ChatGPT

3-Month Sprint Plan: Symbolic Music Genre Classifier (MIDI)

This plan outlines a 3-month solo project (7 sprints of 2 weeks each) to build a symbolic-music genre classifier for short MIDI phrases. Each sprint has a clear goal, bite-sized tasks (1–2h each to suit 1–2h sessions), optional stretch goals, and a deliverable (runnable code or demo). The focus is on achieving high accuracy for classical and jazz genres, while covering other genres (funk, pop, rock, folk) as secondary priorities.

Sprint 1: Setup & Data Preparation

Main Goal: Establish the development environment and prepare the dataset of short musical phrases with genre labels. This involves getting the XMIDI dataset [1] and extracting 2–4 bar phrases for use in model training, with emphasis on classical and jazz pieces.

Tasks (1–2h each):
- **Environment setup:** Install required libraries (e.g. Python MIDI processing like `miditoolkit`/`MidiTok`, PyTorch for modeling) and set up version control.
- **Dataset acquisition:** Download and verify the XMIDI dataset (108k MIDI files with emotion & genre labels [1]). Filter or subset it to manageable size focusing on genres of interest (ensure plenty of **classical** and **jazz** examples, but include some pop, rock, folk for completeness).
- **Genre data exploration:** Write a small script/notebook to parse dataset metadata and count files per genre (to see class distribution). Identify if any target genre (e.g. "funk") is missing or underrepresented in XMIDI.
- **Phrase segmentation:** Implement a function to split full MIDI pieces into short phrases (2–4 bars). Utilize tempo and time signature info to define bar boundaries. For each MIDI, generate multiple phrase clips and inherit the piece's genre label for each clip.
- **Verification:** Test the segmentation on a handful of MIDI files – for example, ensure a classical piece yields 2–4 bar segments that sound musically self-contained. Adjust bar-length or segmentation logic if needed.
- **Data structuring:** Organize the extracted phrases and labels into a training dataset structure (e.g. a folder or CSV listing file paths and genre labels). Make sure classical/jazz phrases are well-represented and consider balancing if one genre has far fewer phrases.

Optional (if time permits):
- *Additional data:* If **funk** genre is not present in XMIDI (likely, since XMIDI's genres include classical, jazz, pop, rock, country, folk [2]), curate a small set of funk MIDI files from external sources. Label them as "funk" and add to the dataset for diversity. Similarly, double-check if "folk" or any **song** category needs augmentation and include a few examples.
- *Deeper EDA:* Log more stats (phrase length in beats, note density, etc.) per genre. This can reveal characteristics (e.g., jazz phrases might have more syncopation) and help later in error analysis.

Deliverable: A Jupyter notebook (or Python script) that loads the XMIDI data, produces a set of 2–4 bar MIDI phrase samples with genre labels, and prints out basic dataset stats (e.g. number of phrases per

genre). The phrases dataset is saved/ready for tokenization in the next sprint, and a couple of example phrases can be played or inspected to confirm the segmentation works.

Sprint 2: Tokenization Pipeline (Octuple Encoding)

Main Goal: Implement an event-based tokenization pipeline for the musical phrases, using the **Octuple** encoding as the primary scheme. By the end of this sprint, the code should convert any given MIDI phrase into a sequence of tokens suitable for transformer input. This sets the stage for model training.

Tasks (1–2h each):
- **Octuple tokenizer setup:** Integrate an Octuple MIDI tokenization method. This can be done via a library like MidiTok or by custom implementation. (Octuple, introduced in MusicBERT, represents each note as an 8-value token: e.g. pitch, position, bar, velocity, duration, instrument/program, etc. [3] – which greatly reduces sequence length by pooling note attributes.) Configure the tokenizer to include relevant token types (especially those affecting classical/jazz, e.g. pitch, duration, tempo, etc.).
- **Tokenize sample phrases:** Run the tokenizer on a small subset of phrases (a few classical and jazz clips) to ensure it produces reasonable token sequences. Inspect the output – e.g., verify that token sequences reflect the original phrase (correct pitches, reasonable timing tokens). If available, decode the tokens back to MIDI and listen or compare to ensure fidelity.
- **Dataset tokenization:** Tokenize the entire phrase dataset. This might involve writing a loop or using the library's batch processing to convert all MIDI phrase files into token sequences stored in memory or on disk (e.g., saved as JSON or pickled sequences). Handle varying sequence lengths by padding or truncation as needed (ensure 2–4 bar phrases generally fit within model length).
- **Prepare input tensors:** Assign integer IDs to tokens (build the vocabulary from the Octuple tokens). Create training/validation splits of the tokenized data. If using PyTorch, implement a Dataset class that yields token sequences and genre labels, ready for the model to consume.
- **Memory/efficiency check:** Given the limited work hours, ensure the tokenization process is efficient (possibly tokenize once and save, rather than re-tokenizing each training epoch). Verify that the sequence lengths with Octuple are manageable for the model (Octuple should yield much shorter sequences than naive event encoding [4]). If any phrase is extremely long (unlikely for 4 bars), document a strategy (such as skip or further split).

Optional (if time permits):
- *Alternate tokenizations:* Implement support for one other tokenization scheme for comparison later. For example, set up a REMI tokenizer (Rhythm-based Event Representation from Pop Music Transformer [5]) or **Compound Word** (CP) tokenizer [6]. Don't fully convert the whole dataset, but tokenize a small sample in an alternate format to see how sequence length or structure differs. This will enable a future experiment on whether Octuple or another scheme yields better accuracy.
- *Instrumentation tags:* If multi-instrument MIDI files are present, ensure the tokenizer handles program changes or instruments (Octuple can include Program tokens [7]). Optionally, test whether including instrument tokens or not affects the representation quality for genre classification.

Deliverable: A data preprocessing module (script or notebook) that can take a MIDI phrase and output a sequence of token IDs using Octuple encoding. The full training and validation datasets are now available in tokenized form (e.g., saved as sequences of integers with corresponding genre labels). You should be able to, for example, load a tokenized jazz phrase and a classical phrase and see their token sequences. This is verifiable by feeding an example MIDI through the code and confirming the output tokens.

Sprint 3: Model Architecture & Baseline Training

Main Goal: Develop a compact transformer encoder model (DistilBERT-sized) for genre classification and train a first baseline model. By sprint's end, there will be a basic model that runs end-to-end: taking token sequences as input and outputting a genre prediction. Initial accuracy will be modest, but this provides a baseline for improvement.

Tasks (1–2h each):
- **Model definition:** Implement the transformer encoder architecture. Aim for a **compact model** – e.g., 4–6 self-attention layers, moderate embedding size (maybe 128–256), and a vocabulary size covering the tokenization from Sprint 2. Include a positional encoding mechanism (though Octuple has bar/position tokens, adding a standard positional encoding or using those tokens as position info is fine). Ensure the model ends in a classification head: e.g., use a special `[CLS]` token or global average pooling over the sequence output to produce a fixed-length representation, then a dense layer for the genre classes.
- **Training pipeline:** Set up the training loop or use a framework (like PyTorch Lightning or HuggingFace Trainer) to handle training. Define loss (cross-entropy for multi-class genre classification) and an optimizer (AdamW is a good default). Load the tokenized dataset from Sprint 2 into this pipeline with batching.
- **Baseline run on subset:** To make sure everything works, run a short training test on a small subset (or a single batch) of data. Verify the model can forward-pass and the loss decreases for a few iterations (catch any shape or type errors). This can be done in ~1h sessions: coding the training loop and another session to debug by running it.
- **Full training run:** Train the model on the full (filtered) dataset. Given limited compute/time, start with a reasonable number of epochs (e.g., 5–10 epochs over the data) or train until the model converges or overfits. This training can run asynchronously (overnight or during free time), but monitor periodically for any crashes or issues.
- **Validation check:** After training, evaluate the model on a validation set. Compute overall accuracy and class-wise accuracy. Record baseline performance, especially noting **classical** and **jazz** accuracy vs. other genres. Don't expect high accuracy yet, but this sets a reference point.
- **Result logging:** Save the model checkpoint and log basic metrics (loss curves, accuracy). Note any obvious problems (e.g., if the model predicts only one class, or if jazz accuracy is particularly low). This informs the next sprint.

Optional (if time permits):
- *Lightweight baseline:* Before or in parallel with the transformer, train a quick non-transformer baseline (e.g., an LSTM or even a logistic regression using simple features like note density or pitch range) on a small sample. This can give a sanity-check that the task is learnable (e.g., logistic regression might achieve some basic accuracy above chance). This is strictly for insight and not a priority since the focus is the transformer model.
- *Binary classifier test:* If focusing on classical vs jazz is critical, try a quick experiment training a binary classifier (classical vs. jazz only) to see how well the model can differentiate those two in isolation. This could reveal if those genres are easily separable from raw token sequences, and results might suggest if further feature engineering is needed for those specifically.

Deliverable: An initial trained genre classifier model (small transformer) and a training notebook/script. The deliverable should include the model's first validation results – for example, "Baseline model achieves X% overall accuracy, with Y% on classical and Z% on jazz." The code is runnable to produce a genre prediction on a given tokenized phrase (demonstrate by predicting a few examples: e.g., feed a known jazz phrase and

see that it predicts “jazz” or see where it fails). This establishes the end-to-end pipeline from MIDI input to genre output.

Sprint 4: Model Refinement & Genre-Focused Tuning

Main Goal: Improve the classifier’s performance, with special emphasis on **classical** and **jazz** genre accuracy. This sprint involves analyzing the baseline, addressing its weaknesses (data imbalance, model capacity, etc.), and tuning the training for better results. By the end, the model should show a clear improvement, particularly on classical and jazz classification.

Tasks (1–2h each):
- **Error analysis:** Review the baseline model’s evaluation. Identify which genres are most often misclassified. Specifically check classical and jazz precision/recall – are classical phrases being confused with others (perhaps “folk” or “pop”?) or jazz mislabeled as rock, etc.? Look at a few misclassified examples (if possible, listen to or inspect the MIDI phrases) to understand if there’s a pattern (e.g., maybe fast jazz phrases get confused with classical).
- **Data balancing:** If the dataset is imbalanced (likely, given differing counts per genre), implement class weighting in the loss or oversample underrepresented classes in training batches. For example, ensure that each epoch has ample classical and jazz samples (even if it means repeating them more often than abundant genres like pop) so the model gets sufficient signal on these categories.
- **Hyperparameter tuning:** Adjust key hyperparameters to improve learning: try a slightly higher model capacity if underfitting (e.g., more attention heads or an extra transformer layer) or regularization if overfitting (dropout, weight decay). Also experiment with training parameters – e.g., learning rate schedule, number of epochs, batch size – within the available time. Use the validation set to compare improvements.
- **Focused re-training:** Retrain or fine-tune the model with the new settings. For instance, you might train for more epochs or do a two-stage training (first on all genres, then a few extra epochs on just classical+jazz data to fine-tune those – careful to avoid forgetting others, perhaps using a mixed strategy). Monitor training curves to ensure the changes are helping (e.g., validation loss going down, accuracy up).
- **Genre-specific boosts:** Implement any genre-targeted improvements discovered in analysis. For example, if **jazz** phrases often have swing rhythms that the model struggles with, consider adding an input feature or token (if available) to indicate swing tempo, or augment jazz data by transposing to increase variety. If **classical** pieces have wide dynamic ranges, ensure velocity is well-represented in tokenization (Octuple already includes Velocity token [7]). Even simple data augmentations can help – e.g., transpose some classical phrases up/down a few semitones (excluding pieces where that would change genre, like those with fixed bass patterns) to increase training samples.
- **Evaluation:** After improvements, evaluate the model on the validation (or a dedicated test set if available). Compare new metrics to the baseline. Ideally, see a boost in classical/jazz F1-scores. Ensure other genres didn’t collapse – they may be lower priority, but should still be reasonable. If classical/jazz accuracy is still unsatisfactory, plan additional tweaks or note potential future work.

Optional (if time permits):
- *Advanced error analysis:* Plot a confusion matrix of genres to visualize where mistakes occur most (e.g., jazz vs. blues confusion if “blues” exists, or classical vs. folk). Also, consider using t-SNE or PCA on the learned phrase embeddings to see if classical and jazz phrases form distinct clusters – this can qualitatively show improvement in the model’s genre differentiation.
- *Ensemble or secondary model:* If one genre (say jazz) remains tricky, consider training a secondary specialized classifier that only triggers between certain genres (for instance, a small classifier to distinguish

jazz vs blues if the model confuses those). This is likely beyond scope, but brainstorming these ideas can inform final adjustments.

Deliverable: An improved genre classifier model with updated training code and a brief report of results. For example, deliver a updated notebook showing "After tuning, overall accuracy is now X+Δ%, classical accuracy improved to Y% (from Y0%), and jazz to Z% (from Z0%)." The sprint's result should be a **working model** ready to handle the primary genre targets well. The code and logs should demonstrate the changes (e.g., a plot of the confusion matrix or metrics table before vs after tuning).

Sprint 5: Extend Genre Coverage & Data Augmentation

Main Goal: Expand and solidify the model's coverage of all target genres (including the "nice-to-have" ones like funk, pop, rock, folk) and incorporate any additional data. This sprint ensures the classifier isn't over-fitted only to classical/jazz, and that it performs acceptably on other genres. It also adds any remaining data (e.g., curated MIDIs for missing genres) and applies final data augmentations to bolster the training set.

Tasks (1–2h each):
- **Incorporate new genre data:** If in Sprint 1–4 we identified missing genres (e.g., *funk* was added with a few MIDIs), integrate those into the training set now. Tokenize any new MIDI files using the established pipeline (from Sprint 2) and add them to the dataset. If new genres were added (like funk), update the model's output layer to include that class if not already.
- **Augment dataset:** Apply light augmentation to increase model robustness. For symbolic music, possible augmentations include transposing phrases (for genres where transposition doesn't alter genre identity, e.g., classical pieces can often be transposed, but a blues riff might not sound like blues if transposed too much – use judgment). Another augmentation: slight random timing shifts or velocity perturbations to simulate performance variations (particularly for jazz which often has humanized timing). Augment especially the genres that have few examples (maybe you have only 20 funk phrases – you can transpose them up a step to make 20 more).
- **Retrain full model:** Train the classifier on the expanded and augmented dataset. This could be a fresh training run or a continuation fine-tune from the Sprint 4 model. Given more data and possibly more classes, monitor training to ensure it still converges well. This training might need the full two weeks in terms of wall-clock (with the user mainly monitoring in bursts).
- **Comprehensive evaluation:** Evaluate on a test set covering all genres. Measure each genre's accuracy. Confirm that classical and jazz remain high (primary goal) and check that the newer/augmented genres (funk, etc.) achieve reasonable accuracy (even if lower, at least better than random). For example, you might report "Model is 90% on classical, 85% on jazz, and ~70% on pop/rock, lower (~50-60%) on the very few funk examples – acceptable for now."
- **Stability check:** Ensure that adding new data didn't significantly degrade the model or cause overfitting. If any issues arose (e.g., the model started overfitting due to augmentations or became unstable), make minor fixes (like reduce learning rate or simplify augmentation). By the end, the model should be **finalized in terms of architecture and training data**.

Optional (if time permits):
- *Hyperparameter sweep:* With the full dataset and final setup, do one more round of hyperparam tuning if possible – e.g., a brief grid search or manual trial of two different learning rates or dropout rates to squeeze out a bit more accuracy. This is optional because it can be time-consuming, but even a small experiment might yield an extra percent or two of performance.

- *User testing:* If you have access to domain experts or just your own musical ear, do some manual testing – pick a few phrase MIDIs from each genre not seen in training, run them through the classifier, and see if the predictions align with expectation. This qualitative check can be a nice validation (and could be part of the demo in the next sprint).
- *Logging & stats:* Implement final logging for training metrics and possibly store dataset metadata (like a CSV of all phrases with their length, genre, source). Having these records is useful for documentation and any future analysis.

Deliverable: A finalized multi-genre classification model and an updated training artifact that reflects the full dataset training. This sprint should produce the **trained model ready for use**. Additionally, intermediate results like the final confusion matrix or per-genre metrics should be available, showing that classical and jazz accuracy meet the goal and others are reasonable. Essentially, the project now has a robust classifier covering the intended genres, setting the stage for demonstrating it.

Sprint 6: Comparative Experiments & Analysis

Main Goal: With a solid model in hand, use this sprint to conduct **experiments and analyses** that were optional in earlier sprints. This includes comparing different tokenization schemes or model variants, and gathering deeper insights into the model and dataset. The goal is to validate that our choices (like Octuple encoding and model size) were appropriate, and to document findings for future work.

Tasks (1–2h each):
- **Tokenization comparison experiment:** Using the pipeline from Sprint 2, tokenize a portion of the dataset (or the whole, if feasible) with an alternative representation, such as **REMI** or **CP-Word**. Train a **small trial model** (maybe fewer epochs or smaller subset) using this alternate tokenization but the same model architecture, and compare classification performance to the Octuple-based model. For example, does REMI (which represents events sequentially with separate tokens [5]) yield lower or higher accuracy than Octuple for jazz/classical? Measure training speed and accuracy. Document the results.
- **Model size/variant experiment:** Experiment with a different model configuration: e.g., try a slightly larger model (more layers or wider embeddings) to see if it improves accuracy notably, or conversely a smaller model to test the limits. Since time is limited, use only a couple of epochs or a subset for these trials. The outcome might be "Model A (DistilBERT-size) vs Model B (half the size) – accuracy drops X% when halved, suggesting the chosen size is reasonable."
- **Dataset statistics logging:** Compile comprehensive stats from the dataset and model behavior. This can include: distribution of phrase lengths, distribution of tempos or keys per genre, any correlation between certain musical attributes and model errors, etc. If not done earlier, generate plots or tables (e.g., average note count in classical vs jazz phrases). This is partly for documentation and understanding the model's context.
- **Jazz phrase generation (exploratory):** As a forward-looking exploration, attempt a **simple music generation experiment** focusing on jazz. For instance, take the jazz phrases subset and train a lightweight language model (could reuse the transformer encoder weights in a seq2seq or use an LSTM) to generate token sequences, just to see if the tokenization and data can produce coherent output. This could be as simple as training a small Transformer decoder on jazz token sequences to predict the next token. The aim is not a polished generator but a proof-of-concept that highlights how the learned representation might be used creatively. If successful, generate a few 4-bar jazz phrases and save them as MIDI.
- **Documentation of findings:** Begin writing down the findings from these experiments. Note whether

Octuple remained the best choice, how the model scales with size, and any interesting data insights. This will flow into final project report/documentation.

Optional (if time permits):
- *Multi-task or transfer learning experiment:* Given interest in generation and possibly emotion (from XMIDI labels), an extra experiment could be to see if the genre classifier's embedding can be repurposed. For example, check if the transformer's embeddings can cluster phrases by emotion or if a simple linear probe on those embeddings can predict another attribute (emotion or composer) to gauge how much musical information the model encodes. This is a stretch goal and only if time allows.
- *Refine generation:* If the jazz generation trial produces something interesting, spend a bit more time improving it (tweak the model or sampling method) just to have a neat demo artifact (e.g., "our classifier project also generated these jazz-like phrases").

Deliverable: A set of experimental results and supporting code. Concretely, this could be a short report (within a notebook or markdown) summarizing the comparison between tokenizations and any model variant tests, accompanied by charts or tables. If the optional generation was attempted, include the sample generated MIDI phrases as output files or brief audio, demonstrating an exciting extension of the project. All code used for these experiments should be saved (even if not part of the core classifier) for transparency. This sprint's outcome is more about insights and documentation rather than a new product – it validates and enriches the project with additional knowledge.

Sprint 7: Final Model, Demo, and Wrap-Up

Main Goal: Conclude the project by packaging the final classifier model into an easy-to-use form and creating a demo or presentation of the results. This includes cleaning up the codebase, writing documentation, and perhaps a small interactive demo to show the classifier in action. The sprint ends with a deliverable that can be shown to others (or future you) as a proof of the project's success.

Tasks (1–2h each):
- **Finalize model training:** If needed, do a last training run incorporating any insights from Sprint 6 (for example, if a slight hyperparameter tweak or adding a bit more data can improve the model, apply it now). Ensure the final model is saved with all necessary files (model weights, tokenizer/vocab files, etc.).
- **Model packaging:** Write a simple inference script or function that takes a raw MIDI file (or a sequence of notes) as input and returns the predicted genre. This should encapsulate tokenization + model forward pass + mapping to genre name. Test it on a couple of known examples for validation.
- **Interactive demo:** Create a small demo for the classifier. For example, a Jupyter Notebook with a few example MIDI phrase files (perhaps one classical, one jazz, one pop) where the model predicts the genre, showing the result. Optionally, implement a simple command-line interface or a Streamlit web app for convenience: the user selects a MIDI file and the app displays the predicted genre. Given the time, a Jupyter Notebook demo with clear instructions and examples may be most realistic.
- **Documentation:** Prepare a professional README or report summarizing the project. This should include: overview of the problem and scope, model architecture (mention the DistilBERT-like transformer encoder), dataset description (XMIDI and any added data), tokenization approach (Octuple and why), training process, and results (with focus on classical/jazz accuracy achieved). Also document how to run the code and use the model. If any optional experiments (tokenization comparison, generation) were done, briefly mention these as additional findings or future work. Ensure to highlight that classical and jazz classification performance was prioritized and achieved to a high standard.

- **Code cleanup:** Spend some time refactoring or commenting the code for clarity. Remove any temporary/legacy code from earlier experiments that isn't needed. Make sure each module (data prep, training, inference) is well-organized. This will make the project maintainable or easier to hand off.
- **Buffer for polish:** Use any remaining time to polish the deliverables – e.g., double-check that the demo examples are correct (the model predicts correctly on them), maybe prepare a short slide or just ensure the README has nice formatting. If possible, gather final feedback (maybe a friend tests the demo) and refine accordingly.

Optional (if time permits):
- *Extended demo:* If very enthusiastic, you could embed a few audio renditions of example phrases with their predicted genres in the documentation or as part of a presentation. Hearing a phrase and seeing the model's label could be a powerful validation of the classifier's ability.
- *Future roadmap:* Jot down ideas for future sprints or projects (this could include improving the model with pre-training on unlabeled music, extending to more genres, or integrating the classifier into a music generation pipeline). While not required, it shows forward thinking and can be added in the documentation's conclusion.

Deliverable: A **complete project package** ready to share or deploy. This includes the final trained model (and tokenizer), a well-documented codebase, and a user-facing demo (notebook or app) where one can input a MIDI phrase and get a genre classification. The deliverable also encompasses a short report or README that highlights the achievements (e.g., "Achieved ~90% accuracy on classical/jazz genre classification for 4-bar MIDI phrases, using a custom transformer classifier"), ensuring that anyone reviewing the project can understand what was done and how to use it. The project is now successfully wrapped up with all goals met.

---

[1] [2] XMusic
https://xmusic-project.github.io/

[3] [4] [5] [6] [7] Tokenizations - MidiTok's docs
https://miditok.readthedocs.io/en/latest/tokenizations.html