ChatGPT

Implementation Plan: XMIDI Music Genre Classifier

This plan describes how to build a music genre classifier on the **XMIDI dataset** (108,023 MIDI files with emotion & genre labels [1] [2]). The classifier will recognize six genres – **classical, jazz, funk, pop, rock, and song** – from short MIDI phrases (roughly a *call-and-response* phrase length). We use an **encoder-only Transformer** model (DistilBERT-like) with a classification head. The plan covers data preparation, MIDI tokenization choices, model architecture, training setup, and options to demo the result. Potential challenges (data size, sequence length, etc.) are noted with realistic mitigation strategies.

1. Data Pipeline

Objective: Prepare XMIDI data into tokenized sequences with genre labels, ready for transformer input. The XMIDI files are in standard `.midi` format with labels embedded in filenames (emotion + genre). We will ignore emotion labels and focus on genre.

Key Steps to Prepare and Tokenize Data:

1. **Data Acquisition & Organization:** Download the XMIDI dataset from the provided source (e.g. Google Drive link) [3]. This yields ~108k MIDI files, named as `XMIDI_<Emotion>_<Genre>_<ID>.midi`. Organize files on disk by genre or keep an index of file paths. Due to the large dataset size (~5,278 hours of music [4]), ensure there is enough storage and use a structured directory (e.g., a folder per genre or a CSV listing file paths and labels).

2. **Label Extraction:** Parse each file's name to get the `<Genre>` tag (e.g. a file `XMIDI_happy_jazz_00001234.midi` has genre **jazz**). Map each target genre ("classic" likely corresponds to *classical*) to a numeric label (0–5). If the dataset contains genres beyond the six target ones (XMIDI also includes *country*, *folk*, etc. [5]), filter out or ignore those files so the model focuses on the six specified classes. Maintain a mapping dictionary, e.g. `{'classical':0, 'jazz':1, 'funk':2, 'pop':3, 'rock':4, 'song':5}` for label alignment. (If "song" is not an official XMIDI genre, it may refer to a miscellaneous category; we will assume any file labeled as "song" or uncategorized goes to this class.)

3. **MIDI Parsing:** Convert each MIDI file into a sequence of musical events. Use a Python library like **Mido**, **pretty_midi**, or **MidiToolKit** to parse MIDI content (notes, velocities, timings, etc.). Each MIDI may have multiple tracks; for genre classification, we can merge tracks or focus on the melody track, since genre is a property of the entire piece. A consistent approach is required (e.g., combine all tracks into one event stream with instrument tokens to preserve instrumentation, since instrument choice is a genre cue). Make sure to handle tempo and time signature events as needed, or set a default if absent.

4. **Segmentation into Phrases:** Since classification is based on short phrases (not full songs), break each MIDI sequence into **short segments**. For example, use a fixed length in musical terms – e.g. **4 bars** (measures) per segment or ~5–10 seconds of music. This "phrase length" should be long enough to convey genre cues (rhythmic pattern, harmony), but short enough for the model's input length. A jazz call-and-response phrase (often 2–4 bars) could guide the choice. If a MIDI piece is long, extract multiple non-overlapping segments from it (e.g., take the first 4 bars, then the next 4, etc.) to increase training samples. Ensure each segment inherits the original file's genre label. *Challenge:* segments from the same song will be very similar; to avoid overfitting, split the dataset **by song** for train/validation/test (so no segment of a training song appears in test). Optionally, add some randomness: choose a random start bar for each piece to get a representative snippet (this can be done per epoch as data augmentation, or once upfront to fix the dataset).

5. **Tokenization of MIDI Events:** Convert the events in each segment into a sequence of **tokens** (discrete symbols) that a transformer can ingest. This requires choosing a MIDI *token representation* (see Section 2). For now, assume we use an event-based encoding like REMI (Revamped MIDI) – this will yield tokens such as `Bar`, `Position`, `Pitch`, `Velocity`, `Duration`, etc. For example, a short phrase might tokenize to: `[Bar_0, Position_0, Pitch_60, Velocity_80, Duration_1.0, Position_8, Pitch_64, ...]`. Using an existing tokenizer library (e.g. **miditok** or **MusicBERT's Octuple script**) can automate this conversion. The output should be a sequence of **integer IDs** representing the MIDI events.

6. **Prepare Transformer Inputs:** Determine a **maximum sequence length** for the model (e.g. 512 tokens). Many short phrases will be fewer tokens, but some could be longer if the phrase is dense. We will **pad** sequences to the max length and use an attention mask to ignore padding. If a segment exceeds the max length, we can truncate or ideally choose a shorter segment length to begin with so truncation is rare (losing musical info might hurt classification). It's wise to analyze typical token sequence lengths after tokenization to set a proper max length.

7. **Dataset Construction:** Using the tokenized segments and labels, create a dataset object. The **Hugging Face Datasets** library is ideal: we can store each example as `{"input_ids": [...tokens...], "label": genre_id}`. We can use `.map()` to apply the tokenization function to raw MIDI files on the fly, avoiding storing large intermediate data in memory. Split the dataset into training, validation, and test sets (e.g. 80/10/10 split of songs). Ensure class distribution is maintained in splits. If some genres (like *funk* or *song*) have far fewer samples, consider stratified split or data augmentation (e.g. transposing MIDI up or down a step for more samples without changing genre characteristics).

Data Format Conversion: The critical format conversion is turning **MIDI** into a sequence of token IDs. This involves encoding time (beats/bars) and notes as discrete vocabulary symbols. We will use a special vocabulary for the chosen tokenizer – for REMI, tokens include note pitches (e.g. Pitch_60), rhythmic positions (Position_16 etc.), durations, etc. These will be mapped to unique IDs (e.g., Pitch_60 -> id 45, Position_16 -> id 12, etc. as defined by the tokenizer). The transformer's embedding layer will later map these IDs to vectors. Additionally, we'll reserve special IDs for padding (`[PAD]`) and possibly a classification token (`[CLS]`). Each training sample sequence might be prefixed with `[CLS]` at index 0 to indicate where the model should output the classification (as done in BERT). The label for that sequence is the genre id.

Label Alignment: Each token sequence (representing one phrase) is paired with one genre label. Using the file-naming convention ensures correct alignment. We will build a lookup or directly parse filenames so that tokenization and label assignment happen together. For example, in a data pipeline script, for each MIDI file: parse genre, tokenize to `input_ids`, and store the tuple `(input_ids, genre_label)`. By the end, we have a list/dataset of thousands of such samples. The labels will be integers [0–5] corresponding to the six genres. We must ensure the label order is consistent throughout training and evaluation.

Potential Challenges & Mitigations:

- **Data Size & Loading:** 108k MIDI files plus segmentation could yield a very large number of training samples (potentially several hundred thousand segments). Loading all into memory may be infeasible. Using **streaming data processing** (with HF Datasets or PyTorch `DataLoader` with on-the-fly tokenization) can help. Also, start with a subset (e.g. 10k files or 1k files) to verify pipeline before scaling up.

- **MIDI Variability:** MIDI files might have varying numbers of tracks and inconsistent formatting. We should decide on handling multi-track (e.g., merge into one token sequence with program/instrument tokens distinguishing instruments). The chosen tokenizer often can handle multi-track (e.g., REMI+ or Octuple include program tokens [6]). If merging tracks, ensure that tokens preserve the timing (sort events by time). If a file has an unusual time signature or tempo changes, the tokenizer should account for those (some tokenizers add TimeSignature and Tempo tokens).

- **Phrase Label Consistency:** All segments from one song share the same genre label. We must avoid a situation where the model simply memorizes a piece from training and recognizes a segment of it in validation (hence the need to split by song). Also, short segments might occasionally be ambiguous in genre (e.g., a generic drum beat could fit rock or pop). We assume overall the dataset phrases carry distinct genre style, but it's something to watch. If ambiguous, longer segments or multiple random segments might be needed to capture enough context.

By completing this data pipeline, we will have a ready-to-use tokenized dataset of short MIDI phrases with genre labels, suitable for transformer input.

2. MIDI Tokenizer Strategy

Symbolic music (MIDI) must be converted into token sequences for a transformer. There are several tokenization strategies in prior research. We compare three notable methods and then recommend one for our classifier:

- **REMI (REvamped MIDI)** – an event-based representation from **Pop Music Transformer** [7].
- **Compound Word (CP)** – a grouped representation from **Compound Word Transformer** (Hsiao et al. 2021) [8].
- **Octuple (OctupleMIDI)** – a compact 8-dimensional representation used in **MusicBERT** (Microsoft, 2021) [9] [10].

Below is a comparison of these tokenization options:

Tokenizer	Key Idea & Sequence Format	Pros	Cons
REMI (Event-based)	Represents each musical event with separate tokens (e.g. Pitch, Velocity, Duration), and uses special tokens for timing (Bar and Position in bar) [7]. Example sequence: Bar, Position, Pitch, Velocity, Duration, ...	- Simple, intuitive representation of notes and timing. - Proven effective in music transformers [7]. - Easy to implement with off-the-shelf code.	- **Long sequences**: Each note expands to multiple tokens (pitch/vel/dur + time tokens), so polyphonic music yields very long sequences. - More tokens ⇒ higher memory and compute per sequence.
Compound Word (CP)	Groups note attributes into one *compound token*. Uses **embedding pooling**: pitch, velocity, duration are embedded then merged into a single token representation [8]. Time is still marked with bar/ position tokens.	- **Shorter sequences** than REMI (note info compressed into one token). - Retains explicit timing tokens (bar/ position) for rhythmic context. - Successful in prior work for generating full songs.	- More **complex to implement**: model must output multiple attributes per token (multi-head output) [11], which complicates training. - Not readily supported by standard Transformer classes (requires custom code for multi-attribute tokens). - Less tested for classification tasks.
Octuple (Note-level tuple)	Encodes each note as an **8-element tuple**: e.g. *(Pitch, Position, Bar, Velocity, Duration, Instrument (Program), Tempo, TimeSignature)* [10]. All aspects of a note event are combined into one token embedding [9].	- **Very compact** sequence (notes as single tokens). Often yields ~4× shorter sequences than REMI [12], easing Transformer modeling of long music sequences. - Handles multi-track music naturally (Program/instrument included in token). - Effective in MusicBERT, which achieved state-of-art on music understanding tasks [13].	- **High complexity**: requires specialized tokenizer and model embedding. The model must split each token's embedding into multiple feature vectors internally (for each of the 8 attributes) [14], or one must implement a custom embedding layer. - If treated naively as a single ID per tuple, the vocabulary size could explode (many possible combinations). - Harder to debug encoding errors due to compactness.

Recommendation: For our genre classifier, **REMI (or a similar event-based tokenizer)** is the most pragmatic choice. It strikes a balance between information fidelity and implementation simplicity. REMI explicitly represents note events and timing in a linear token sequence, which we can feed directly into a standard Transformer encoder [7]. Its vocabulary (consisting of tokens like `Pitch_###`, `Velocity_###`, `Duration_###`, `Bar`, `Position_#`) is manageable and can be used with a normal embedding layer.

While REMI sequences can be long, we mitigate that by using short musical phrases as input (so the number of tokens per sample is limited). For example, a 4-bar phrase might result in on the order of 100–300 tokens, which is within a reasonable length for a transformer.

Using REMI, we can leverage existing libraries (e.g. **Miditok** or **Mido** with custom code) to tokenize MIDI: - **Miditok** directly supports REMI and variants; it can output a list of token IDs from a MIDI file with a few lines of code. It also supports adding *Program* tokens to handle multi-instruments (REMI+ extension) [15], which we might enable so the model knows which instruments appear (instrumentation is often genre-specific). - If not using a library, implementing REMI involves iterating through sorted MIDI events and outputting tokens in order: when a new bar starts, output a `Bar` token; for each note, output `Position_{beatFraction}`, then `Pitch_X`, `Velocity_Y`, `Duration_Z` tokens. This is labor-intensive and error-prone, so a well-tested library is preferred.

Alternative: Octuple encoding is very appealing for compact sequences – if we had to classify *entire songs* or very long sequences, Octuple would reduce memory usage drastically [12]. However, adopting Octuple would require customizing the model to accept 8-field tokens (like the MusicBERT architecture did). Since our model is a relatively standard DistilBERT-like encoder, it's much easier to feed it a flat sequence of single-valued tokens (REMI or CP). Compound Word (CP) is something of a middle ground, but it similarly complicates the model (needing multiple loss outputs if we were generating notes [11], though for classification we might bypass that). CP and Octuple are more beneficial in **music generation** tasks to handle sequence length, but for **genre classification on short segments**, the simplicity of REMI outweighs the benefits of the more compact encodings.

Tokenizer Implementation Notes: We will adopt **REMI** tokenization using the Miditok library for reliability. We will configure it to include necessary token types: - Use `Bar` tokens (to mark measure boundaries) and a fixed **beat resolution** (e.g. 24 or 48 ticks per beat) to derive `Position` tokens within each bar. - Include `Pitch` tokens for note-on events, `Velocity` tokens for note dynamics, and `Duration` tokens instead of separate note-off events (as REMI defines) [7]. - Enable multi-track support by adding `Program` tokens (this is essentially REMI+ [15]) so that notes from different instruments are tagged — this helps the model learn, for example, that a presence of distorted guitar (program for electric guitar) is indicative of rock/metal genre, or a string ensemble program suggests classical. If multi-track data proves too complex, an alternative is to use single-track by merging all notes and ignoring instrument tokens, though that loses some info. - Exclude tempo and time signature tokens for now (unless needed). If the MIDI files cover varied tempi or time signatures by genre, including those could provide cues (e.g. many rock songs are 4/4, ~120bpm; a waltz is 3/4 which might imply classical or folk). Miditok can add Tempo and TimeSig tokens; we'll consider adding them if beneficial.

After tokenization, we will have a dictionary of tokens to IDs (the vocabulary). The vocabulary size will be on the order of a few hundred tokens (e.g., 128 possible pitches, ~10 velocity bins if we quantize velocities, various durations, plus special tokens). This small vocab is efficient to handle. We will add special tokens: `[PAD]` for padding, `[CLS]` for the classifier token (and possibly `[SEP]` if needed, though here we deal with single sequences, so `[SEP]` might be unused).

In summary, the **tokenizer strategy is to use REMI tokens**, ensuring compatibility with a standard transformer encoder. This gives us an expressive, if somewhat verbose, sequence per music phrase, which our model can then learn to map to the correct genre.

3. Model Architecture

We will use a **DistilBERT-like transformer encoder** as the base of our genre classifier. DistilBERT is essentially a smaller BERT model: a stack of Transformer **encoder** layers (self-attention blocks) without any decoder component. It was shown that a 6-layer DistilBERT retains ~97% of BERT's performance while being 40% smaller and 60% faster [16], which is ideal for a resource-conscious solution. Our plan is to configure an encoder with a size inspired by DistilBERT, then add a classifier head on top.

Architecture Components:

- **Input & Embeddings:** The model takes as input the sequence of token IDs produced by the tokenizer. We will use a *learned embedding matrix* to convert each token ID into a dense vector (just like word embeddings in NLP). If using Hugging Face's API, this is handled by the model's embedding layer. The embedding size can be, for example, **256 or 512**. (DistilBERT uses 768 dimensions, but we might choose a slightly smaller size like 512 to reduce parameters given our task complexity is lower than natural language.) We will also use **positional embeddings** to encode the position of each token in the sequence (since Transformers have no inherent notion of order). DistilBERT uses sinusoidal or learned positional embeddings up to a max length (we'll set this to the max sequence length, e.g. 512). Thus, the input to the first Transformer layer is a sequence of vectors `[Embed(token_1)+Pos(1), Embed(token_2)+Pos(2), ...]`. We will also prepend a special `[CLS]` token at position 0 of every sequence; this gets its own embedding and serves as a summary representation for classification (same approach as BERT).

- **Transformer Encoder Layers:** The core of the model will be **N layers of Transformer encoder**. DistilBERT uses 6 layers, so we'll start with **6 self-attention layers**. Each layer has:

- Multi-head self-attention: we might use 8 attention heads (DistilBERT uses 12 heads with dim 64 each to total 768; if we use 512 dim, 8 heads of 64 dim each = 512 fits nicely).
- Feed-forward network: after attention, a position-wise fully connected feed-forward (two linear layers with a nonlinearity, typically GELU activation as in BERT).
- Layer normalization and residual connections around each sub-layer (standard Transformer structure).
- Dropout in attention and FFN for regularization (e.g. 10% dropout rate).

These layers allow the model to contextualize each token with respect to others in the phrase (capturing musical context like chord patterns, rhythms, instrumentation co-occurrence). We keep the model **encoder-only** (no generative decoder) since we just need a sequence representation, not sequence output.

- **Distillation or Pretraining:** We won't actually perform knowledge distillation as done to create DistilBERT (that required a teacher BERT model). Instead, we are using the *architecture style* of DistilBERT for efficiency. All transformer weights will be initialized from scratch (randomly, e.g. Xavier initialization) because we don't have a pre-trained weights specific to this token vocabulary. (Transferring actual DistilBERT weights is not feasible as-is – those weights encode English word relationships, not musical events.) We will rely on training data to learn the necessary patterns. We do inherit the **smaller size and depth** from DistilBERT so the model isn't overly large for our dataset. If needed, we can experiment with fewer layers (e.g. 4) or narrower layers (e.g. embedding dim 256) to further reduce complexity, especially if the training data per class is limited, to avoid overfitting.

- **Classifier Head:** On top of the final encoder layer, we add a classification head to predict the genre. This head will take the Transformer's output at the `[CLS]` position as input. Typically, the `[CLS]` token's final hidden state is considered a representation of the whole sequence (after the model has attended over all tokens). Our classifier head will be a simple feed-forward layer:

- Optionally, a dropout layer (e.g. 0.1) to further prevent overfitting.
- A dense linear layer mapping from the hidden size to the number of genres (6).
- Softmax activation to produce probabilities for each of the 6 classes (during inference).

In implementation terms, using Hugging Face's `DistilBertForSequenceClassification` is convenient: it automatically adds a classification layer on top of DistilBERT. We would configure it with `num_labels=6`. The architecture then is essentially DistilBERT base (6 layers, 768 hidden by default) + a classifier layer. We will adjust the config for our needs (e.g., if we want a smaller hidden size or different vocab size). The classification layer will be learned jointly during training.

- **Special Tokens and Masks:** We will include `[PAD]` tokens for padding shorter sequences to the max length. The model will use an **attention mask** to ignore padded positions (the Hugging Face model handles this if we provide `attention_mask` with 0s on padding tokens). We include a `[CLS]` token at start; BERT models usually learn a special embedding for this and treat its hidden state as the sequence representation for classification. We might not need a `[SEP]` token since we're not dealing with multiple sequences (in text tasks [SEP] separates sentences). However, if the HF model expects one (some BERT configs do), we can include it at sequence end – it won't harm anything. The important part is that the `[CLS]` token is present and its state is used by the classifier head.

Model Size and Capacity: Using ~6 layers and ~512 hidden size gives on the order of 20–30 million parameters (rough estimate), which is quite feasible on a single GPU. If using the full DistilBERT (768-dim, 66M params [16]), that's also manageable. The dataset is fairly large, so a moderately sized model can be trained without severe overfitting, but we will monitor validation loss. If the model seems to overfit (val accuracy peaks early then drops), we can reduce size or increase regularization. Conversely, if underfitting (not reaching good accuracy), we might try a slightly larger model or pretraining.

Genre Classification Specifics: Because genre is a high-level feature, certain **global patterns** in the MIDI will be important (instrument set, chord patterns, tempo). The transformer's self-attention is well-suited to capture global relationships – e.g., it can learn to pay attention to instrument tokens or certain rhythmic tokens that correlate with genre. We expect, for example, the presence of many fast *Swing* eighth-note positions and certain jazz instrument programs might cue **jazz**, while a steady 4/4 drum backbeat pattern with guitar chords might cue **rock**. The model's attention heads can learn these correlations across the sequence. By using an encoder architecture, we explicitly allow the model to build a holistic representation before making a decision, rather than making token-level predictions.

Summary: Our model is a **distilled Transformer encoder** with self-attention layers and a classification head. It will accept tokenized MIDI phrases and output one of 6 genre labels. Using Hugging Face's framework:

```python
from transformers import DistilBertConfig, DistilBertForSequenceClassification
config = DistilBertConfig(
    vocab_size=vocab_size,  # size of our MIDI token vocab
    dim=512,                # embedding and hidden dimensionality
    n_layers=6,
    n_heads=8,
    max_position_embeddings=512,  # max sequence length
    num_labels=6,           # number of genres
    dropout=0.1
)
model = DistilBertForSequenceClassification(config)
```

This will create the architecture as described. We'll then train this model from scratch on our dataset.

Challenges & Considerations:

- **Positional Embedding Limit:** If we choose a max length (say 512) but a few phrases exceed that, those extra tokens are truncated. This is acceptable if only a few outliers; otherwise, we may want to increase `max_position_embeddings` (with a slight memory cost) or reconsider segment length. We must ensure not to exceed what our GPU can handle in terms of sequence length times batch size.

- **Pad/CLS Impact:** We should verify that the `[CLS]` token embedding and position 0 are properly utilized by the model. Hugging Face's DistilBERT uses the first token's hidden state for classification by default. We just need to confirm that our data prep always places `[CLS]` at index 0 and that the label is attached to it. The `[PAD]` token should have an embedding but ideally not contribute to the sequence meaning; the attention mask will zero it out in attention calculations.

- **Activation Function:** The default in BERT/DistilBERT is GELU in the intermediate layers. We will stick to that unless we encounter issues, as it's been shown effective for these architectures.

- **Output Layer:** We'll use cross-entropy loss on the logits produced by the classifier head. The model will output logits of size [batch, 6]; applying softmax will give a probability distribution over the 6 genres.

Overall, the architecture is a straightforward adaptation of NLP classifiers to music tokens. Encoder-only transformers are a natural fit for sequence classification tasks, treating a sequence of MIDI events similarly to a sequence of words or characters, and DistilBERT's design makes it computationally efficient for our needs.

4. Training Environment

We plan a two-stage approach for the training environment: start local, then potentially scale up. Leveraging the **Hugging Face Transformers** and **Datasets** libraries will streamline training in either case.

Local Development (PC with GPU): - Begin on a local machine, ideally with a CUDA-compatible GPU. A single GPU with ~8–16 GB VRAM (common in gaming PCs or a workstation) should handle a moderate batch size for our model. This environment is used for prototyping: verifying the data pipeline, tokenizer, and model code. By training on a small subset of data (or a few epochs on full data), we can ensure everything runs without crashing and the model starts to learn (e.g., training loss decreases). - Set up the Python environment with necessary libraries: `transformers`, `datasets`, `miditok` (or other MIDI parsing lib), `torch` (PyTorch), plus tools for logging (e.g., `wandb` or TensorBoard). Ensure the GPU (if available) is recognized by PyTorch. - If no powerful GPU is available locally, consider using a free service like **Google Colab** or **Kaggle Kernels** for initial development (they provide Tesla T4/K80 GPUs for limited time). These can be used to run experiments on smaller scale before committing to full training.

Scaling Up: - **Full Dataset Training:** With ~100k pieces (and possibly several times that in segments), full training might be time-consuming. If local GPU is not very powerful or if training time is a concern, we plan to move to a more powerful environment after validating the approach. Options include cloud VM instances (AWS, GCP) with V100/A100 GPUs, or academic cluster with multi-GPU nodes. We might not need multiple GPUs unless we want to speed up experimentation; a single modern GPU (like an NVIDIA A100 40GB) could likely handle this in reasonable time with proper batching. - We will use **Hugging Face Trainer/Accelerate** to abstract hardware details. The Trainer API can automatically use multiple GPUs if available or use mixed-precision (fp16) training to speed up and save memory. We'll enable FP16 (especially on larger models) for efficiency. - If the dataset is extremely large (which it might be, depending on segments), we can use **data streaming** with `datasets.load_dataset` streaming mode, or write a custom PyTorch `Dataset` that loads data on the fly from disk. This avoids memory blow-up. We should also consider caching tokenized data to disk (the HuggingFace dataset library can save a preprocessed dataset in Arrow format for fast loading later).

Resource Considerations: - **Memory (RAM):** Tokenizing all data could use a lot of memory if done in one go. Using the `datasets` library, we can process in chunks or use `.map(batch_size=...)` to tokenize gradually. We might also shuffle and batch on the fly. The `datasets` library efficiently mmaps data from disk when possible. - **Disk:** Storing tokenized sequences for 100k files * maybe multiple segments each, say ~1e6 sequences, each of length 100–300 tokens, is not too bad (maybe a few hundred million integers at most, which is a few hundred MB). So it's feasible to store a processed dataset on disk for reuse. We should ensure to have fast disk I/O (SSD). - **Compute:** The chosen model (tens of millions of params) and sequence lengths (few hundred tokens) are moderate. Training should be on the order of a few hours to a day on a single high-end GPU for a few epochs. If that's too slow, multi-GPU data parallelism can be used (Trainer makes this easy; or use **HuggingFace Accelerate** for more control). - **Library Use:** We will heavily use Hugging Face: - `transformers` for model and training loop. - `datasets` for handling the dataset and possibly metrics (they have built-in accuracy/precision metrics we can use). - `miditok` for tokenization (once-off overhead). - Optionally `wandb` (Weights & Biases) for experiment tracking, which can log metrics, model gradients, etc., and is especially useful if training on remote servers (to monitor progress in a dashboard).

Progressive Scaling: We will adopt a *gradual scaling* strategy: - **Phase 1:** Train on a smaller subset (e.g., 5k samples) for a couple of epochs locally to ensure the model can differentiate genres above chance. This checks that the pipeline and model work end-to-end. - **Phase 2:** Train on the full dataset (or a large portion) on the best available hardware. If on local PC, possibly overnight runs for multiple epochs. If using cloud, allocate enough time (monitor GPU utilization to maximize it). - **Phase 3:** If results are not satisfactory or to further improve, explore scaling to multi-GPU or using more epochs. Given the dataset size, we likely won't need an extremely large number of epochs (maybe a few epochs might suffice, since each epoch is lots of data). But if training set per genre is still limited, more epochs or data augmentation might help.

Potential Issues & Solutions: - *Environment Setup Issues:* Installing `miditok` can bring in many dependencies (like Mido). We will document environment setup (Python version, pip requirements) clearly. Using a consistent environment (Conda or requirements.txt) will help reproducibility. - *GPU Memory Overflow:* If we see OOM errors, reduce batch size or sequence length. Gradient accumulation can mimic a larger batch if needed (Trainer supports `gradient_accumulation_steps` to accumulate gradients over multiple mini-batches). - *Long Training Time:* If on a local PC, training might be slow. We might use a smaller model first. We can also checkpoint the model periodically so we can resume if needed rather than start over (Trainer does this automatically every epoch or as configured). - *Debugging:* Train with `debug` mode on small data to ensure loss is decreasing. If model isn't learning at all, something may be wrong in data labeling or model configuration.

In summary, the environment plan is to **prototype locally with available GPUs and libraries**, then **scale to larger compute** if needed for the full training. Using Hugging Face's tools ensures we can seamlessly move from one environment to another (since the code remains the same, just the hardware changes). We will keep the setup flexible to adapt to either scenario.

5. Training Process

With data and model ready, the training process will involve fine-tuning the transformer on the genre classification task. We outline the training loop, evaluation, and any additional techniques (pretraining, transfer learning) to consider:

Data Splitting: First, split the dataset into Train, Validation, and Test sets. A reasonable split is 80% train, 10% val, 10% test. Ensure that all segments from a given MIDI file belong to the **same split** to prevent the model from seeing part of a song in training and another part in validation. We also ensure each genre is represented in each split (stratified split by genre). The **validation set** will be used to tune hyperparameters and decide when to stop training (early stopping), and the **test set** will evaluate final model performance.

Training Loop & Hyperparameters: - We will train using **supervised learning** with **cross-entropy loss**. Each input sequence has a single genre label, so this is a standard multi-class classification. The model's final layer outputs logits `[logit_class0, ..., logit_class5]` and we use `nn.CrossEntropyLoss` which internally applies softmax and compares to the true label. - **Optimizer:** Use **AdamW** (Adam optimizer with weight decay) which is recommended for transformers. We might start with learning rate ~ **1e-4** for training from scratch. (When fine-tuning pre-trained BERT, a smaller LR ~2e-5 is used, but with a new model on a new task, we can go higher). We'll use weight decay ~0.01 to regularize, except we will not decay the bias and layernorm weights (standard practice). - **Learning Rate Schedule:** A linear decay schedule with warm-up is common for transformer training. For example, a short warm-up over 5% of training steps to let the model stabilize, then linearly decrease LR to 0 by the end. The Hugging Face Trainer can apply this automatically if we specify `warmup_steps` or a fraction. This helps training converge more smoothly. - **Batch Size:** Choose the largest batch size that fits in memory for efficiency. Perhaps start with batch size 32 sequences (this depends on sequence length and model size). If using fp16, we might squeeze in more. If on smaller GPU, batch size 8 or 16 may be necessary. We can adjust and use gradient accumulation if we want an effective larger batch for stability. - **Epochs:** Because the training set (80k songs * maybe ~5 segments each → ~400k samples, as a rough figure) is large, one epoch is already a lot of steps. We might not need many epochs. We could start with **3-5 epochs** and monitor metrics. Genre classification should improve within a couple epochs if learnable. If training longer, risk of overfitting increases, especially if the model memorizes training songs. We will rely on validation accuracy to decide when to stop. If validation accuracy plateaus or starts dropping, we stop (early stopping). We will save the model at the epoch with best val accuracy. - **Evaluation Metrics:** The primary metric is **classification accuracy** (overall percentage of segments correctly classified by genre). We will compute accuracy on the validation set at the end of each epoch. Additionally, because class distribution might be uneven, we'll compute **macro-averaged F1 score** to ensure each genre's performance is considered. We can also look at the confusion matrix to see if certain genres get confused (e.g., maybe "funk" vs "jazz" could confuse the model if the phrase is ambiguous). The Hugging Face `datasets` library can integrate with sklearn or its own metrics for computing F1, etc. We will log accuracy and potentially F1 each epoch. For a more detailed view, we might occasionally evaluate precision/recall per class to know which genres are lagging. - **Logging:** We will use the Trainer's logging to output training loss every few batches and evaluation metrics at epoch end. Optionally, integrate Weights & Biases (`wandb`) for nice graphs of loss vs. steps, accuracy vs. epoch, etc. Logging is crucial if training on a remote server so we can watch progress. We'll also keep an eye on GPU utilization (to ensure we're efficiently using the hardware).

Training Execution (HuggingFace Trainer): Using the Trainer simplifies the loop:

```
from transformers import TrainingArguments, Trainer
training_args = TrainingArguments(
    output_dir="./genre_model",
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=5,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    logging_steps=100,  # log every 100 batches
    learning_rate=1e-4,
    weight_decay=0.01,
    warmup_ratio=0.05,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
    save_total_limit=2,  # keep only 2 model checkpoints
    fp16=True  # use mixed precision if available
)
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics_function
)
trainer.train()
```

Here, `compute_metrics_function` will calculate accuracy (and optionally F1) from the model's predictions. We set it to monitor accuracy for saving the best model. Logging steps of 100 means every 100 gradient steps it will report loss (we can adjust frequency depending on dataset size).

Monitoring and Overfitting: As training progresses, we'll monitor: - Training loss decreasing — indicates the model is fitting the data. - Validation accuracy increasing — indicates generalization. We expect it to start maybe around ~1/6 (random guess ~16%) and climb significantly if the model learns to identify genres. - If training loss keeps dropping but validation stalls or worsens, that's overfitting. To combat this, we have in place: - Dropout layers (0.1) in the model. - Weight decay (0.01). - We can also reduce epochs or use early stopping if we see this pattern. - If overfit is still an issue, we could further simplify the model or use more aggressive regularization (e.g., increase dropout to 0.3). - If underfitting (both training and val accuracy are low and not improving much), possibly the model needs more capacity or the learning rate is off. We'd try a bit higher LR or allow more epochs. We could also incorporate more of the MIDI sequence (maybe using longer than 4 bars) to give more info per sample.

Pretraining or Transfer Learning Considerations: - Since our model is trained from scratch on the classification objective, it might benefit from **pretraining on unlabeled music data** to learn general musical patterns. For example, we could do a self-supervised pretraining like **Masked Language Modeling (MLM)** on the MIDI tokens (similar to BERT's text pretraining). This would involve randomly masking some tokens in the sequence and training the model to predict them. The XMIDI dataset itself (108k pieces) could be used for MLM pretraining (ignoring labels) – it's large enough to learn common structures (chords, scales, rhythms). After a phase of pretraining, we'd then fine-tune on the genre labels. This two-step process might yield better feature representations, especially for subtle genre features. However, pretraining will add significant time and complexity. If initial training results are subpar, we will consider an MLM pretraining for a few epochs on all data (maybe using MusicBERT's bar-level masking idea to avoid trivial clues [13]). If initial results are already good (e.g., >80% accuracy), pretraining might not be necessary. - Another form of transfer learning: If an existing model like **MusicBERT** (pre-trained on a million MIDIs [17]) was publicly available, we could try to fine-tune it for our task. MusicBERT already uses Octuple encoding and was shown effective for genre classification [13]. But integrating that could be complex: we'd have to use Octuple tokenization and the exact architecture. Given our plan focuses on an encoder we build, we stick to that. We note it as an inspiration: MusicBERT's success suggests our approach is viable, but we won't directly reuse it due to encoding differences.

- We also consider data augmentation to assist learning: In addition to random phrase extraction and transposition as mentioned, we could augment training data by slight tempo variations or adding small random noise in timing/velocity (making the model robust to performance differences). Since genre is a high-level attribute, moderate augmentation shouldn't flip the genre label, but can make the model invariant to key or tempo. We must ensure not to break the musical sense (e.g., extreme tempo change might alter feel from ballad to upbeat, potentially confusing genre). Likely, simple pitch transposition (± a few semitones) on some MIDI files is safe and can multiply data for low-count genres. We can implement this in the data pipeline (e.g., for each MIDI in funk, produce a version transposed up a whole tone).

Evaluation and Testing: After training, we'll evaluate on the held-out **test set** (which the model never saw). We'll report overall accuracy and per-genre performance. For example, we might find the model gets 90%+ on clear categories like classical vs rock, but maybe "funk" vs "jazz" gets confused occasionally (we'll check the confusion matrix). These insights could drive further tuning (maybe adding more training data for confusing pairs or refining the input representation).

Logging & Experiment Tracking: Throughout training, we will log: - Epoch and batch progress, - Training loss, - Validation loss and accuracy, - Possibly example predictions on the validation set after each epoch (to qualitatively see if it's making sensible predictions). Using `wandb`, we would have nice charts and also record hyperparameters for each run, which helps when trying multiple runs (say, different tokenization or different model sizes).

Finally, we'll save the **best model checkpoint** (with highest val accuracy). This includes the model weights and the tokenizer (so we can seamlessly use it later for inference or deployment).

Realistic Challenges in Training: - It's possible some genres in XMIDI (like "song") are underrepresented or vaguely defined. The model might struggle if the training data for that class is inconsistent. We might then refine the genre definition or drop a problematic class if it's not learnable. But since the task explicitly asks for those six, we'll strive to handle it, perhaps by curating more samples of that style if needed. - Class imbalance: If one genre dominates (say pop or classical might have many more MIDI files), the model could become biased. We can address this by weighting the loss inversely to class frequency (e.g., give more weight to errors on minority classes). The Trainer allows specifying weights for classes in the loss function. We will compute the distribution of genres in the train set and apply weighting if the imbalance is large. - Convergence issues: Transformers can sometimes have unstable training initially (loss might not decrease if LR is too high). The warm-up helps. If still an issue, we might try gradient clipping (Trainer has `max_grad_norm`, often set to 1.0 for transformers) to prevent large gradient spikes.

By planning carefully and monitoring, we aim to have a robust training process that yields a well-generalizing genre classifier.

6. Demo Options

After training the model, we want to demonstrate its genre classification on new MIDI phrases. There are a few options for creating a user-facing demo, each with trade-offs:

A. Localhost Front-End: - We can build a simple local web or desktop interface that loads a MIDI file, runs the model, and displays the predicted genre. For instance, using a Python web framework like **Flask** or a UI library like **Gradio** or **Streamlit**. - A quick approach is to use **Gradio**: it can create an interface with a file upload component for MIDI, and a function that processes the file and returns the genre. This can run on `localhost` for personal testing. The interface might have an option to play the MIDI (for user reference) and show the top genre prediction (and maybe the confidence scores for each genre). - Since the user mentioned "optional help from Cursor," if a developer is using the *Cursor IDE* (which has AI assistance), it could expedite writing the UI code or integration, but it's not required. Essentially, one can use their development tools of choice to set up the local app. - **Local Demo Workflow:** The user selects or drags a

MIDI file of a short phrase into the app → the app uses the same tokenizer to convert it to tokens → the model's `predict` method yields logits → the app displays the genre with highest probability (and perhaps a probability bar chart for all genres). The response time should be quick (our model is lightweight, inference on a short sequence is under a second on CPU). - This option is great for development and private demos. The downside is it requires the environment (model, code) to be set up on the user's machine, which might not be trivial for non-technical users.

B. Hugging Face Space (Web Demo): - Hugging Face Spaces allow hosting small ML demos for free (for open-source). We can create a **Gradio app** or **Streamlit app** on a Space. This would make our genre classifier accessible via a browser to anyone. - We'd need to upload the model artifact (the trained weights and tokenizer) to the Hugging Face Hub (which is straightforward using `trainer.push_to_hub()` after training, for example). Then the Space code can load the model from the hub. - The interface would be similar to the local Gradio, but running on HF's infrastructure. A user could upload a MIDI file or possibly select from some embedded sample phrases, and the model would return the predicted genre. - We might include a few example MIDI clips (royalty-free or simple melodies) in the Space so that a user can click an example and immediately see the classification, to illustrate the model's usage. - **Challenges:** MIDI processing might require the environment on Spaces to have `miditok` or `pretty_midi` installed. We'd specify those in a `requirements.txt` for the Space. There's also the matter of file size/time – our model is not huge, and a short MIDI is just a few KB, so it should be fine. We do need to ensure the Space can handle MIDI binary file uploads (Gradio supports generic file uploads, so likely fine). - This approach makes the demo **public and easy to access**, which is ideal for showcasing the model's capabilities to a wider audience or stakeholders.

C. GitHub Repository & Kaggle Notebook: - We will create a GitHub repo (or add to the project's repo) containing: - The training code (so others can reproduce or fine-tune further). - The trained model files (or a link to download them, if file size is large, perhaps using GitHub LFS or huggingface hub). - A README with usage instructions. - Possibly a notebook (`demo.ipynb`) that loads the model and runs prediction on some sample inputs. - A **Kaggle Notebook** or **Google Colab** notebook can also be prepared as an interactive demo. This would walk through loading the model (from HF hub or from files), then allow the user to upload a MIDI or select an example, and output the genre. Kaggle even supports small audio playback or piano roll visualization if we wanted to be fancy, but at minimum it can show the classification result. - The advantage of a notebook is that it's interactive and reproducible – users can see the code. Colab/ Kaggle provide free GPU for running the model if needed (though for inference CPU is fine). - In the GitHub or notebook, we can include a few **sample MIDI phrases** (maybe one from each genre) for demonstration. For instance, a classical Bach phrase, a jazz blues lick, a funk bassline, a pop melody, a rock guitar riff, and a folk song melody. We'll show the model's predictions on these known examples. Ideally, the model should label them correctly – if not, that's an insight for future improvement.

D. (Optional) Integration with Music Software: - This is beyond the core request, but worth noting: one could integrate the classifier into a DAW (Digital Audio Workstation) environment or a MIDI player as a plugin that listens and labels genre in real-time. That would be a complex demo, but interesting. Since our focus is on the main deliverables, we note this only as a future possibility.

Realistic Considerations for Demo: - We must ensure the **tokenizer is identical** during demo as in training. That means saving the tokenizer config (if using miditok, save the vocab or the miditok `TokenizerConfig`). If using our own code, ensure the mapping of events to IDs is consistent. We will package this with the model (Hugging Face's `transformers` can save a tokenizer object). - Performance:

The model is small, so even on CPU, classifying a sequence of a few hundred tokens is fast (<0.1s). The overhead might be loading the MIDI and tokenizing – but that's also quite fast for short files. In a demo setting, this is near-instantaneous. - User Input Validation: If a user provides a very long MIDI or a format that's slightly off (e.g., Type 1 MIDI with many tracks, or unusual meta events), our demo code should handle it (perhaps by truncating long inputs, merging tracks, etc., similar to training). We should document that the classifier works best on short monophonic or simple polyphonic phrases, and not guaranteed on full-length songs (if someone uploads a full song, the model might attempt to classify just based on the beginning or could be overwhelmed if we limit length). - Explanation: In the demo interface or readme, include a note that this is an experimental model, and maybe a short explanation of how it works (for lay users, e.g. "This AI model reads the MIDI notes and rhythms and guesses the genre. It was trained on thousands of examples of each genre from the XMIDI dataset."). This sets correct expectations.

Finally, we will choose one or more of these demo approaches based on the audience: - For sharing with the community, the **Hugging Face Space** is likely the most convenient. - For internal demonstration or further development, the **local UI** and **notebooks** are useful. - We can even do both: have a Space for easy access, and a GitHub with code and a Colab link for those who want to delve deeper.

By following this plan, we'll implement a comprehensive genre classifier: from data preprocessing and model design to training and deployment. We remain mindful of challenges (data size, sequence encoding, model generalization) and address them with appropriate strategies (efficient tokenization, careful model sizing, robust training regimen, and thorough evaluation). This ensures a realistic path to a working solution that can classify short musical phrases into genres **classical, jazz, funk, pop, rock,** and **song**. With the model trained and a demo in place, we can analyze its mistakes and further refine it, but this plan provides a solid foundation for the project.

---

1 2 3 4 GitHub - xmusic-project/XMIDI_Dataset: XMIDI Dataset: A large-scale symbolic music dataset with emotion and genre labels.
https://github.com/xmusic-project/XMIDI_Dataset

5 XMusic
https://xmusic-project.github.io/

6 7 8 9 11 14 15 Tokenizations - MidiTok's docs
https://miditok.readthedocs.io/en/latest/tokenizations.html

10 12 13 17 MusicBERT: Pre-Trained Model For Symbolic Music Understanding
https://analyticsindiamag.com/global-tech/musicbert-microsofts-large-scale-pre-trained-model-for-symbolic-music-understanding/

16 Distilbert: A Smaller, Faster, and Distilled BERT - Zilliz Learn
https://zilliz.com/learn/distilbert-distilled-version-of-bert