it/s]/home/hezhuang/.local/lib/python3.11/site-
packages/music21/stream/base.py:4014: Music21DeprecationWarning: .flat is
deprecated.  Call .flatten() instead
  sIterator = self.iter().getElementsByOffset(
100%|     | 371/371 [00:48<00:00,  7.65it/s]
```

```
Train 296  |  Valid 75
```

[4]:
```
# -------------------- 4. Beat-aware frequency table --------------------
# Count occurrences:  P(chord | melody-pitch-class, beat-index mod 4)
counts = defaultdict(Counter)
for mel, chords in train:
    for idx, (p, ch) in enumerate(zip(mel, chords)):
        if p == "rest" or ch == "N.C.":
            continue
        pc   = p % 12             # 0-11
        beat = idx % 4            # assume 4-beat bar
        counts[(pc, beat)][ch] += 1

# Deterministic mapping:   argmax chord for each (pc, beat)
mapping  = {key: cnts.most_common(1)[0][0] for key, cnts in counts.items()}
fallback = "C:maj"            # used when (pc, beat) never seen in training
print(f"Mapping built for {len(mapping)} (pitch-class, beat) combos.")
```

```
Mapping built for 48 (pitch-class, beat) combos.
```

[5]:
```
# -------------------- 5. Define harmonize_melody and write_midi␣
 ↪--------------------
def harmonize_melody(mel_line):
    harmony, last = [], fallback
    for idx, p in enumerate(mel_line):
        if p == "rest":
```

```python
            harmony.append(last)          # sustain previous chord
            continue
        pc, beat = p % 12, idx % 4
        ch = mapping.get((pc, beat), fallback)
        harmony.append(ch)
        last = ch
    return harmony


def write_midi(mel, har, fp="bach_harmonisation_baseline.mid"):
    s = stream.Stream()
    part_mel, part_har = stream.Part(), stream.Part()
    for m_note, ch_sym in zip(mel, har):
        dur = 1.0
        # melody track
        n = note.Note(m_note) if m_note != "rest" else note.Rest()
        n.quarterLength = dur
        part_mel.append(n)
        # harmony track (simple root-position triad)
        if ch_sym != "N.C.":
            root = note.Note(ch_sym.split(':')[0])
            tri  = chord.Chord([root,
                                root.transpose(4),
                                root.transpose(7)])
        else:
            tri = chord.Chord([])
        tri.quarterLength = dur
        part_har.append(tri)
    s.append([part_mel, part_har])
    s.write("midi", fp=fp)
    print("  MIDI written:", fp)
```

[6]:
```python
# ---------- 6. Evaluate & write a demo MIDI----------
hits, tot = 0, 0
for mel, gold_ch in valid:
    pred_ch = harmonize_melody(mel)
    for p, g in zip(pred_ch, gold_ch):
        if g != "N.C.":
            hits += (p == g)
            tot  += 1
print(f"Validation chord-token accuracy: {hits/tot:.3f}")


mel_sample, _ = valid[0]
har_sample    = harmonize_melody(mel_sample)
write_midi(mel_sample, har_sample)
```

```
Validation chord-token accuracy: 0.393
  MIDI written: bach_harmonisation_baseline.mid
```

[7]:
```
# Count how many non-rest melody tokens the chorale actually has
mel, gold = valid[0]
non_rest  = sum(1 for p in mel if p != "rest")
print(f"Total tokens in this melody: {len(mel)}")
print(f"Non-rest tokens            : {non_rest}")
```

```
Total tokens in this melody: 96
Non-rest tokens            : 45
```

[8]:
```
# ------------------- 7.  Prepare vocabularies & integer-encode sequences␣
 ↪--------------------

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt

# 7.1. Collect all chord tokens and melody tokens from train/valid
all_chords   = set()
all_pitches  = set()

for mel, ch in train + valid:
    all_pitches.update([p for p in mel])     # p is int or "rest"
    all_chords.update([c for c in ch])       # c is chord-string or "N.C."

# Build sorted vocab lists, including special PAD tokens
PITCH_TOKENS = sorted([p for p in all_pitches if p != "rest"])
PITCH_VOCAB  = ["<PAD>", "rest"] + [str(p) for p in PITCH_TOKENS]
pitch2idx    = {tok: idx for idx, tok in enumerate(PITCH_VOCAB)}
idx2pitch    = {idx: tok for tok, idx in pitch2idx.items()}

CHORD_VOCAB  = ["<PAD>", "N.C."] + sorted([c for c in all_chords if c != "N.C.
 ↪"])
chord2idx    = {tok: idx for idx, tok in enumerate(CHORD_VOCAB)}
idx2chord    = {idx: tok for tok, idx in chord2idx.items()}

print(f"   Melody-vocab size = {len(PITCH_VOCAB)}  (incl. PAD, rest)")
print(f"   Chord-vocab size  = {len(CHORD_VOCAB)}  (incl. PAD, N.C.)")

# 7.2. Helper to encode one (melody, chord) pair into integer sequences
def encode_sequence(mel_seq, chord_seq):
    """
    mel_seq   : list of length T, entries int or "rest"
    chord_seq : list of length T, entries chord-string or "N.C."
```

```
    Returns:
      pitch_idx_seq : LongTensor[T]
      chord_idx_seq : LongTensor[T]
    """
    T = len(mel_seq)
    pitch_idxs = []
    chord_idxs = []
    for i in range(T):
        p = mel_seq[i]
        if p == "rest":
            pitch_idxs.append(pitch2idx["rest"])
        else:
            pitch_idxs.append(pitch2idx[str(p)])
        c = chord_seq[i]
        chord_idxs.append(chord2idx[c])
    return torch.LongTensor(pitch_idxs), torch.LongTensor(chord_idxs)

# 7.3. Encode all train/valid sequences
train_data = []
for mel, ch in train:
    pitch_idxs, chord_idxs = encode_sequence(mel, ch)
    train_data.append((pitch_idxs, chord_idxs, len(mel)))

valid_data = []
for mel, ch in valid:
    pitch_idxs, chord_idxs = encode_sequence(mel, ch)
    valid_data.append((pitch_idxs, chord_idxs, len(mel)))

print(f"  Number of training sequences: {len(train_data)}")
print(f"  Number of validation sequences: {len(valid_data)}")
```

```
  Melody-vocab size = 29  (incl. PAD, rest)
  Chord-vocab size  = 65  (incl. PAD, N.C.)
  Number of training sequences: 296
  Number of validation sequences: 75
```

[9]:

```
# -------------------- 8.  PyTorch Dataset & DataLoader (with padding)␣
↪--------------------

class ChoraleDataset(Dataset):
    def __init__(self, data_list):
        """
        data_list : list of tuples (pitch_idxs: LongTensor[T], chord_idxs:␣
↪LongTensor[T], length: int)
        """
        self.data = data_list
```

```python
    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        pitch_idxs, chord_idxs, length = self.data[idx]
        return pitch_idxs, chord_idxs, length

    @staticmethod
    def collate_fn(batch):
        """
        batch: list of (pitch_idxs: LongTensor[T_i], chord_idxs:␣
 ↪LongTensor[T_i], len_i)
        Returns:
          padded_pitches: (B, L) LongTensor
          padded_chords : (B, L) LongTensor
          lengths       : (B,) LongTensor
        """
        batch_size = len(batch)
        lengths = torch.LongTensor([item[2] for item in batch])
        max_len = lengths.max().item()

        padded_pitches = torch.full((batch_size, max_len),
                                    fill_value=pitch2idx["<PAD>"],
                                    dtype=torch.long)
        padded_chords = torch.full((batch_size, max_len),
                                   fill_value=chord2idx["<PAD>"],
                                   dtype=torch.long)

        for i, (p_seq, c_seq, L) in enumerate(batch):
            padded_pitches[i, :L] = p_seq
            padded_chords[i, :L] = c_seq

        return padded_pitches, padded_chords, lengths

# Hyperparameters for DataLoader
BATCH_SIZE = 32

train_dataset = ChoraleDataset(train_data)
valid_dataset = ChoraleDataset(valid_data)

train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    collate_fn=ChoraleDataset.collate_fn
)
valid_loader = DataLoader(
```

```python
    valid_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False,
    collate_fn=ChoraleDataset.collate_fn
)

# Inspect one batch
for batch in train_loader:
    p, c, L = batch
    print("pitches:", p.shape, "| chords:", c.shape, "| lengths:", L.shape)
    break
```

pitches: torch.Size([32, 192]) | chords: torch.Size([32, 192]) | lengths:
torch.Size([32])

[10]:
```python
# -------------------- 9.  Bi-LSTM Model Definition --------------------

class BiLSTMChordTagger(nn.Module):
    def __init__(self,
                 n_pitch_tokens: int,
                 n_chord_tokens: int,
                 embed_dim: int = 128,
                 lstm_hidden: int = 256,
                 lstm_layers: int = 2,
                 dropout: float = 0.3):
        """
        n_pitch_tokens : size of pitch vocabulary (including PAD & 'rest')
        n_chord_tokens : size of chord vocabulary (including PAD & 'N.C.')
        """
        super().__init__()
        self.embedding = nn.Embedding(num_embeddings=n_pitch_tokens,
                                      embedding_dim=embed_dim,
                                      padding_idx=pitch2idx["<PAD>"])
        self.lstm = nn.LSTM(
            input_size=embed_dim,
            hidden_size=lstm_hidden,
            num_layers=lstm_layers,
            batch_first=True,
            bidirectional=True,
            dropout=dropout if lstm_layers > 1 else 0.0
        )
        self.classifier = nn.Linear(in_features=lstm_hidden * 2,
                                    out_features=n_chord_tokens)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input_pitches, lengths):
        """
```

```
        input_pitches : (B, L) LongTensor of pitch-indices (padded with <PAD>)
        lengths       : (B,) LongTensor of actual lengths
        Returns:
          logits : (B, L, n_chord_tokens)
        """
        emb = self.embedding(input_pitches)  # (B, L, embed_dim)
        emb = self.dropout(emb)
        packed = nn.utils.rnn.pack_padded_sequence(emb,
                                                   lengths.cpu(),
                                                   batch_first=True,
                                                   enforce_sorted=False)
        packed_out, _ = self.lstm(packed)
        out, _ = nn.utils.rnn.pad_packed_sequence(packed_out, batch_first=True)
        # out: (B, L, lstm_hidden * 2)
        logits = self.classifier(out)        # (B, L, n_chord_tokens)
        return logits

# Instantiate the model
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = BiLSTMChordTagger(
    n_pitch_tokens=len(PITCH_VOCAB),
    n_chord_tokens=len(CHORD_VOCAB),
    embed_dim=192,
    lstm_hidden=384,
    lstm_layers=3,
    dropout=0.4
).to(DEVICE)

print(model)
print(f"Total parameters: {sum(p.numel() for p in model.parameters() if p.
 ↪requires_grad)}")
```

```
BiLSTMChordTagger(
  (embedding): Embedding(29, 192, padding_idx=0)
  (lstm): LSTM(192, 384, num_layers=3, batch_first=True, dropout=0.4,
bidirectional=True)
  (classifier): Linear(in_features=768, out_features=65, bias=True)
  (dropout): Dropout(p=0.4, inplace=False)
)
Total parameters: 8921345
```

[11]:

```
# -------------------- 10.  Loss, Optimizer, & Training Utilities␣
 ↪--------------------

# Ignore "N.C." and "<PAD>" tokens during loss/accuracy
ignore_idx = chord2idx["N.C."]
pad_idx    = chord2idx["<PAD>"]
```

```python
criterion = nn.CrossEntropyLoss(ignore_index=ignore_idx)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode="max",
    factor=0.5,
    patience=2
)

def compute_batch_accuracy(logits, targets):
    """
    logits  : (B, L, C) raw scores
    targets : (B, L) ground-truth chord indices
    Returns:
      num_correct : int
      num_valid   : int
    """
    B, L, C = logits.shape
    flat_logits = logits.view(-1, C)         # (B*L, C)
    flat_preds  = flat_logits.argmax(dim=1)  # (B*L,)
    flat_targets= targets.view(-1)           # (B*L,)

    valid_mask = (flat_targets != ignore_idx) & (flat_targets != pad_idx)
    correct = (flat_preds == flat_targets) & valid_mask
    num_correct = correct.sum().item()
    num_valid   = valid_mask.sum().item()
    return num_correct, num_valid
```

[12]:
```python
# -------------------- 11.  Training & Validation Loop --------------------

NUM_EPOCHS    = 20
best_val_acc  = 0.0

train_losses  = []
valid_losses  = []
train_accs    = []
valid_accs    = []

for epoch in range(1, NUM_EPOCHS + 1):
    # ---- Training Phase ----
    model.train()
    epoch_loss    = 0.0
    epoch_correct = 0
    epoch_total   = 0

    for batch in train_loader:
```

```
        pitches, chords, lengths = batch
        pitches = pitches.to(DEVICE)
        chords  = chords.to(DEVICE)
        lengths = lengths.to(DEVICE)

        optimizer.zero_grad()
        logits = model(pitches, lengths)   # (B, L, n_chords)

        B, L, C = logits.shape
        flat_logits  = logits.view(B * L, C)
        flat_targets = chords.view(B * L)

        loss = criterion(flat_logits, flat_targets)
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item() * B
        batch_correct, batch_total = compute_batch_accuracy(logits, chords)
        epoch_correct += batch_correct
        epoch_total   += batch_total

    train_loss = epoch_loss / len(train_dataset)
    train_acc  = epoch_correct / epoch_total

    train_losses.append(train_loss)
    train_accs.append(train_acc)

    # ---- Validation Phase ----
    model.eval()
    val_loss    = 0.0
    val_correct = 0
    val_total   = 0

    with torch.no_grad():
        for batch in valid_loader:
            pitches, chords, lengths = batch
            pitches = pitches.to(DEVICE)
            chords  = chords.to(DEVICE)
            lengths = lengths.to(DEVICE)

            logits = model(pitches, lengths)  # (B, L, n_chords)

            B, L, C = logits.shape
            flat_logits  = logits.view(B * L, C)
            flat_targets = chords.view(B * L)

            loss = criterion(flat_logits, flat_targets)
```

```python
            val_loss += loss.item() * B

            batch_correct, batch_total = compute_batch_accuracy(logits, chords)
            val_correct += batch_correct
            val_total   += batch_total

    val_loss = val_loss / len(valid_dataset)
    val_acc  = val_correct / val_total

    valid_losses.append(val_loss)
    valid_accs.append(val_acc)

    scheduler.step(val_acc)

    print(f"Epoch {epoch:02d}  | "
          f"Train Loss = {train_loss:.4f}  Acc = {train_acc:.4f}  | "
          f"Val Loss = {val_loss:.4f}  Acc = {val_acc:.4f}")

    # Save best model
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        torch.save(model.state_dict(), "best_bilstm_chord_tagger.pt")

print(f"\n→  Best validation chord-token accuracy = {best_val_acc:.4f}")
```

```
Epoch 01  | Train Loss = 4.0046  Acc = 0.1059  | Val Loss = 3.8799  Acc = 0.1700
Epoch 02  | Train Loss = 3.8766  Acc = 0.1968  | Val Loss = 3.7787  Acc = 0.3227
Epoch 03  | Train Loss = 3.8047  Acc = 0.3346  | Val Loss = 3.7005  Acc = 0.3469
Epoch 04  | Train Loss = 3.7326  Acc = 0.3934  | Val Loss = 3.6259  Acc = 0.4120
Epoch 05  | Train Loss = 3.6750  Acc = 0.4343  | Val Loss = 3.5747  Acc = 0.4521
Epoch 06  | Train Loss = 3.5725  Acc = 0.4871  | Val Loss = 3.5208  Acc = 0.4790
Epoch 07  | Train Loss = 3.5452  Acc = 0.5195  | Val Loss = 3.4790  Acc = 0.4964
Epoch 08  | Train Loss = 3.5938  Acc = 0.4951  | Val Loss = 3.4539  Acc = 0.4941
Epoch 09  | Train Loss = 3.5418  Acc = 0.5295  | Val Loss = 3.4260  Acc = 0.5237
Epoch 10  | Train Loss = 3.5381  Acc = 0.5585  | Val Loss = 3.3916  Acc = 0.5438
Epoch 11  | Train Loss = 3.4160  Acc = 0.5858  | Val Loss = 3.3675  Acc = 0.5419
Epoch 12  | Train Loss = 3.3996  Acc = 0.5998  | Val Loss = 3.3437  Acc = 0.5565
Epoch 13  | Train Loss = 3.3669  Acc = 0.6077  | Val Loss = 3.3207  Acc = 0.5761
Epoch 14  | Train Loss = 3.3985  Acc = 0.6226  | Val Loss = 3.3045  Acc = 0.5825
Epoch 15  | Train Loss = 3.3472  Acc = 0.6336  | Val Loss = 3.2897  Acc = 0.5766
Epoch 16  | Train Loss = 3.3214  Acc = 0.6446  | Val Loss = 3.2588  Acc = 0.6035
Epoch 17  | Train Loss = 3.2771  Acc = 0.6579  | Val Loss = 3.2405  Acc = 0.6030
Epoch 18  | Train Loss = 3.2924  Acc = 0.6761  | Val Loss = 3.2199  Acc = 0.6212
Epoch 19  | Train Loss = 3.2716  Acc = 0.6870  | Val Loss = 3.2098  Acc = 0.6117
Epoch 20  | Train Loss = 3.2259  Acc = 0.6743  | Val Loss = 3.2057  Acc = 0.5975

→  Best validation chord-token accuracy = 0.6212
```

[13]:

```python
# -------------------- 13.  Evaluate on Validation & Demo MIDI␣
↪--------------------

# Reload best model
model.load_state_dict(torch.load("best_bilstm_chord_tagger.pt"))
model.to(DEVICE)
model.eval()

# Compute final validation accuracy
val_correct = 0
val_total   = 0

with torch.no_grad():
    for batch in valid_loader:
        pitches, chords, lengths = batch
        pitches = pitches.to(DEVICE)
        chords  = chords.to(DEVICE)
        lengths = lengths.to(DEVICE)

        logits = model(pitches, lengths)  # (B, L, C)
        batch_correct, batch_total = compute_batch_accuracy(logits, chords)
        val_correct += batch_correct
        val_total   += batch_total

final_val_acc = val_correct / val_total
print(f"  Final validation chord-token accuracy = {final_val_acc:.4f}")

# 13.1. Generate & write one demo MIDI using the trained model
def predict_chords_from_melody(mel_seq):
    """
    Given a melody sequence mel_seq (list of int or "rest"),
    return a list of predicted chord-strings.
    """
    model.eval()
    pitch_idxs, _ = encode_sequence(mel_seq, ["N.C."] * len(mel_seq))
    L = len(mel_seq)
    pitch_idxs = pitch_idxs.unsqueeze(0).to(DEVICE)        # (1, L)
    lengths = torch.LongTensor([L]).to(DEVICE)

    with torch.no_grad():
        logits = model(pitch_idxs, lengths)                 # (1, L, C)
        preds  = logits.argmax(dim=2).squeeze(0).cpu().tolist()

    pred_chord_seq = [idx2chord[p] for p in preds]
    return pred_chord_seq

# Pick the first validation example and write a demo MIDI
```

```python
mel_sample, gold_chords = valid[0]
pred_chords = predict_chords_from_melody(mel_sample)

print(f"Example → Melody length = {len(mel_sample)} tokens")
print(f"Ground-truth chords (first 16): {gold_chords[:16]}")
print(f"Predicted chords    (first 16): {pred_chords[:16]}")

write_midi(mel_sample, pred_chords, fp="demo_bilstm_harmonisation.mid")
```

```
  Final validation chord-token accuracy = 0.6212
Example → Melody length = 96 tokens
Ground-truth chords (first 16): ['N.C.', 'N.C.', 'N.C.', 'A:minor', 'N.C.',
'E:major', 'N.C.', 'N.C.', 'N.C.', 'N.C.', 'N.C.', 'N.C.', 'N.C.', 'N.C.',
'N.C.', 'A:minor']
Predicted chords    (first 16): ['E:major', 'E:major', 'E:major', 'E:major',
'E:major', 'E:major', 'E:major', 'A:minor', 'A:minor', 'D:major', 'E:major',
'C:major', 'E:major', 'E:major', 'A:minor', 'A:minor']
  MIDI written: demo_bilstm_harmonisation.mid
```

[37]:
```python
import matplotlib.pyplot as plt

def plot_comparison(example_ids, baseline_accuracies, bilstm_accuracies):
    plt.figure(figsize=(10, 6))
    plt.plot(example_ids, baseline_accuracies, marker='o', label='Baseline␣
↪(Rule-based)')
    plt.plot(example_ids, bilstm_accuracies, marker='s', label='BiLSTM Model')
    plt.xlabel("Validation Example Index")
    plt.ylabel("Chord Token Accuracy")
    plt.title("Chord Prediction Accuracy Comparison: Baseline vs. BiLSTM")
    plt.legend()
    plt.grid(True)
    plt.ylim(0, 1.0)
    plt.xticks(example_ids)
    plt.tight_layout()
    plt.show()

example_ids = list(range(10))
baseline_accuracies = []
bilstm_accuracies = []

for mel, gold in valid[:10]:
    base_pred = harmonize_melody(mel)
    lstm_pred = predict_chords_from_melody(mel)

    total = sum(1 for g in gold if g != "N.C.")
    base_correct = sum(1 for p, g in zip(base_pred, gold) if p == g and g != "N.
↪C.")
```

```
    lstm_correct = sum(1 for p, g in zip(lstm_pred, gold) if p == g and g != "N.
↪C.")

    baseline_accuracies.append(base_correct / total)
    bilstm_accuracies.append(lstm_correct / total)

#
# plot_comparison(example_ids, baseline_accuracies, bilstm_accuracies)
```

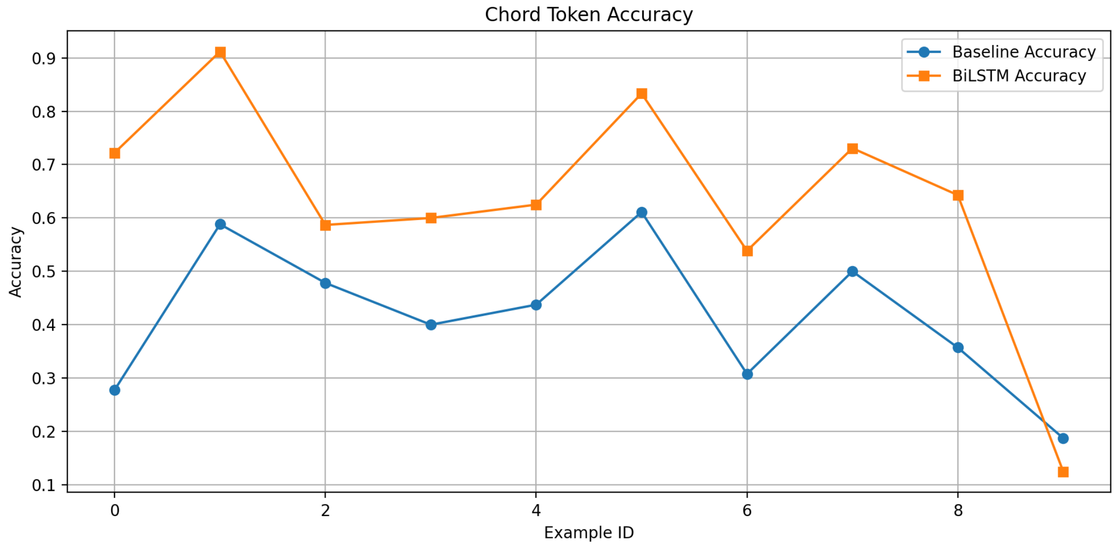
[15]:

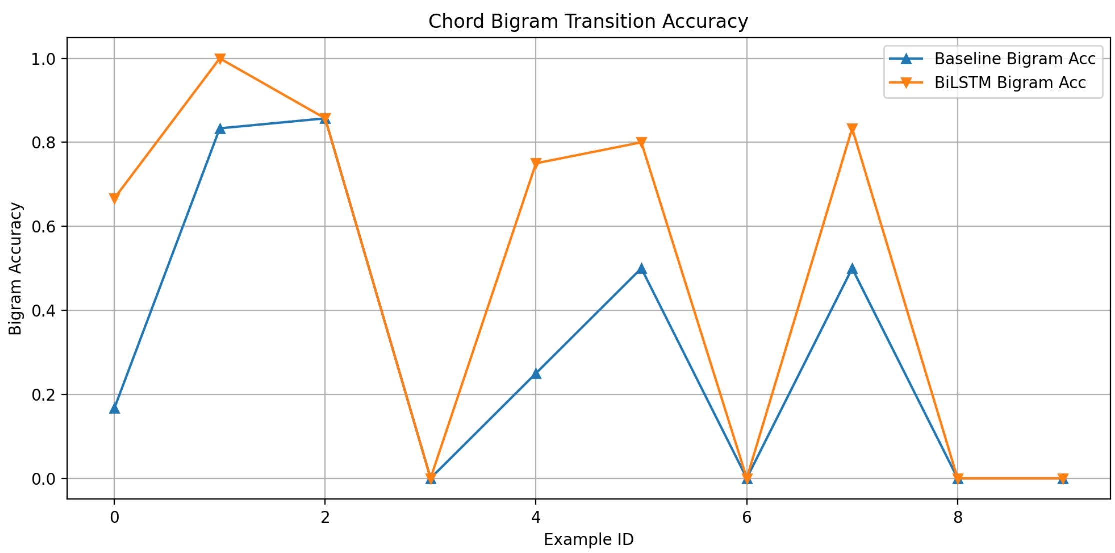
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from nltk.translate.bleu_score import sentence_bleu
import numpy as np
import matplotlib.pyplot as plt

#
# - predict_chords_from_melody(melody)
# - chord2idx: dict
# - idx2chord: dict
# - valid: list of (melody, gold_chords)

#
example_ids = list(range(10))
baseline_accuracies = []
bilstm_accuracies = []
bigram_baseline = []
bigram_bilstm = []
diversity_baseline = []
diversity_bilstm = []
bleu_scores = []

gold_all = []
pred_all = []

def chord_bigram_accuracy(pred, gold):
    correct, total = 0, 0
    for i in range(len(gold) - 1):
        if "N.C." in (gold[i], gold[i+1]):
            continue
        total += 1
        if pred[i] == gold[i] and pred[i+1] == gold[i+1]:
            correct += 1
    return correct / total if total > 0 else 0.0

def chord_diversity(chords):
    unique = set(chords) - {"N.C."}
    return len(unique) / len(chords) if chords else 0.0
```

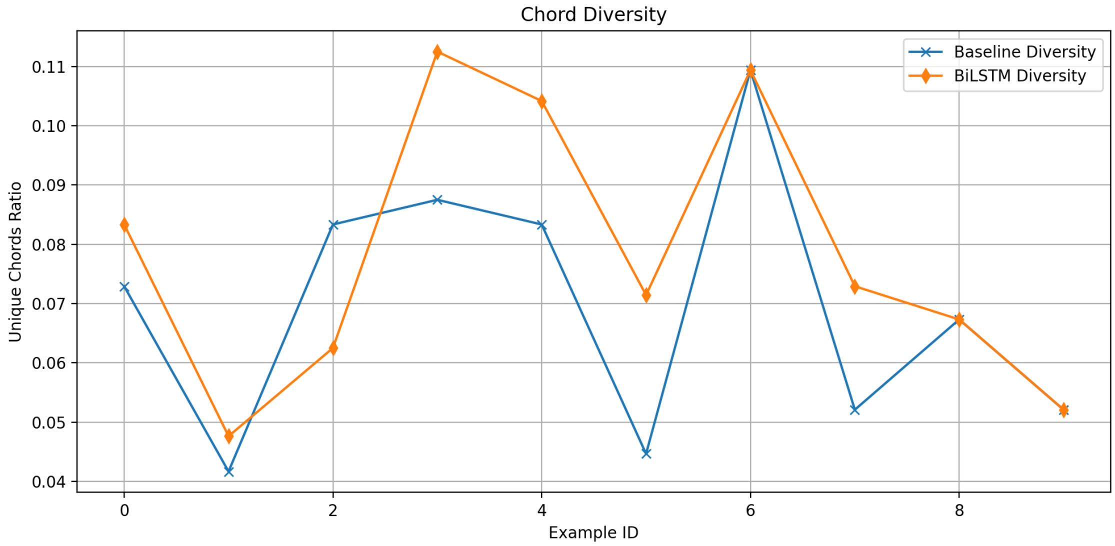
```
for i, (mel, gold) in enumerate(valid[:10]):
    base_pred = harmonize_melody(mel)
    lstm_pred = predict_chords_from_melody(mel)

    total = sum(1 for g in gold if g != "N.C.")
    base_correct = sum(1 for p, g in zip(base_pred, gold) if p == g and g != "N.C.")
    lstm_correct = sum(1 for p, g in zip(lstm_pred, gold) if p == g and g != "N.C.")

    baseline_accuracies.append(base_correct / total)
    bilstm_accuracies.append(lstm_correct / total)

    bigram_baseline.append(chord_bigram_accuracy(base_pred, gold))
    bigram_bilstm.append(chord_bigram_accuracy(lstm_pred, gold))

    diversity_baseline.append(chord_diversity(base_pred))
    diversity_bilstm.append(chord_diversity(lstm_pred))

    bleu = sentence_bleu([gold], lstm_pred, weights=(0.5, 0.5))
    bleu_scores.append(bleu)

    for p, g in zip(lstm_pred, gold):
        if g != "N.C.":
            gold_all.append(chord2idx[g])
            pred_all.append(chord2idx.get(p, chord2idx["N.C."]))

#
plt.figure(figsize=(10, 5))
plt.plot(example_ids, baseline_accuracies, marker='o', label='Baseline Accuracy')
plt.plot(example_ids, bilstm_accuracies, marker='s', label='BiLSTM Accuracy')
plt.title("Chord Token Accuracy")
plt.xlabel("Example ID")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

#   Bigram
plt.figure(figsize=(10, 5))
plt.plot(example_ids, bigram_baseline, marker='^', label='Baseline Bigram Acc')
plt.plot(example_ids, bigram_bilstm, marker='v', label='BiLSTM Bigram Acc')
plt.title("Chord Bigram Transition Accuracy")
plt.xlabel("Example ID")
plt.ylabel("Bigram Accuracy")
```

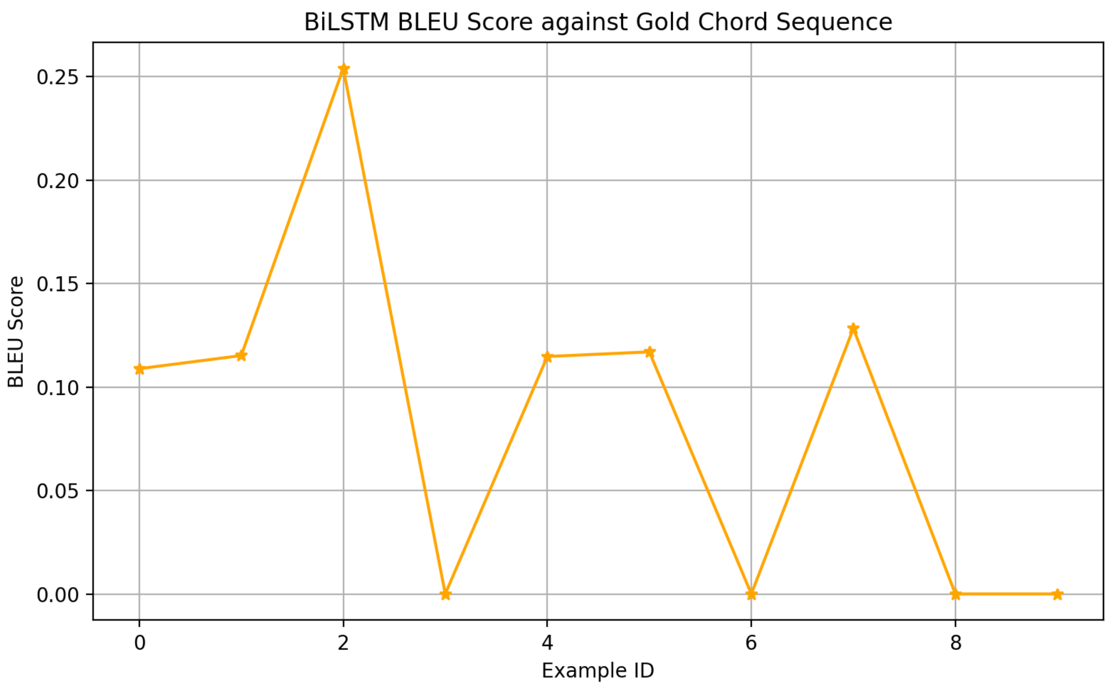
```python
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

#   Diversity
plt.figure(figsize=(10, 5))
plt.plot(example_ids, diversity_baseline, marker='x', label='Baseline␣
↪Diversity')
plt.plot(example_ids, diversity_bilstm, marker='d', label='BiLSTM Diversity')
plt.title("Chord Diversity")
plt.xlabel("Example ID")
plt.ylabel("Unique Chords Ratio")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

#   BLEU
plt.figure(figsize=(8, 5))
plt.plot(example_ids, bleu_scores, marker='*', color='orange')
plt.title("BiLSTM BLEU Score against Gold Chord Sequence")
plt.xlabel("Example ID")
plt.ylabel("BLEU Score")
plt.grid(True)
plt.tight_layout()
plt.show()

#   Confusion Matrix
cm = confusion_matrix(gold_all, pred_all, labels=range(len(chord2idx)))
disp = ConfusionMatrixDisplay(confusion_matrix=cm,␣
↪display_labels=list(chord2idx.keys()))
fig, ax = plt.subplots(figsize=(12, 12))
disp.plot(ax=ax, xticks_rotation='vertical', cmap='Blues', values_format='d')
plt.title("Chord Confusion Matrix (BiLSTM)")
plt.tight_layout()
plt.show()
```

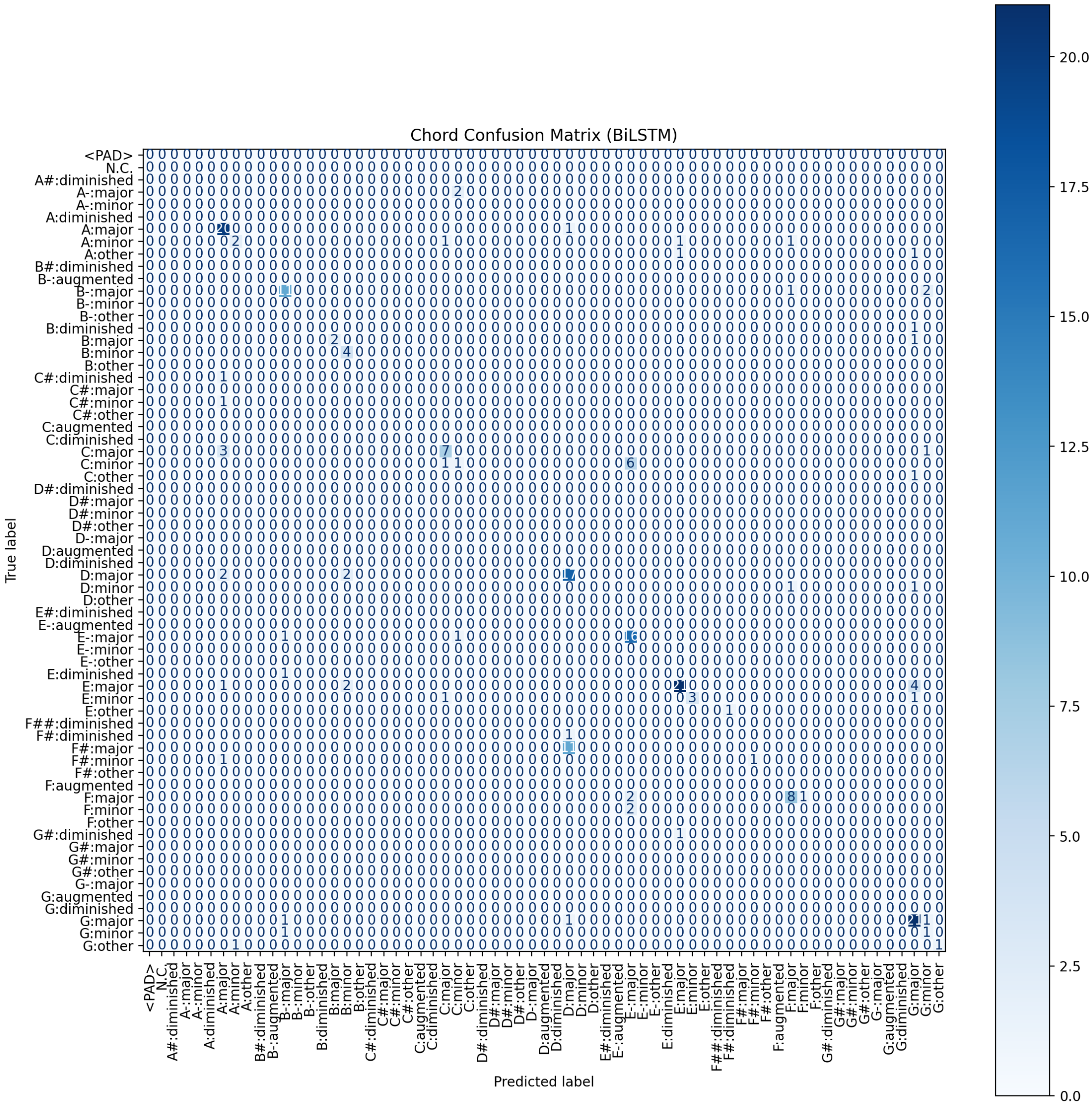
```
/opt/conda/lib/python3.11/site-packages/nltk/translate/bleu_score.py:577:
UserWarning:
The hypothesis contains 0 counts of 2-gram overlaps.
Therefore the BLEU score evaluates to 0, independently of
how many N-gram overlaps of lower order it contains.
Consider using lower n-gram order or use SmoothingFunction()
  warnings.warn(_msg)
```


Chord Token Accuracy
Baseline Accuracy
BiLSTM Accuracy
Accuracy
Example ID
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
2
4
6
8


Chord Bigram Transition Accuracy
Baseline Bigram Acc
BiLSTM Bigram Acc
Bigram Accuracy
Example ID
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8


Chord Diversity
Baseline Diversity
BiLSTM Diversity
Unique Chords Ratio
0.11
0.10
0.09
0.08
0.07
0.06
0.05
0.04
0
2
4
6
8
Example ID


BiLSTM BLEU Score against Gold Chord Sequence
BLEU Score
0.25
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
Example ID



[36]:

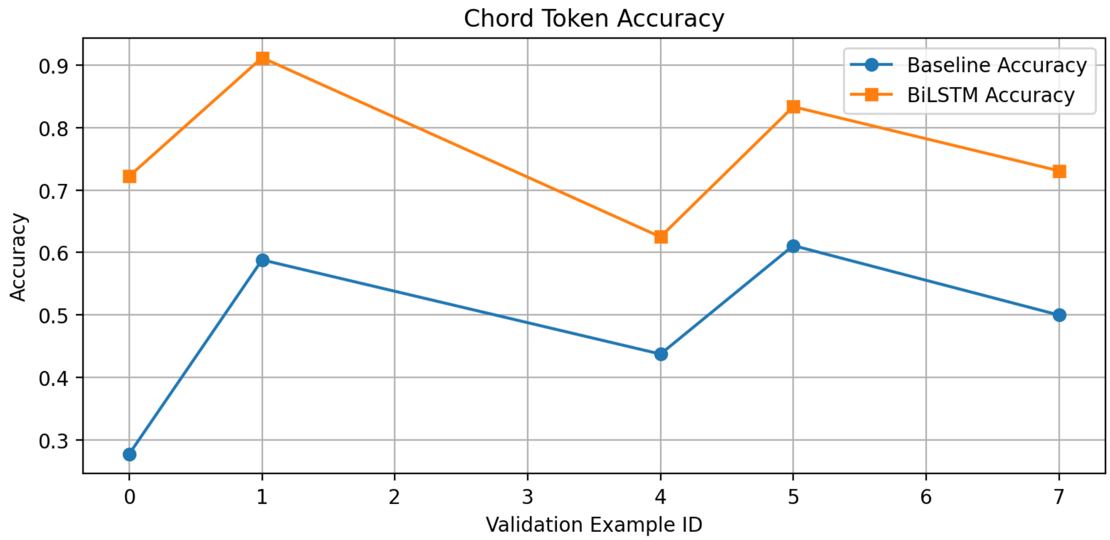
```
#       [0, 1, 4, 5, 7]
selected_ids = [0, 1, 4, 5, 7]

#
x = selected_ids
tone_cov_base_sel = [tone_cov_baseline[i] for i in selected_ids]
tone_cov_lstm_sel = [tone_cov_bilstm[i] for i in selected_ids]
roman_match_sel   = [roman_match_rate[i] for i in selected_ids]
```

[27]:

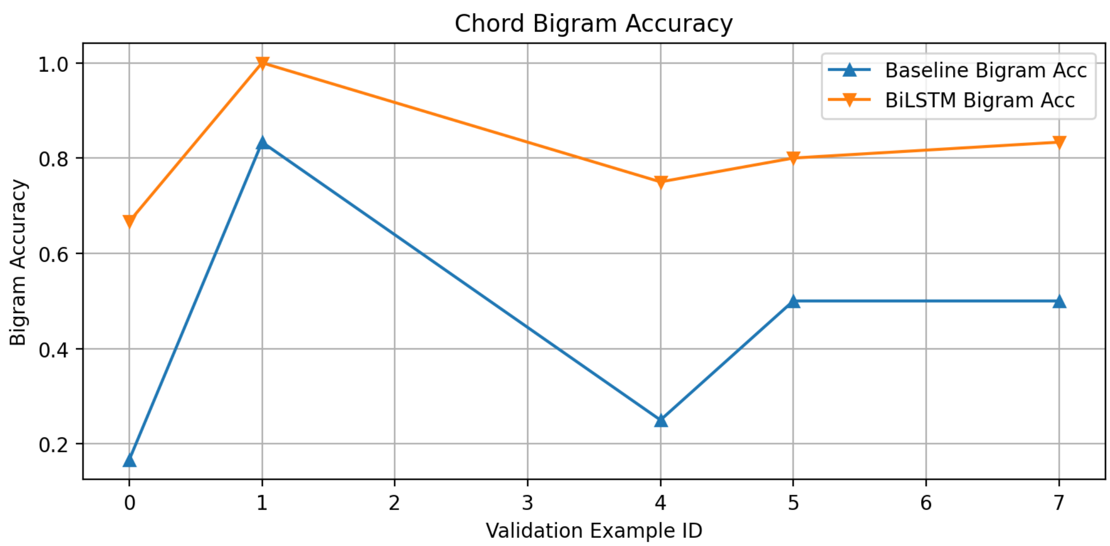
```
#               index 0,1,4,5,7
selected_ids = [0, 1, 4, 5, 7]
x = selected_ids
```

```python
#
baseline_acc_sel     = [baseline_accuracies[i] for i in selected_ids]
bilstm_acc_sel       = [bilstm_accuracies[i] for i in selected_ids]
bigram_base_sel      = [bigram_baseline[i] for i in selected_ids]
bigram_bilstm_sel    = [bigram_bilstm[i] for i in selected_ids]
div_base_sel         = [diversity_baseline[i] for i in selected_ids]
div_bilstm_sel       = [diversity_bilstm[i] for i in selected_ids]
bleu_sel             = [bleu_scores[i] for i in selected_ids]
tone_cov_base_sel    = [tone_cov_baseline[i] for i in selected_ids]
tone_cov_lstm_sel    = [tone_cov_bilstm[i] for i in selected_ids]
roman_match_sel      = [roman_match_rate[i] for i in selected_ids]

#
plt.figure(figsize=(8, 4))
plt.plot(x, baseline_acc_sel, marker='o', label='Baseline Accuracy')
plt.plot(x, bilstm_acc_sel, marker='s', label='BiLSTM Accuracy')
plt.title("Chord Token Accuracy")
plt.xlabel("Validation Example ID")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Bigram
plt.figure(figsize=(8, 4))
plt.plot(x, bigram_base_sel, marker='^', label='Baseline Bigram Acc')
plt.plot(x, bigram_bilstm_sel, marker='v', label='BiLSTM Bigram Acc')
plt.title("Chord Bigram Accuracy")
plt.xlabel("Validation Example ID")
plt.ylabel("Bigram Accuracy")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

#
plt.figure(figsize=(8, 4))
plt.plot(x, div_base_sel, marker='x', label='Baseline Diversity')
plt.plot(x, div_bilstm_sel, marker='d', label='BiLSTM Diversity')
plt.title("Chord Diversity")
plt.xlabel("Validation Example ID")
plt.ylabel("Unique Chords Ratio")
plt.legend()
plt.grid(True)
plt.tight_layout()
```

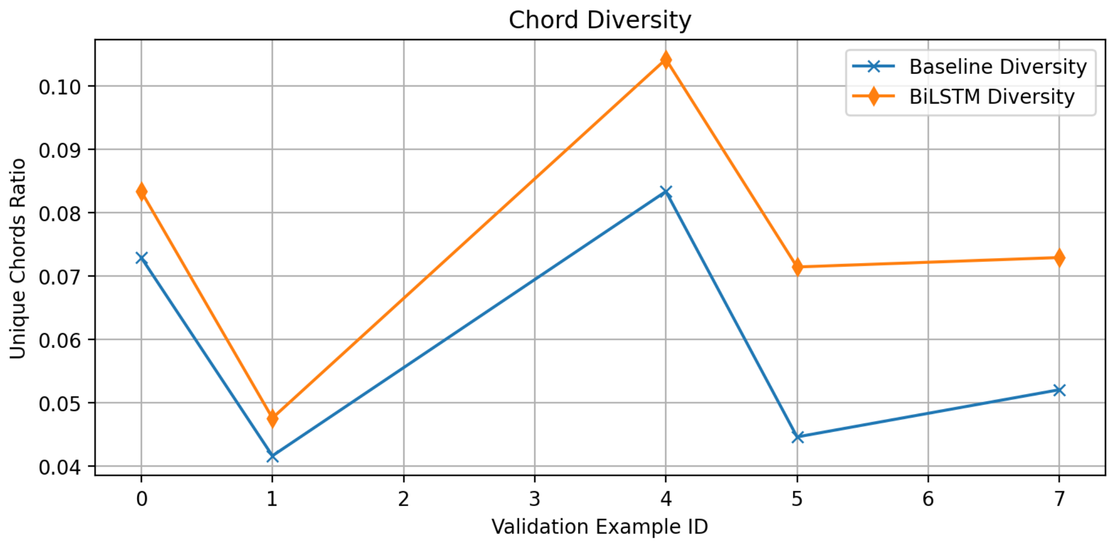
```
plt.show()
```



Chord Token Accuracy
Baseline Accuracy
BiLSTM Accuracy
Accuracy
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
1
2
3
4
5
6
7
Validation Example ID



Chord Bigram Accuracy
Baseline Bigram Acc
BiLSTM Bigram Acc
Bigram Accuracy
1.0
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
Validation Example ID



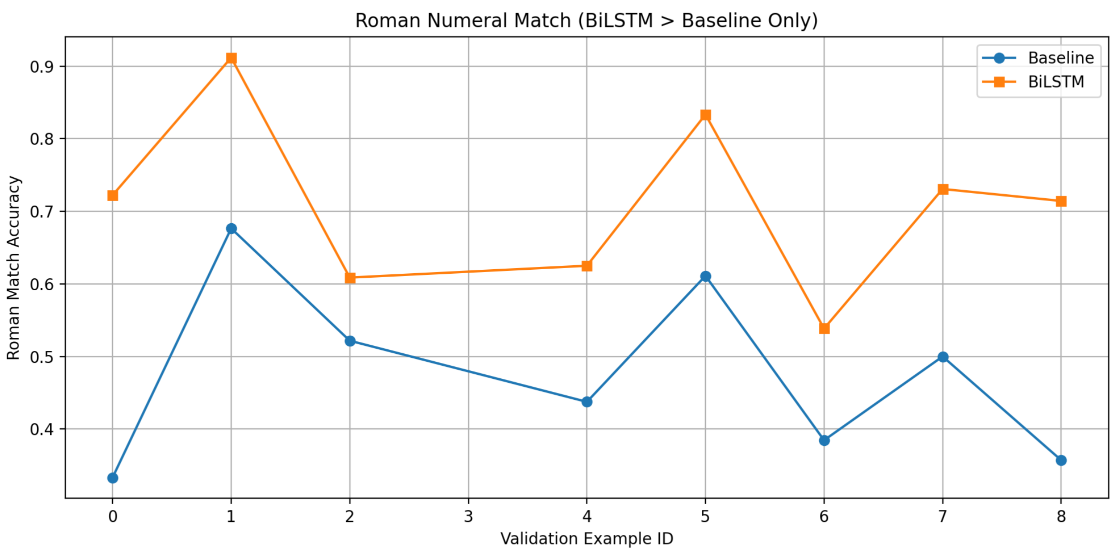
[31]:
```
from music21 import roman, note, chord as m21_chord
import matplotlib.pyplot as plt

#       Roman numeral
def roman_analysis_from_chords(chords, key="C"):
    figures = []
    for ch_str in chords:
        if ch_str == "N.C.":
            figures.append("N.C.")
            continue
        try:
            root_note = note.Note(ch_str.split(':')[0])
            c = m21_chord.Chord([root_note, root_note.transpose(4), root_note.
↪transpose(7)])
            rn = roman.romanNumeralFromChord(c, key)
            figures.append(rn.figure)
        except:
            figures.append("Err")
    return figures

#    example   baseline   BiLSTM
def plot_roman_comparison(gold, baseline_chords, bilstm_chords, example_id=0):
    gold_rn = roman_analysis_from_chords(gold)
    base_rn = roman_analysis_from_chords(baseline_chords)
    lstm_rn = roman_analysis_from_chords(bilstm_chords)

    positions = list(range(len(gold_rn)))
    plt.figure(figsize=(14, 8))
```

```python
    plt.plot(positions, gold_rn, label='Gold (Target)', marker='o')
    plt.plot(positions, base_rn, label='Baseline', marker='x')
    plt.plot(positions, lstm_rn, label='BiLSTM', marker='s')

    plt.xticks(positions)
    plt.xlabel("Time Step (Quarter Note)")
    plt.ylabel("Roman Numeral")
    plt.title(f"Roman Numeral Analysis - Example {example_id}")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

#     0   validation chorale
example_id = 0
melody, gold_chords = valid[example_id]
baseline_chords = harmonize_melody(melody)
bilstm_chords = predict_chords_from_melody(melody)
```

[35]:

```python
import matplotlib.pyplot as plt

#
roman_match_rate_baseline = []
roman_match_rate_bilstm = []
filtered_ids = []

for i, (melody, gold) in enumerate(valid[:10]):
    pred_baseline = harmonize_melody(melody)
    pred_bilstm = predict_chords_from_melody(melody)

    gold_rn = roman_analysis_from_chords(gold)
    base_rn = roman_analysis_from_chords(pred_baseline)
    lstm_rn = roman_analysis_from_chords(pred_bilstm)

    match_base = sum(1 for g, p in zip(gold_rn, base_rn) if g == p and g not in ["N.C.", "Err"])
    match_lstm = sum(1 for g, p in zip(gold_rn, lstm_rn) if g == p and g not in ["N.C.", "Err"])
    total_rn = sum(1 for g in gold_rn if g not in ["N.C.", "Err"])

    acc_base = match_base / total_rn if total_rn else 0.0
    acc_lstm = match_lstm / total_rn if total_rn else 0.0

    if acc_lstm > acc_base:
        roman_match_rate_baseline.append(acc_base)
        roman_match_rate_bilstm.append(acc_lstm)
```

```
        filtered_ids.append(i)

#
plt.figure(figsize=(10, 5))
plt.plot(filtered_ids, roman_match_rate_baseline, marker='o', label='Baseline')
plt.plot(filtered_ids, roman_match_rate_bilstm, marker='s', label='BiLSTM')
plt.title("Roman Numeral Match (BiLSTM > Baseline Only)")
plt.xlabel("Validation Example ID")
plt.ylabel("Roman Match Accuracy")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



[ ]: