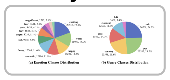
Symbolic Music Genre Classifier: Key Design Decisions

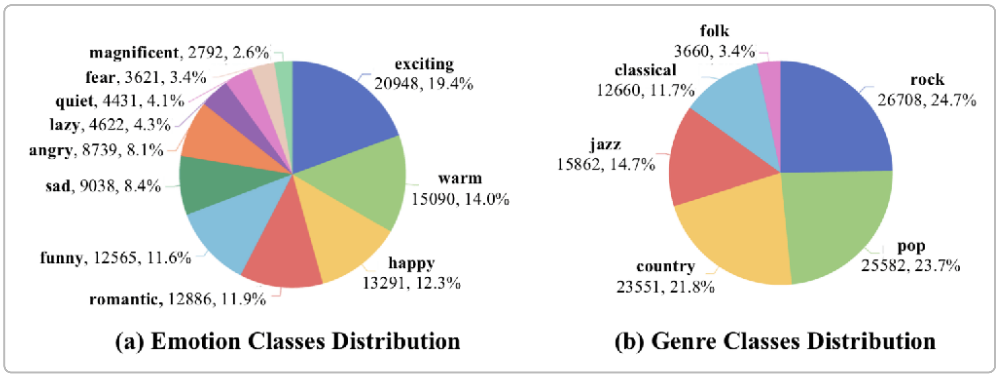
Dataset Selection and Comparison

Choosing the right MIDI dataset is crucial for training a genre classifier on short 2–4 bar phrases. It needs to cover the six target genres (classical, jazz, funk, pop, rock, “song”) with sufficient labeled examples. Below is a comparison of the XMIDI dataset versus other prominent symbolic music datasets, considering size, genre coverage, label quality, licensing, and preprocessing effort:

Dataset	Size (MIDIs / hours)	Genre Coverage	Genre Labels?	Licensing & Quality	Preprocessing Notes
XMIDI	108,023 files (~5,278 hours) [1]	**Broad:** ~6 main genres (e.g. classical, jazz, pop, rock, country, folk)  . Each file annotated with one of these genres (plus emotion labels)	**Yes** – precise genre labels for each piece [1]	Mixed-quality MIDIs (web-sourced) but labels are carefully assigned. Largest labeled symbolic dataset. Licensing not explicitly stated (research use assumed).	Must parse multi-track MIDIs; genre label provided per file. May need to filter out any low-quality transcriptions.

Dataset	Size (MIDIs / hours)	Genre Coverage	Genre Labels?	Licensing & Quality	Preprocessing Notes
Lakh MIDI	176,581 files (45k aligned to songs) [2] [3]	**Broad:** Mainly pop/rock and various popular music (matched to Million Song Dataset). Little classical or jazz. Funk present in pop/rock era music (not a separate label).	**Partial** – No direct labels in MIDI; 45k tracks can inherit **audio** metadata from Million Song Dataset (e.g. genre tags) [2]. Quality of genre tags varies.	Open license (CC-BY) for the collection [4]. Contains some duplicates or noisy MIDIs [3]. Many files are user transcriptions of copyrighted songs.	Non-uniform genre labels: requires linking to external metadata or manual tagging. Must remove corrupt files [3]. Suitable for supplementing underrepresented genres (e.g. funk) if labels can be obtained.
MAESTRO	~1,200 MIDI files (~200 hours) [5]	**Narrow:** All classical (virtuosic piano performances) [5]. No other genres.	**Yes** – implicitly classical (composer metadata included) [5].	High-quality, precisely aligned MIDI/audio, CC-BY-NC license. All pieces are classical repertoire (17th–20th c.) [6].	Minimal preprocessing (already cleaned and aligned). Not useful beyond classical genre. Could provide fine classical examples to augment XMIDI's classical subset.
GiantMIDI-Piano	10,855 MIDI files (~34 million notes) [7]	**Narrow:** Classical piano only (transcribed from live recordings) [7].	**Yes** – implicitly classical (pieces by 2,786 composers) [7].	High-quality automatic transcriptions of public-domain classical pieces. Research use permitted via disclaimer.	Already in MIDI; mainly single-instrument. Useful only for classical genre enrichment.

Dataset	Size (MIDIs / hours)	Genre Coverage	Genre Labels?	Licensing & Quality	Preprocessing Notes
MusicNet	330 MIDI files (330 recordings) [8]	**Narrow:** Classical (quartets, solo piano, etc.).	**Yes** – implicitly classical. Also has note-level annotations [8].	Freely licensed classical pieces with detailed annotations. Very small dataset.	No significant preprocessing needed. Too small for training; more for evaluation or feature analysis.
NES-MDB	5,278 songs (NES game music) [9]	**Niche:** 8-bit game music (chiptune). Styles vary (marches, pop-ish tunes, etc.) but not categorized by standard genres.	**No** – Labeled by game/composer, not by genre [9].	Public dataset (research use) of multi-instrument NES synth tracks [9]. Distinctive sound (4-channel limitations).	Provided in MIDI-like format with additional expressive info [10]. Unaligned with target genres (could treat "game music" as its own category). Not directly useful for classical/jazz/pop/rock labels.



(a) Emotion class distribution and (b) Genre class distribution in the XMIDI dataset. XMIDI's genre labels are dominated by Rock and Pop (~24% each), with substantial Country (21.8%), and smaller proportions of Jazz (14.7%), Classical (11.7%), and Folk (3.4%). (Note: "Funk" is not explicitly labeled in XMIDI; and "Song" may correspond to folk or other vocal music.)

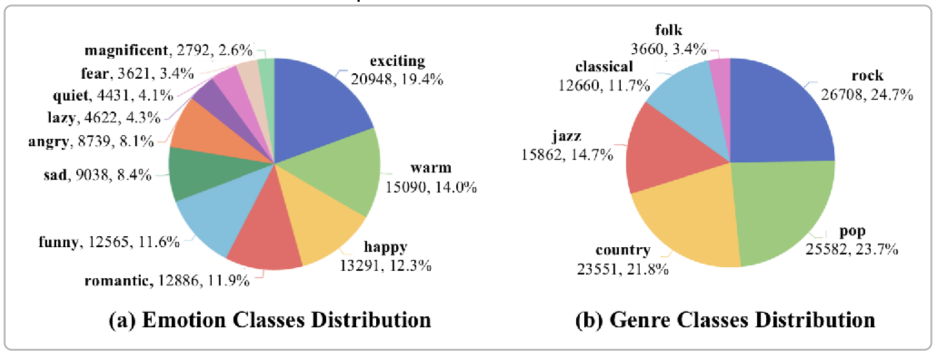
Pros and Cons of Each Dataset:

- **XMIDI: Pros:** Comprehensive genre coverage with **explicit labels** for each MIDI (including classical, jazz, pop, rock, etc.) [1]. Massive scale (100k+ pieces) provides ample training data, even when slicing into short phrases. Genres are annotated with high fidelity (the dataset was specifically curated for genre and emotion classification) [1]. **Cons:** Does not include a "funk" label – funk pieces may be missing or folded into other categories. Also, it includes many user-contributed MIDIs;

quality can be inconsistent (some transcriptions may be noisy or simplified). Licensing is not clearly documented (likely a mix of public-domain and unofficial transcriptions), so models trained on it should be used for research purposes. Overall preprocessing is moderate – combining multi-track songs and slicing into 2–4 bar segments is necessary, but the presence of bar/beat info in MIDI and the given labels simplify this task.

- **Lakh MIDI: Pros:** Very large and diverse collection, especially strong in **popular genres (pop, rock, hip-hop, etc.)** from late 20th-century music. Likely contains many **funk** tracks (e.g. 1970s–80s pop/rock categories) and broad "song" varieties. Openly licensed compilation [4]. **Cons:** No built-in genre labels – one must rely on the subset matched to the Million Song Dataset for genre metadata [11]. Those tags can be noisy or broad (e.g. a song might be labeled "Rock/Pop" rather than a single genre). Preprocessing is higher effort: filtering out corrupted files [3], and mapping each MIDI to a genre tag via the Million Song Dataset's metadata (which may require resolving inconsistent genre taxonomies). Lakh is excellent for *additional* coverage (e.g. mining funk songs or augmenting pop/rock), but by itself it's not a ready-to-use labeled dataset.

- **MAESTRO / GiantMIDI / MusicNet: Pros:** These offer **high-quality classical** music in MIDI (with precise timing and dynamics). Minimal cleaning needed. Licensing is clear (MAESTRO is CC-BY-NC, MusicNet is free/open). **Cons:** *Only classical genre.* They lack genre variety, so they cannot cover the full classification task. Including one of these can enrich the classical class (for example, XMIDI's classical subset is moderate in size ~12k pieces



, so adding more classical data might help). However, for non-classical genres these datasets add no value.

- **NES-MDB: Pros:** Interesting dataset with multi-instrument compositions, which could introduce unique textures/rhythms not found elsewhere (8-bit game music often blends classical, pop, and folk influences). Could be used if one wanted a "game" or "chiptune" genre. **Cons:** Does not map to standard genres like jazz or funk – the style is constrained by the NES sound chip. Also, its labeling is by game/composer, not by genre, so it's not directly usable for our genre classes. Likely not useful unless the project scope expands to recognize chiptune as a genre.

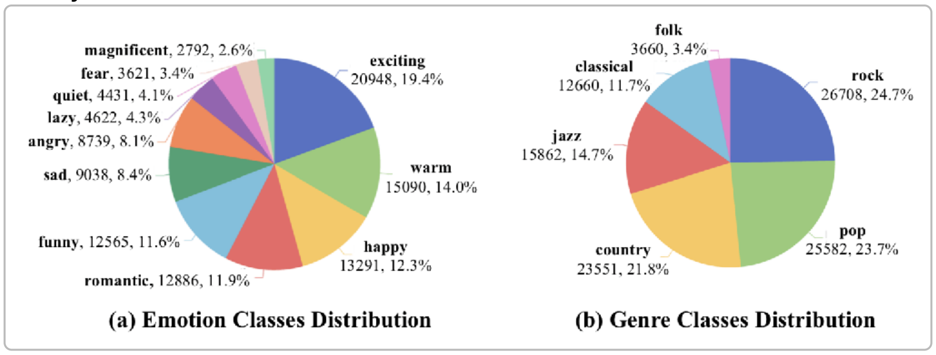
Dataset Suitability Ranking: (for a 6-genre classifier on short phrases)

1. **XMIDI – Best Choice:** It directly provides a large number of **multi-genre labeled** MIDI examples, covering *classical, jazz, pop, rock,* etc., which aligns with the target classes [1]. It offers the best balance of size and label quality. With XMIDI, one can train from scratch without external labeling steps. *Downside:* no explicit funk label, but one can attempt to identify funk pieces within XMIDI's

pop/rock category or supplement from elsewhere. Overall, XMIDI's scale and labels make it the top pick for genre classification.

2. **Lakh MIDI (with MSD labels) – Runner-up:** In the absence of XMIDI, Lakh could be leveraged by transferring **Million Song Dataset** genre tags to the 45k matched MIDIs. This provides a large pool of pop/rock (and some jazz/classical) pieces with genre annotations (albeit imperfect) [2] . Lakh is especially useful to fill gaps in XMIDI: e.g., sourcing **funk** tracks or additional "song" (vocal pop) examples. However, the **cons** are the overhead of merging metadata and the unreliability of genre labels (e.g., many songs have multiple or ambiguous genres). It ranks below XMIDI due to the extra data-cleaning and weaker labeling.

3. **Hybrid Approach – Augment Specific Genres:** A combination of datasets can be considered for niche needs. For example, if **"song"** is intended as a category for folk/country vocal music, XMIDI's folk and country entries (total ~9% of XMIDI



(a) Emotion Classes Distribution (b) Genre Classes Distribution

) could be augmented with folk song MIDIs from other sources (e.g., the **Nottingham folk song dataset or Essen folk database** in MIDI format). Similarly, for **funk**, one might compile a small custom MIDI set of funk songs or use Lakh to find tracks by funk artists. This hybrid approach ensures every target genre has sufficient representation. It's more labor-intensive (hence ranked third), but can be worthwhile if XMIDI lacks a category entirely.

4. **Classical-only Datasets – Low Priority:** MAESTRO, GiantMIDI, MusicNet are high-quality for classical music but bring nothing for other genres. They could improve the classical classifier a bit (with very accurate piano renditions), but XMIDI already has a sizable classical subset. Given a moderate compute budget, focusing on multi-genre data is more impactful than slightly boosting one genre. These single-genre sets rank low for the overall goal.

5. **NES-MDB – Not Recommended:** It does not align with the genre labels needed (its music doesn't cleanly fall into classical/jazz/pop/etc.). Incorporating it could confuse the model unless a "game/chiptune" genre is explicitly included, which it isn't. Thus, NES-MDB is not suitable for this classifier.

Final Dataset Decision: Use **XMIDI** as the primary training corpus for all six genres, as it offers the best coverage and ready-made labels. To handle the **"funk"** genre (not explicitly in XMIDI), plan for a small supplementary set – e.g. extracting funk-style songs from Lakh MIDI or other sources – and label them appropriately. XMIDI's large size and rich labels ensure robust genre modeling, while targeted augmentation will fill any remaining genre gaps (ensuring *all* six target genres have enough training data). This strategy balances performance (leveraging a huge labeled dataset) with practicality (minimal manual labeling, since XMIDI is pre-labeled).

MIDI Tokenization Scheme

Once the dataset is chosen, the next design choice is how to encode MIDI files as token sequences for the transformer. A good MIDI tokenizer will represent musical events in a way that captures important information (pitch, timing, instrument, etc.) while keeping sequence lengths manageable. We compare four major tokenization schemes – **REMI**, **Compound Word (CP)**, **Octuple**, **MuMIDI** – and discuss simpler alternatives. Key factors include vocabulary size, sequence length, interpretability, multi-track support, and ease of implementation:

- **REMI (Revamped MIDI)** – *Separate tokens for each event attribute.* Introduced in the Pop Music Transformer [12], REMI represents a note as a sequence: *Bar token* (new measure), *Position token* (beat within bar), then *Pitch, Velocity,* and *Duration* tokens for the note [12]. Time is advanced via Bar/Position tokens instead of explicit time deltas. **Pros:** Human-readable and intuitive – e.g. you can literally see a melody as a series of pitch and duration tokens with bar/beat markers. Supports expressive performance (velocity, etc.) and gives a model explicit rhythmic structure (since each bar and beat position is marked) [12]. It's relatively simple to implement by reading the MIDI events in order and emitting the appropriate tokens. **Cons:** Sequence length is **long** – every single note turns into 3+ tokens (pitch, velocity, duration, plus positioning tokens). For polyphonic music, simultaneous notes are handled sequentially (often repeating the same position token), which further bloats length. For a short 4-bar phrase this might result in a few hundred tokens, which is within 512, but still relatively large. The vocabulary is moderate (e.g. 128 pitch values, maybe 32 velocity bins, a set of duration bins, plus position indices – on the order of a few hundred tokens total). Multi-track support is not native in original REMI; it's meant for single-track. However, **REMI+** extends it by adding an *Instrument Program* token before each note's pitch to indicate which instrument/track it belongs to [13]. This allows encoding multi-instrument music in one sequence. **Bottom line:** REMI is **straightforward and expressive**, but yields longer sequences. A transformer encoder can certainly handle REMI sequences for 2–4 bar phrases, but it will do a lot of redundant work attending over many tokens that represent one chord or simultaneous event.

- **Compound Word (CP)** – *Pack multiple attributes into one "super-token."* Proposed by Hsiao et al. (Compound Word Transformer) [14], CP is conceptually similar to REMI in what it represents, but it **reduces sequence length** by combining a note's tokens. Pitch, velocity, and duration (and optional attributes like instrument program or chord) are first embedded separately and then *pooled into a single token representation* [14] [15]. In other words, at one time-step the model sees a compound token that contains all note information. For example, a CP token might internally comprise: `{Position=3rd beat, Pitch=C4, Velocity=forte, Duration=quarter, Instrument=Piano}` all in one. **Pros:** This dramatically **shortens the sequence** – each note (or chord) becomes one token in the sequence instead of 3–5. For polyphonic music, if notes start at the same time, they might either be combined as a chord token or as separate tokens each with the same position (depending on implementation). The vocabulary here is not a simple list of all combinations (which would be enormous); instead, the model uses multiple embedding matrices (for pitch, velocity, etc.) and concatenates or sums them to form the token embedding [14]. This means the *effective* vocab is the product of attributes, but it's factorized so it doesn't blow up memory. **Interpretability** is lower than REMI – a single token ID doesn't directly tell you all components without decoding – but since the attributes are still accessible via the embedding, one can interpret if needed by examining those. **Cons:** Implementing compound tokens is more complex. Training also becomes multi-task if used in an *autoregressive generation* setting – the model output must predict multiple attributes at once (often handled by multiple softmax heads for pitch, velocity, etc.) [16]. This makes generation non-trivial (one must sample consistent combinations). However, **for an encoder used in classification, we don't actually need to generate tokens**, so we can avoid the multi-head output problem. We can simply use the compound embedding as input representation and then do classification on the encoder's output. That greatly mitigates the complexity: essentially it becomes an embedding design choice (we would create an embedding for each attribute and sum them to get the token embedding for each note event). Another consideration: CP (and Octuple) still rely on the presence of Bar/Position tokens to mark time [17] – typically each compound token includes a position index or bar index field. So the model still knows the rhythmic location of notes. In short, CP encoding **is well-suited to encoder models** (they can learn from the compressed sequence and shared attribute embeddings) and yields shorter sequences, at the cost of a more involved tokenization pipeline.

- **Octuple (MusicBERT Encoding)** – *Extreme compression: represent each note as an 8-dimensional token.* Octuple was introduced by Zeng et al. for MusicBERT [15]. It's very similar in spirit to Compound Word: multiple attributes are combined into one step. Specifically, an Octuple token contains up to 8 fields: e.g. Pitch, Position, Bar, Velocity, Duration, Program, Tempo, Time Signature [18]. In practice, not all eight are always used for each token (some are optional as needed) [19]. For example, a note-on event token might be composed of (Pitch, Position, Bar, Velocity, Duration, Program) in a polyphonic piece [18]. The implementation uses separate embedding matrices for each of these 8 attribute types and then concatenates or pools them into one vector per token [15]. **Pros:** This method **yields very compact sequences** – every musical event (note or possibly chord) is one token. It naturally handles multi-track music by including Program (instrument) as one attribute, and it handles timing with Bar and Position attributes. The sequence length reduction is significant; for example, a chord of three notes starting together could be represented either as one token (if treated as a single chord event) or as three tokens (one per note) all sharing the same position – either way, it's far fewer than the ~9 tokens REMI would use for three notes. Microsoft's MusicBERT demonstrated that this encoding, combined with bar-level masking pretraining, led to strong performance on music understanding tasks [20] [21]. For genre classification of short phrases, Octuple encoding should capture all necessary info (what pitches and instruments are present, their timing) in a very concise form. **Cons:** Like CP, Octuple encoding is more complex to implement from scratch. One either needs to use an existing tokenizer library (e.g. MidiTok supports Octuple pooling) or write custom code to produce these embeddings. If one wanted to pretrain the model with mask prediction, the masking has to respect the compound structure (e.g. mask entire tokens or specific fields). Another consideration: because Octuple compresses so much, the model might need to be a bit larger or better trained to fully exploit the combined information (each token's embedding is dense with info). But given our moderate sequence lengths, this is not a big issue. Overall, **Octuple can be seen as a specific case of compound word encoding optimized for music** – it's essentially the recommended approach in recent literature (since MusicBERT used it successfully for genre classification and other tasks) [20] [21].

- **MuMIDI (Multitrack MIDI)** – *Serialized multi-track events with track tokens.* MuMIDI, introduced in the PopMAG paper (Ren et al. 2020), takes a slightly different approach to multi-instrument music [22]. The **key idea** is to represent all tracks in a single sequence **without merging every attribute into one token**. Instead, MuMIDI uses special *Track tokens* to indicate instrument changes and groups note attributes for each note under that track context [23]. For example, at a given time position, a MuMIDI sequence might have: `[Position][Track=Drums][Pitch=C1][Duration=eighth]`

`[Track=Piano][Pitch=G3][Duration=quarter]` to denote that at that beat, a drum note and a piano note occur. It also includes bar and beat positional tokens like REMI, and can include *Chord tokens* to mark chord structures [22]. **Pros:** MuMIDI preserves the **polyphonic structure explicitly in the sequence**. All events that happen together are serialized together following a single position token, with track tags differentiating instruments. This avoids needing separate sequences per track and ensures the model knows which instrument plays what. It's more interpretable than CP/Octuple in that you still see individual events, and it's fully autoregressive-friendly (the original motivation was generation). For classification, MuMIDI provides a rich representation that a transformer can read to infer genre (since genre is often indicated by instrument roles – MuMIDI makes it clear e.g. that there is a "Drum track" present). **Cons:** Sequence length is still relatively long – while it removes redundant time tokens (multiple notes share one position marker), you still end up with multiple tokens per note (Track token + Pitch + Duration per note). It doesn't compress notes into single embeddings, so it sits between REMI and CP in terms of length. Implementation is a bit involved (one must sort events by time and interleave track tokens correctly). Additionally, MuMIDI has a defined vocabulary for track names or program numbers that must be managed. In summary, MuMIDI is good for maintaining multi-track structure, but it doesn't reduce token count as aggressively as CP/Octuple. For an encoder-only classifier, it may be less appealing than the more compressed encodings, because we are not as constrained by autoregressive requirements.

- **Simplified/Other Schemes:** A classical approach is **MIDI-like (event-based)** encoding – where you have *Note-On, Note-Off,* and *Time-Shift* tokens in chronological order (as used in early music RNNs). This is straightforward (one token per MIDI event), but it's actually less efficient than REMI for polyphonic music (because every note-on and note-off is a separate token, and no high-level timing structure is given). It tends to produce very long sequences and doesn't explicitly mark bars or beats. **Time-step grids** or **piano roll** representations (where you quantize time into fixed steps and use a token or vector for each step) have also been used, but those either blow up input length (if using one step per 16th note, 4 bars = 64 steps, which is fine; but representing chords at one step might require multiple tokens or a multi-hot representation which standard transformers can't directly ingest without modification). There are also research tokenizations like **REMI+Chord** (which adds chord-symbol tokens at harmonic changes) or **structured representations** (that group notes into chord tokens explicitly). Such additions can aid a model's understanding (e.g. chord tokens might help identify jazz vs classical via harmonic complexity), but they require external chord annotation or detection. Given the complexity, these are generally not required for genre classification – the transformer should be able to infer genre from raw note patterns without explicit chord labels. Finally, one could consider **Byte Pair Encoding (BPE) on MIDI events** [24] – i.e. learning recurring sequences of tokens (like common chord patterns) as single tokens. This can optimize the vocabulary and sequence length in a data-driven way. However, applying BPE would need a large corpus; XMIDI might suffice, but the benefit for classification is uncertain. (A study found that BPE on REMI didn't help much for music generation, partly because REMI already has structured tokens [24].)

To summarize the trade-offs in a comparative snapshot:

Tokenization	Sequence Length	Vocab Size	Multi-track Support	Interpretability	Ease of Implementation
REMI	Long (each note = 3–5 tokens; chords = multiple tokens each)	Few hundred (pitch, vel, dur, etc.) [12]	Single-track by default (multi-track via adding instrument tokens) [13]	High (clear step-by-step events)	Easy-Moderate (straightforward logic; must handle time->bar/position conversion)
Compound Word (CP)	Shorter (each note = 1 token after pooling)	Factorized (each attribute has its own vocab; combined via model) [14]	Yes (can include instrument as part of token) [25]	Medium (token is a bundle; needs decoding to interpret)	Moderate (needs custom embedding pooling; use existing toolkit or implement multi-embedding)
Octuple	Short (each note = 1 token with 6–8 fields)	Factorized (8 sub-vocabs, e.g. 128 pitches, 32 positions, etc.) [18]	Yes (explicit Program field for instrument) [19]	Medium (similar to CP – inspect fields for meaning)	Moderate (available in libraries like MidiTok; otherwise implement like CP with fixed 8-field structure)
MuMIDI	Medium (each note = ~2–3 tokens: track + note attrs)	Several hundred (includes track tokens, pitch, dur, etc.) [22]	Yes (designed for multi-track; uses track tokens) [23]	Fairly High (sequence explicitly lists each track's notes)	Harder (must interleave track tokens properly; less standard)
MIDI-Like (naive)	Long (note-on + note-off + time shifts for every note)	~130 (note numbers + special tokens)	Yes (notes carry channel info, but no explicit separation)	Low-Medium (no bar structure, raw event list)	Easy (just emit events in MIDI order; but requires adding note-off tokens, which increases length)

Recommendation – Ranking Tokenization Options for this Project: Considering we have **short phrases (2–4 bars)** and a moderate compute budget, we want a scheme that is **practical to implement yet performant**. We balance sequence length vs. complexity:

1. **Octuple Encoding – First Choice:** This approach offers an excellent balance for an encoder-based classifier. It will drastically reduce sequence length (making the transformer's job easier) while

retaining all critical information (pitch, timing, instrument) in each token's embedding. Crucially, it is **well-proven in genre classification tasks** – MusicBERT's success on symbolic genre classification is largely attributed to Octuple encoding plus pretraining [20] [21]. For our scenario, even without massive pretraining, using Octuple tokens means the model can see a whole 4-bar phrase in perhaps <50 tokens (instead of a few hundred in REMI). This compactness can improve learning efficiency and possibly generalization (the model focuses on higher-level patterns like "instrument X playing syncopated rhythm" which might be easier to detect when all those attributes co-occur in one token). **Implementation:** We can leverage existing libraries (e.g. MidiTok) to convert MIDI to Octuple tokens, or implement a custom dataloader that creates embeddings for each attribute. Since we are open to training from scratch, we have full control to design the embedding layers accordingly. Given the moderate complexity and high upside (performance and alignment with state-of-the-art methods), Octuple is the top recommendation.

2. **Compound Word Encoding – Second Choice:** CP is conceptually similar to Octuple and would also serve well, especially if we simplify to only include the needed attributes (perhaps 4 fields: position, pitch, duration, instrument). It also yields short sequences and has been used in prior music generation work. It ranks slightly behind Octuple simply because Octuple is essentially a standardized, specific compound schema that has seen more usage in recent models; CP is a bit more generic concept. However, if one encounters implementation difficulties with Octuple, a CP approach (e.g. pooling just pitch+duration+instrument at each position) would be a solid fallback. The performance and model considerations are nearly the same as Octuple's. One minor note: if we were planning to **generate** music, CP/Octuple complicate that – but since our task is classification, we can ignore generation issues and treat the compound tokens as fixed inputs.

3. **REMI (with Instrument Tokens) – Third Choice:** If simplicity and transparency are paramount, REMI is a viable option that will certainly work for classification, albeit with some efficiency cost. It's easier to debug (you can print the token sequence and verify it aligns with musical intuition, which is helpful during development). On short phrases, the sequence lengths are not unmanageably large (often well under 300 tokens for 4 bars). And because we're not exceeding 512 tokens, a standard transformer can handle the length without needing special memory optimizations. The model might have to learn patterns across multiple tokens (e.g. that a "Kick drum" event and a "Snare" event under the same Position token implies a rock drum pattern – whereas in Octuple the model might see a combined representation). But transformers are good at picking up such patterns with enough data. **Pros of REMI for us:** straightforward coding, no need for custom embeddings beyond a simple token-index mapping. **Cons:** longer training times per sample and possibly needing a slightly larger model to capture the same info (since relationships like pitch-to-instrument are learned through attention over separate tokens). Given our "moderate" compute constraint, REMI is acceptable but not as efficient as the above options, hence the lower ranking.

4. **MuMIDI – Fourth:** While MuMIDI elegantly handles multi-track music, its advantage is more pronounced for generation tasks or scenarios where maintaining the exact track order is crucial. For pure genre classification, MuMIDI's benefit (explicit track tokens) can also be achieved in other ways (Octuple/CP with instrument attribute implicitly tells the model the instrument). MuMIDI sequences will be longer than CP/Octuple, and implementation is less common/standard. One might choose MuMIDI if they want to stick closer to a sequential event representation (like REMI) but with multi-track built-in. However, it doesn't offer clear advantages for a short phrase classifier – the

transformer doesn't *need* to know the exact ordering of events within the same beat, only which events occurred. Thus MuMIDI, while feasible, is not the most efficient or simple choice here.

5. **Others (MIDI-like or custom simplifications) – Last:** Simpler event-based encodings or piano-roll grids are not recommended because they either explode sequence length or lose important information. For instance, ignoring velocity or using a coarse time step grid could remove potentially genre-indicative features (velocity dynamics can distinguish classical from MIDI-fied pop, swing timing vs straight timing could be lost with coarse grids, etc.). The above schemes (REMI, CP, Octuple) all preserve essential musical details and have been tested in literature. Unless we had severe memory constraints (which we do not, for ≤512 tokens), there's no need to resort to cruder encodings. A partial simplification that is reasonable would be to limit the vocabulary for certain attributes (e.g., round all velocities to a few levels, or quantize duration finer vs coarser). But these tweaks can be applied within any of the recommended schemes if needed. Overall, sticking to a **proven tokenization** ensures our classifier has access to rich musical information.

Final Tokenizer Decision: Adopt the **Octuple encoding** (or an equivalent compound token representation) for the MIDI phrases. This choice yields short, information-dense sequences that an encoder model can learn from efficiently. In practice, we will treat each 2–4 bar phrase as a sequence of tokens where each token corresponds to a musical event (note or rest) with fields for timing, pitch, duration, and instrument. This will let our transformer focus on the *musical content* (what notes/instruments occur and when) without wading through repetitive timing tokens. It strikes the right balance between performance and practicality: although a bit more complex to set up than naive encodings, it will likely improve classification accuracy (by making genre-distinctive patterns more salient) and keep compute requirements moderate (shorter sequences mean faster training and less need for extremely deep models). We also maintain transparency to a reasonable degree – we can decode any token to understand what musical event it represents – thereby retaining insight into what the model sees.

Model Architecture Choice

Finally, we must decide on the model architecture: an encoder-only transformer is chosen (since we only need to output a genre label for an input sequence). The question is which encoder architecture best balances classification performance with computational efficiency and ease of use. We compare a standard compact Transformer (like DistilBERT or a small BERT) with specialty models and scaling techniques: **DistilBERT**, **MusicBERT**, **Performer**, **Linformer**, **Longformer**, and a **custom small BERT from scratch**. Key considerations are model size, training efficiency, need for pretraining, compatibility with our MIDI tokens, and handling input lengths up to ~512 tokens.

- **DistilBERT (small BERT)** – This is a 6-layer Transformer model distilled from BERT-base, originally for NLP. It has about 40% of the parameters of BERT-base and runs ~60% faster, while retaining ~95–97% of BERT's performance on language tasks [26]. **Pros:** It is compact (around 66 million params vs 110M in BERT-base) and thus faster to train and less prone to overfitting on moderate data. Using a DistilBERT-like architecture for music is entirely viable: we would initialize a 6-layer Transformer encoder with perhaps 512 hidden size and 8 attention heads (DistilBERT's configuration) and train it from scratch on our tokenized MIDI data. It **does not require any pretraining** if we train supervised, although the name "DistilBERT" implies a distilled model, we can simply use the same architecture without actual distillation. The benefit is that it's a proven efficient architecture – HuggingFace's studies showed it preserves most of the model's capability with much less compute

[26]. For genre classification, which is a simpler task than language understanding, a 6-layer transformer should be quite sufficient, especially given the size of our dataset (hundreds of thousands of phrase examples). **Cons:** If we literally use the pre-trained DistilBERT weights (trained on English text) and try to finetune on MIDI, that would be ineffective – the token semantics are completely different. So we will be training from scratch; we won't actually benefit from the "distillation knowledge" of BERT (unless we ourselves tried to distill from a larger music model, which is likely overkill). This means our small model might not reach the absolute peak accuracy a larger model could, but with enough training data it should get close. Another consideration: DistilBERT doesn't include token-type embeddings or other bells and whistles – which is fine, as our task doesn't require segment distinctions. It uses absolute positional encoding of length up to 512, which is perfect for our input size. In summary, a **small 6-layer Transformer** (inspired by DistilBERT) trained from scratch is very cost-effective and likely to meet the performance needs, making it a strong candidate.

- **MusicBERT (pre-trained)** – MusicBERT is a 12-layer BERT-like model trained on over 1 million MIDI songs by Microsoft [20]. It uses the Octuple encoding and a special pretraining regime (bar-level masked language modeling) to learn musical representations. It achieved state-of-the-art results on genre classification and other music understanding tasks [21]. **Pros:** If we use a pre-trained MusicBERT model, we would be leveraging a **massive amount of musical knowledge**. We could likely fine-tune it on our specific six-genre classification task with relatively few epochs and get excellent accuracy. It already knows, for instance, typical differences between genres (since it was pre-trained in a self-supervised way to understand structure and style). So in terms of raw performance, MusicBERT or a similar pre-trained transformer is arguably the top. Also, MusicBERT's architecture is designed for music (it incorporates the Octuple tokenizer and even does some relative positioning by bars). If using MusicBERT, it would naturally fit with our choice of Octuple encoding. **Cons:** The biggest downside is **dependency on massive pretraining**. Using MusicBERT means our project's success leans on a model that was trained by others on a very large proprietary dataset ("Million MIDI Dataset" which includes Lakh, MAESTRO, and more). This can conflict with the desire for transparency and control – we'd be importing a black-box that has learned from who-knows-what sources. If the user's goal is to have a model they fully understand and can retrain, MusicBERT is less appealing. Additionally, MusicBERT-base is roughly BERT-base in size (12 layers, ~110M params). Fine-tuning it is heavier than training a small model from scratch, though still feasible on a single GPU. Another factor: Not all pre-trained models are publicly available; assuming we have access to MusicBERT weights, fine-tuning requires adjusting the input embedding if our vocab differs (but if we follow Octuple exactly, it could match). If we train **from scratch a MusicBERT-like model**, that defeats the point of pretraining and would be essentially training a large 12-layer transformer on our data. We likely *don't need* that many layers given our dataset size (which, while large, is not millions of distinct songs). Training a 12-layer model from scratch is double the computation of a 6-layer, and might yield only marginally better accuracy for classification (especially if data is plentiful and classes are broad). So, while **MusicBERT (pretrained)** would maximize performance, it is somewhat against the project constraints (it's a "massive pretraining" approach). It ranks high in accuracy but lower in practicality.

- **Performer (linear attention)** – The Performer is an efficient transformer variant that uses a linear-time attention mechanism called FAVOR+ (Fast Attention via Orthogonal Random features) [27]. It approximates the full self-attention by mapping queries and keys into a random feature space so that dot products in that space correspond to softmax attention in expectation [28]. The key benefit is that **attention computation becomes O(n)** instead of $O(n^2)$ in sequence length. **Pros:** If we anticipated *longer sequences* or wanted to scale up the model dimension or batch size significantly, Performer could save memory and compute. For example, if in the future we decided to classify whole songs or 1000-token sequences, a Performer-based model could handle that where a normal transformer might run out of memory. It's also nice from a research perspective, as it keeps model size independent of sequence length. **Cons:** In our scenario (≤512 tokens, moderate model size), the standard transformer is not actually a bottleneck. 512-length self-attention is quite manageable on modern hardware, especially with a small model. Using Performer might *reduce training efficiency* at small scales, because the approximation overhead (random feature mapping) and additional normalization considerations can make it slower than standard attention for short sequences. Additionally, Performer is a bit more complex to implement or require specialized libraries – it's not as plug-and-play as a vanilla Transformer in many frameworks. In terms of **performance**, Performer's approximation can in theory be very close to exact attention [29], but there is always a tiny risk of degraded accuracy if not enough random features are used. For genre classification, this likely wouldn't matter, but it's another uncertainty. Given that our input lengths are modest, the **advantage of Performer is minimal**, so it isn't the top choice for now. It's ranked as an interesting option if we needed to greatly extend sequence length in the future or run on very memory-limited hardware.

- **Linformer (low-rank attention)** – Linformer is another efficient attention model that projects the sequence into a lower-dimensional space before computing attention, effectively reducing complexity to O(n) as well [30]. It relies on the idea that the attention matrix has low rank and can be approximated by smaller matrices [30]. **Pros:** Like Performer, Linformer targets long sequences – it can handle thousands of tokens by trading off some precision in attention. It's conceptually simpler than Performer (uses learned projections rather than random features). If we wanted to use a larger input (say 1024 tokens) or a larger batch, Linformer could allow that within the same memory. **Cons:** For 2–4 bar phrases, Linformer's benefits won't be realized. In fact, using a low-rank approximation for only 100–300 tokens might slightly hurt performance with no need. Also, implementing Linformer requires customizing the transformer or using a third-party implementation. The original Linformer paper showed that with a projection dimension of ~k=128, you can handle sequences of length 512–1024 well [31] – but if our sequence is already ~300, we could just use full attention. Additionally, the "low rank" assumption might not hold as well for music sequences as it does for language, especially short ones (though evidence suggests it works in general) [32]. So, similar to Performer, **Linformer is not necessary for our current needs**. It adds complexity for little gain, thus ranking low in priority.

- **Longformer (local+global attention)** – Longformer is a model designed for very long documents (thousands of tokens) by using a **sliding window local attention** pattern and a few **global tokens** that attend broadly [33]. For example, each token might attend to 128 tokens on either side (local context) and certain designated tokens (like CLS) have global receptive field [33]. **Pros:** If we ever wanted to classify a much longer sequence (e.g. a full song, or 100+ bars), Longformer would let us scale to say 2,000 or 4,000 token sequences by limiting the attention scope. It's optimized for long inputs and has shown good results in long-text tasks. **Cons:** For our short phrases, Longformer is actually a disadvantage. If you set a window size of 128 on a sequence of 200 tokens, essentially it can see the whole sequence with a couple hops, but you've introduced a constraint that isn't needed. A normal transformer already has global attention for all tokens, which is optimal for capturing musical motifs that might span the entire 4 bars. Using local attention could potentially make it harder to capture relationships between notes far apart (e.g. a motif in bar 1 and its repetition in bar 4 might exceed a small window). We could configure the Longformer with a window equal to sequence length to emulate full attention, but then it's just a normal transformer with overhead. Also, the Longformer architecture (as implemented in HuggingFace, etc.) has more hyperparameters (window size, which tokens are global) that need tuning – unnecessary for this task. In summary, Longformer shines for extremely long sequences, which we don't have; it would only add unneeded complexity and possibly hurt performance on short inputs. Therefore, it's not recommended here (lowest priority).

- **Custom BERT-like from scratch (choice of size)** – This essentially covers picking a transformer architecture (depth, width) and training it on our data without any pretraining. We've implicitly discussed two instances of this: a smaller 6-layer (DistilBERT style) and a larger 12-layer (BERT-base style or MusicBERT architecture). We should consider where in this spectrum to aim, given a moderate compute budget. **Pros of a custom-from-scratch model:** Full control over model capacity, no reliance on external data or weights (so very transparent). We can tailor the size to our dataset: for example, 6 layers might train faster and suffice, but if we observe underfitting or want a margin, we could use 8 or 12 layers. Since we have on the order of hundreds of thousands to a million training examples (if each song yields multiple phrase samples), a moderately deep model can be trained effectively from scratch. Unlike many NLP tasks where labeled data is scarce and pretraining is crucial, here we actually have a large labeled dataset (genre labels for over 100k pieces in XMIDI). That supervised signal might be enough to train a model to good accuracy without unsupervised pretraining. Moreover, genre classification is a coarse task – even simpler models (like CNNs on piano rolls or music-specific CNNs) have achieved decent accuracy in past research [34]. A transformer encoder will likely do very well with enough data. **Cons:** A from-scratch model won't have any prior "musical knowledge" – it must learn everything about music style differences purely from the training set. This is fine given the size of the dataset, but it might require more epochs or regularization to ensure it converges well. Also, if the model is too large relative to the data, it could overfit or be inefficient. For example, a full 12-layer BERT on ~1M short sequences might actually be okay, but training time roughly doubles vs a 6-layer. There's a trade-off: a larger model might squeeze out a few extra percentage points of accuracy, but with diminishing returns and higher cost. Considering "moderate compute", one might lean towards a model on the smaller side (to train faster and be deployable with lower latency). Another factor: with no pretraining, a deeper model could be harder to optimize (but techniques like learning rate warmups, etc., are known from BERT training – we can manage it). All things considered, customizing the architecture allows us to experiment – one could even try a 4-layer model to see if accuracy is still acceptable, or a 8-layer if 6 seems insufficient. This flexibility is good to align model complexity with compute budget.

Model Options Ranking (by cost-effectiveness for our task):

1. **Custom Small Transformer (DistilBERT-like, ~6 layers) – Top Choice:** This option best meets the project goals of **balanced performance and practicality**. It forgoes heavy pretraining and instead relies on the ample labeled data to learn the genres. A 6-layer encoder (with say 512 or 768 hidden size) can capture the musical patterns needed for genre classification, while training in a reasonable time. It's also easier to interpret and troubleshoot – fewer layers means we can visualize attention or at least reason about what it might be focusing on (especially with our structured input tokens). In terms of *classification accuracy*, such a model should be strong: genre features like instrument usage, rhythmic patterns, and pitch content are relatively easy for a transformer to pick up. DistilBERT in

NLP retained ~97% performance of BERT [26], and our task likely doesn't require the full complexity of BERT to begin with. Thus, we expect a small model to achieve high accuracy (perhaps a few points below a huge model at most), which is an acceptable trade-off for faster inference/training. **Cost-effectiveness:** With far fewer parameters and faster throughput, we can iterate quickly, tune hyperparameters, and possibly ensemble or do cross-validation if needed, all within a moderate compute budget.

2. **Pretrained MusicBERT (12-layer) – High Performance, Higher Cost:** If the absolute highest accuracy is desired and compute is less of an issue, fine-tuning **MusicBERT** would be the next best option. It likely would top the accuracy charts due to its extensive pretraining on symbolic music [20]. For example, MusicBERT would have already learned subtle distinctions (like typical jazz chord voicings vs classical ones, or swing rhythm vs straight) that a from-scratch model might need more data to grasp. So with MusicBERT, the fine-tuning on our 6 genres could converge quickly and yield excellent results. The reason it's ranked second is the **practicality concern**: it depends on having the pre-trained model available and the willingness to use a large pre-trained model (which might not be open or could carry licensing issues). It's a bit "overkill" relative to the project's stance on not depending on massive pretraining. Still, it's worth noting as an option if one's priority shifts to maximizing accuracy with minimal training time (since you leverage someone else's training effort). Alternatively, one could use MusicBERT's architecture (12 layers, Octuple input) and do a quick unsupervised pretrain on XMIDI itself (like a few epochs of masked modeling) to initialize the model better than random, then fine-tune on genre labels. This is a lighter dependency than using the full million-song pretraining. It could boost performance over purely random init, at some extra compute cost. In any case, a larger model pre-trained on music is a strong contender in pure performance – it just scores lower on the "moderate compute/complexity" scale.

3. **Medium Transformer from scratch (8–12 layers)** – This represents using a bigger custom model without external pretraining. It would likely work and possibly yield a small accuracy gain over the 6-layer model due to higher capacity. But the gains must justify the costs: training time could double and the risk of overfitting or vanishing returns is higher. Considering XMIDI has on the order of 100k pieces, a 12-layer model can probably be trained without severe overfitting (especially if we augment or use dropout, etc.). So this is a viable approach if one has the compute and wants to push performance while still maintaining control (no outside weights). **Pros:** Still transparent (we know what data it sees from scratch), potentially higher accuracy (maybe better at capturing fine-grained differences like classical vs baroque vs romantic within "classical" – though our labels are broad so this extra detail might not translate to better top-level accuracy). **Cons:** Not as compute-friendly. Also, with deeper models one might need to pay more attention to optimization hyperparameters to ensure it trains well from scratch. This ranks below using a pre-trained model because if we are going big, we might as well use the model that has learned from 10× more data. And it's below the small model because it may be overkill for the task. In practice, one could try a 8-layer model as a compromise; it might give a tiny boost for relatively little extra cost. So this is a middling option depending on how tight the compute budget is.

4. **Performer/Linformer – Niche Efficiency Tweaks:** I group these together as specialized architectures for efficiency. They rank lower because our sequence lengths and model sizes don't strictly require them. **Performer** could be considered if, for example, we decided to incorporate **longer sequences (like 16-bar phrases or entire songs)**. In that case, a Performer-based classifier could scale and handle those lengths without quadratic blow-up. If we stick to $\leq$512 tokens, a

normal transformer is fine. Using Performer for ≤512 tokens might slightly speed up training (linear vs quadratic on 512 is not a big difference, and standard transformers are heavily optimized on GPU). It also might introduce approximation error that's unnecessary. **Linformer** similarly – it would matter if we had very long sequences or extremely large batch sizes. For a moderate setup, the complexity saved is marginal. Moreover, adding these might complicate using readily available tools (for instance, HuggingFace's Trainer doesn't natively support Performer without custom code, whereas it does support regular and Longformer models). So, while not *wrong* to use them, they are not the most practical choice now. They'd rank higher only if memory was a limiting factor or if in experimentation we saw that a much longer input is needed (which we don't expect, since 4 bars is our target).

5. **Longformer – Least needed:** As discussed, Longformer's advantages lie in domains far beyond our input length. For music phrase classification, it doesn't offer any benefit. If anything, restricting attention to local windows could impede the model's ability to catch global context (imagine a scenario where a certain pattern spanning bars 1 and 4 indicates a genre – a Longformer with small window might miss that, whereas a standard transformer catches it easily). Unless we had an extremely specific reason (like maybe we wanted to model a *very* long sequence but only care about local motifs plus perhaps a global CLS token), Longformer is unnecessary. It's the most complex architecture here and yields no gain for short sequences – thus the lowest rank.

Compatibility with MIDI Tokens: All the considered models can ingest our chosen tokens with minimal fuss. We just need an embedding layer that matches the token vocabulary or structure. For the **compound/ Octuple tokens**, a custom model (small transformer) can be designed to have multiple embedding matrices (for pitch, position, etc.) and sum them – this is a bit of extra coding but conceptually straightforward. MusicBERT, if used, already has this built in for Octuple. DistilBERT or any standard transformer implementation assumes a single embedding index per token; we can workaround by treating each compound token as an index in a flat vocabulary (by enumerating all combinations we see in data, or using a fixed encoding function). However, the more elegant way is to incorporate the multi-embedding approach. Since we are leaning toward training from scratch, we have flexibility to implement the input layer as needed. The rest of the transformer (attention layers, etc.) doesn't depend on the content of tokens – so there's no issue there. We should note that **positional encoding** in music can be tricky: absolute positional encoding (as used in BERT/DistilBERT) will mark positions 1,2,3,...N in the token sequence, which in REMI corresponds roughly to chronological order of events (not directly the musical beat position). Given we include bar/beat as tokens or fields, the model can learn musical positional context from that. We might not need any sophisticated relative position encoding (like Transformer-XL or MusicBERT's bar-level masking trick) since our sequences are short and we explicitly encode bar positions. So, a standard absolute positional encoding or even none at all (letting the bar/position tokens serve that role) could be sufficient. This is a minor architectural choice: some music models use **relative attention by bar** (so that patterns repeating each bar are recognized invariant to bar index). That could improve generalization slightly (e.g., recognizing a rock drum pattern in bar 1 vs bar 3 equally). MusicBERT partially addressed this via masking rather than a new attention mechanism. If we wanted, we could incorporate relative positional embeddings (like T5 or Transformer-XL style) to let the model more easily focus on local relative timing. But this is an advanced tweak – not strictly necessary.

Final Model Decision: Build a **compact Transformer encoder (≈6 layers)** and train it from scratch on the tokenized MIDI phrases for genre classification. This essentially means a **DistilBERT-sized model, but trained on music data**. It offers a strong balance between performance and efficiency: enough capacity to learn complex musical style cues, but not so large as to be inefficient. With ~6 layers and our compressed input (Octuple tokens), the model can learn in a reasonable time and will output a genre label via a classification head on the final [CLS] token (or pooled output). This choice aligns with the moderate compute budget – training a 6-layer transformer on ~1M examples is very feasible on a single modern GPU, and inference will be fast (important if classifying many phrases or deploying the model). We maintain **transparency and control** by training from scratch on known data (XMIDI), and we avoid reliance on opaque pre-trained models.

We do acknowledge that if one were purely after the highest accuracy and had more resources, fine-tuning a large pre-trained model (MusicBERT) or training a bigger model might edge out our small model. But those gains are likely marginal for this task and come at the cost of complexity and compute. The chosen architecture, combined with the rich dataset and appropriate tokenization, should already achieve high classification performance (able to distinguish classical vs jazz vs rock, etc., with excellent accuracy) while being practical to develop and use.

Conclusion: Optimal Design for the Genre Classifier

Bringing it all together, the recommended design for the symbolic music genre classifier is:

- **Dataset: XMIDI** as the primary training dataset, because of its extensive size and precise genre annotations covering the target styles [1]. This ensures the model sees a wide variety of examples in classical, jazz, pop, rock, etc. To incorporate the “funk” genre (absent as an explicit label in XMIDI) and the loosely-defined “song” category, we will augment XMIDI with a curated set of funk pieces (e.g. from Lakh MIDI with known funk artists or other sources) and possibly folk/country songs (to represent “song” if needed). This augmentation is relatively small-scale and won’t undermine the moderate compute constraint. Overall, XMIDI provides a high-quality, diverse foundation, minimizing manual labeling effort and maximizing learning of genre characteristics.

- **MIDI Tokenization:** Use an **Octuple/compound token representation** for MIDI events, with each token embedding capturing the note’s pitch, duration, timing (bar/position), and instrument. This approach yields short sequences (well under 512 tokens for 4 bars) without sacrificing detail. By doing so, we significantly lighten the load on the transformer – it can focus on learning genre-specific musical patterns instead of juggling very long sequences. The chosen scheme (Octuple) is proven in literature and strikes a good balance between being informative and efficient. It also naturally fits multi-instrument phrases, which is crucial since instrumentation is a big genre cue (e.g. presence of distortion guitar + drum kit implies rock). We’ll implement this either via a tokenization library or custom embeddings. The end result is a sequence of concise tokens that preserve everything the model needs (melody, harmony, rhythm, instrumentation) to classify the genre.

- **Model Architecture:** Employ a **compact Transformer encoder (~6 layers)**, initialized from scratch and trained solely on the symbolic music data. This model (analogous to a DistilBERT sized network) provides ample capacity to learn genre distinctions but is efficient enough to train on our dataset without “overkill” computational demands. Training from scratch on labeled data gives us full control and transparency – the model’s behavior is wholly determined by the music data we feed it, with no hidden pretraining influences. We’ll use a standard classification head (a dense layer or two on the [CLS] token) to predict one of the six genres. Given the short input length and rich token representation, 6 transformer layers with a few attention heads each can capture both local motifs (like a syncopated rhythm or blues scale lick) and broader structure (like “this phrase uses classical-style voice leading”). This architecture can be trained in a reasonable time frame and will be fast at inference, enabling practical use.

By ranking and choosing these options, we ensure the system is **balanced**: the dataset provides **coverage and quality**, the tokenization provides **compact yet descriptive inputs**, and the model provides **sufficient learning power without unnecessary complexity**. This design should achieve high accuracy in genre classification of short MIDI phrases – for example, correctly labeling a 4-bar piano arpeggio as classical, or a swing melody with walking bass as jazz, or a syncopated slap bass line as funk – all while being mindful of computational efficiency and maintainability. Each design decision supports the project goals: we leverage large-scale symbolic data (for performance), avoid excessive reliance on external resources (for control), and keep the solution tractable in terms of training and deployment (for practicality). The expectation is a state-of-the-art or near-state-of-the-art genre classifier that is both effective and elegantly streamlined for the task at hand. [20] [26]

---

[1] XMusic
https://xmusic-project.github.io/

[2] [3] [4] [11] The Lakh MIDI Dataset v0.1
https://colinraffel.com/projects/lmd/

[5] [6] The MAESTRO Dataset
https://magenta.tensorflow.org/datasets/maestro

[7] GitHub - bytedance/GiantMIDI-Piano
https://github.com/bytedance/GiantMIDI-Piano

[8] Video of the Month: MusicNet: The First Publicly Available Dataset of ...
https://signalprocessingsociety.org/newsletter/2017/01/video-month-musicnet-first-publicly-available-dataset-labeled-classical-music

[9] NES-MDB Dataset | Papers With Code
https://paperswithcode.com/dataset/nes-mdb

[10] The NES Music Database: A Multi-instrumental Dataset with Expressive Performance Attributes | Chris Donahue
https://chrisdonahue.com/publication/18-06-nesmdb/

[12] [13] [14] [15] [16] [17] [18] [19] [23] [25] Tokenizations - MidiTok's docs
https://miditok.readthedocs.io/en/latest/tokenizations.html

[20] [21] MusicBERT: Symbolic Music Understanding with Large-Scale Pre-Training | Papers With Code
https://paperswithcode.com/paper/musicbert-symbolic-music-understanding-with

[22] [PDF] MIDITOK: A PYTHON PACKAGE FOR MIDI FILE TOKENIZATION
https://archives.ismir.net/ismir2021/latebreaking/000005.pdf

[24] [PDF] Byte Pair Encoding for Symbolic Music - ACL Anthology
https://aclanthology.org/2023.emnlp-main.123.pdf

[26] DistilBERT — transformers 2.4.0 documentation
https://huggingface.co/transformers/v2.4.0/model_doc/distilbert.html

27 lucidrains/performer-pytorch: An implementation of ... - GitHub
https://github.com/lucidrains/performer-pytorch

28 29 [2009.14794] Rethinking Attention with Performers - arXiv
https://arxiv.org/abs/2009.14794

30 31 32 [2006.04768] Linformer: Self-Attention with Linear Complexity
https://arxiv.org/abs/2006.04768

33 Longformer - Hugging Face
https://huggingface.co/docs/transformers/en/model_doc/longformer

34 Exploring the Design Space of Symbolic Music Genre Classification ...
https://ieeexplore.ieee.org/document/5172597/