For oversampling, we stratify-resample the datasets as previously while oversampling genres that are under the desired track count to the wanted ratio (e.g if we want 2000 tracks per genre, metal will be resampled to a 1.3 ratio) and preprocess the track as many times as it is resamples. As we wil see later, we introduce a random cropping operation in the preprocessing pipeline which minimizes the artifical aspect of resampling the same track, as the given track is sampled at various starting points each time.

8. Pre-processing pipeline design

In this section we discuss the specifics of the preprocessing pipeline in terms of design and implementation. This includes audio preprocessing but also the wrangling, subsampling and downloading pipeline to smooth out the iterative testing process on multiple datasets and models, while taking into account the possibility of changing input length as well as preprocessing hyperparameters along the way.

In terms of pure audio preprocessing and based on techniques described in Chapter 3, we choose the following steps:

1. Render the file into mono.
2. Resample the file to the target sample rate using the base algorithm described in section 3. Given the data presented in the previous section concerning the available datasets, we choose 16kHz as it is both a limiting factor due to the FMA dataset and a predominant parameter of choice in the state of the art.
3. Clip out the start and end of the tracks (5s) to remove intros and outros
4. Random crop clip to approximate fixed length input (30s)
5. Split audio into overlapping audio clips of target input length computed as $floor(length(s) * sr(Hz))$ with an overlap of half of the target input length (e.g a 10s clip divided into 5s clips with a hop of 2.5s will yield 4 5s clips)
6. Generate log-power mel spectrograms with 128 mel bins based on the state of the art observed in [51]. *Note that for now, only individual melgrams are saved but it could be interesting to look into sequences of melgrams for convolutional RNNs and/or convolutional transformers and more generally sequence classification.*

An example of the preprocessing hyperparameters used to generate a melgram dataset for one of the trained models is shown below :

```
    preprocessing_config = {
    "mono": True,
    "target_sample_rate": 16000,
    "min_sample_rate": 11250,
    "melgram_hop": 1.5,
    "len_split": 3,
    "melgram_params": {
        "window_length": 512,
        "window_stride": 256,
        "n_fft": 512,
        "n_mels": 128,
    },
    "start_cut": 5,
    "end_cut": 5,
    "random_crop": 30,
    "random_crop_occurence": True,
}
```

To conduct the full construction of a dataset ready for training, including train-test-validation split generation, we first wrangle and merge the selected tags for each tag type based on the work explored in the previous section. unique reproducible identifiers are generated based on track names, paths, and random seeding. We then proceed to subsample and oversample the merged dataset as described previously. To save disk space, we only download the required tracks after resampling. These tracks are stored in the AWS EC2 service and are downloaded from the largest version of every dataset described previously. They take up in total about 2TB of space, but only about 30000 tracks are downloaded at a time.

During download, we also delete the local storage of unrequired tracks (*e.g if the split has changed or the wrangling type has changed*). The local audio is then preprocessed into melgrams as described previously, which yields a local dataset of melgrams on which train-test-validation splitting is conducted. The global pipeline is shown in the following figure (Figure 8.1)

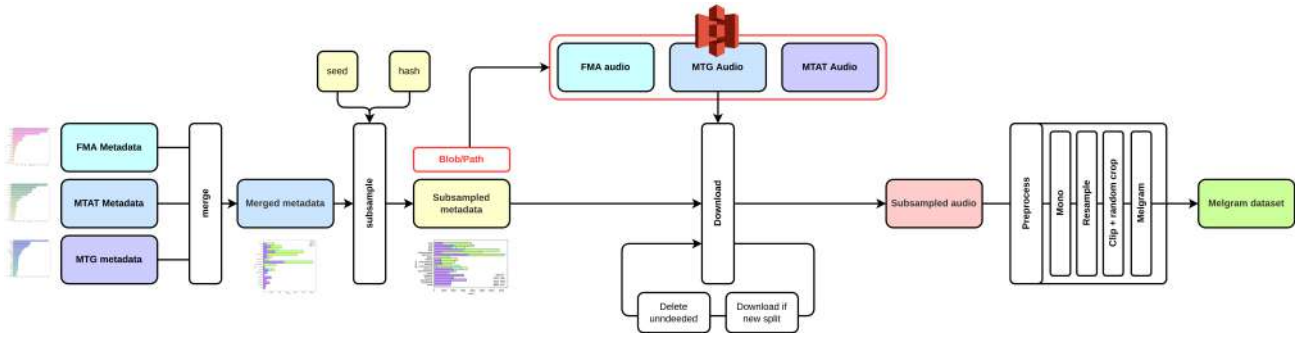

Figure 8.1.: General pipeline : from wrangling to train test splitting

Refactoring the pipeline to avoid sampling bias and data leakage Up till now, the code used to generate the training data and the whole preprocessing pipeline led to some form of data leakage between the train and test sets, meaning that some songs from the train set ended up in the validation set. Furthermore, sub/oversampling was conducted before train/test/validation splitting which is obviously undesirable as it introduces a sampling bias into the results. To summarize the previous full pipeline, the following schematic shows the pipeline from wrangling to training BEFORE refactoring (Figure 8.2).

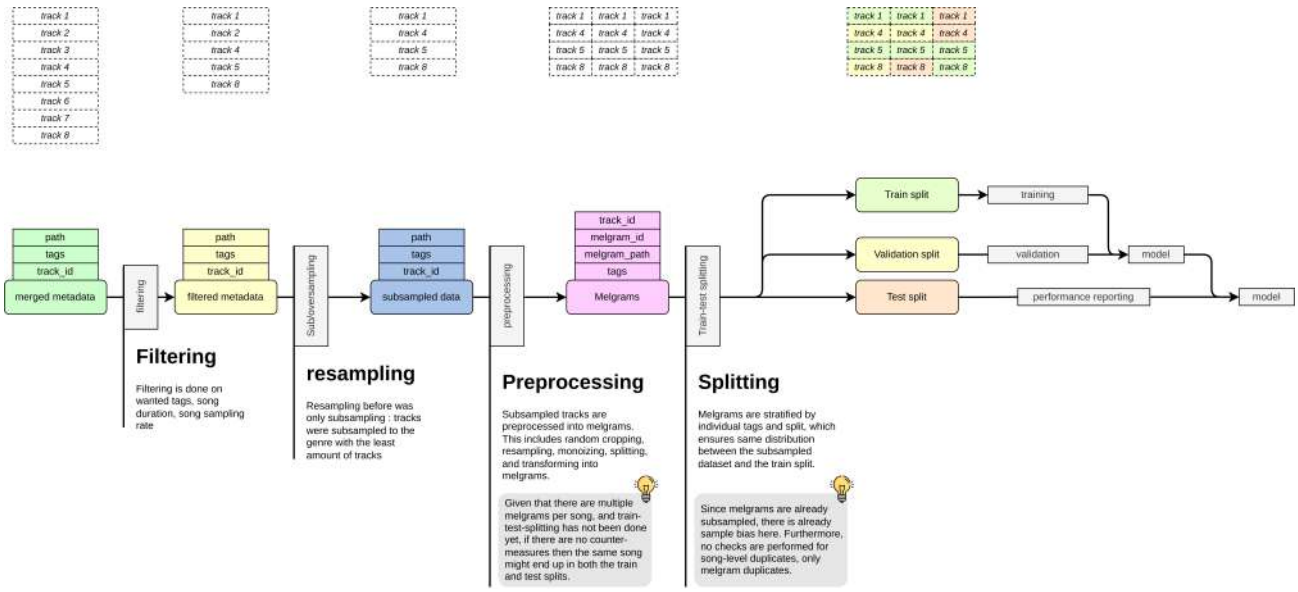

Figure 8.2.: Full pipeline before refactoring for leaking and sampling bias

One can see how both the issues of sample bias and information leakage arises for the test and validation set.

We refactor the pipeline to adjust for this problem by moving the subsampling downstream and the train-test-validation split downstream. Furthermore, since we don't want to introduce sampling bias in the validation and test set, we only resample the training set. These changes lead to the following pipeline (8.3):

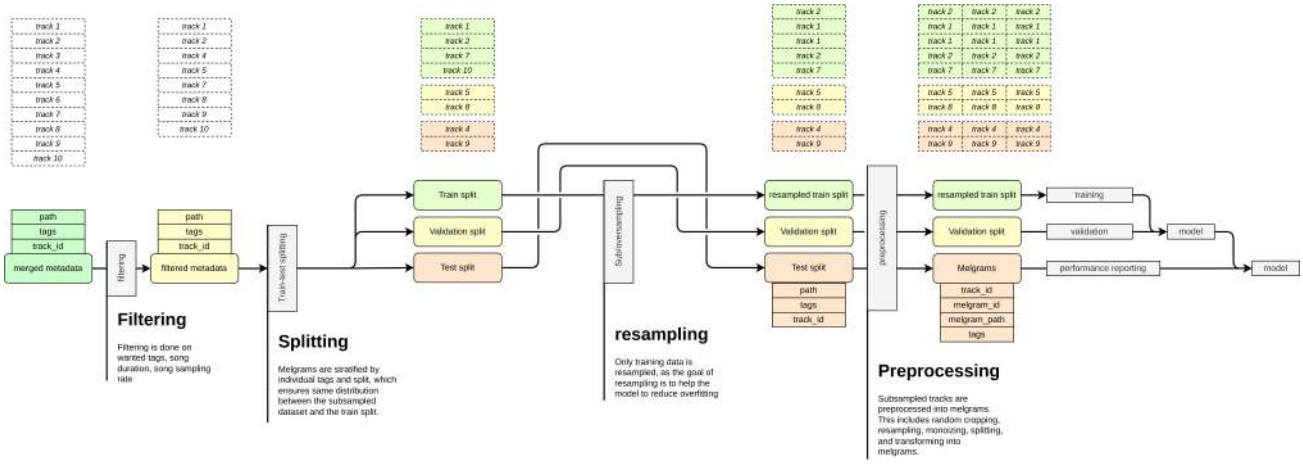

Figure 8.3.: Full pipeline after refactoring for leaking and sampling bias

This major problem being fixed, a quick sanity check shows that after the refacto track-ids are now exclusive to either the train, test or validation sets, and that there is no overlap between the sets. we can also check the result of oversampling only the training set. If all goes well, the test and validation set should have about the same distribution of flattened genres, and the training set should be more equally split. This is shown in figure (8.4). The same graphs for the rest of the tag types (subgenres, moodthemes, alltags) are shown in annex B (B.2, B.3, 7.5d).

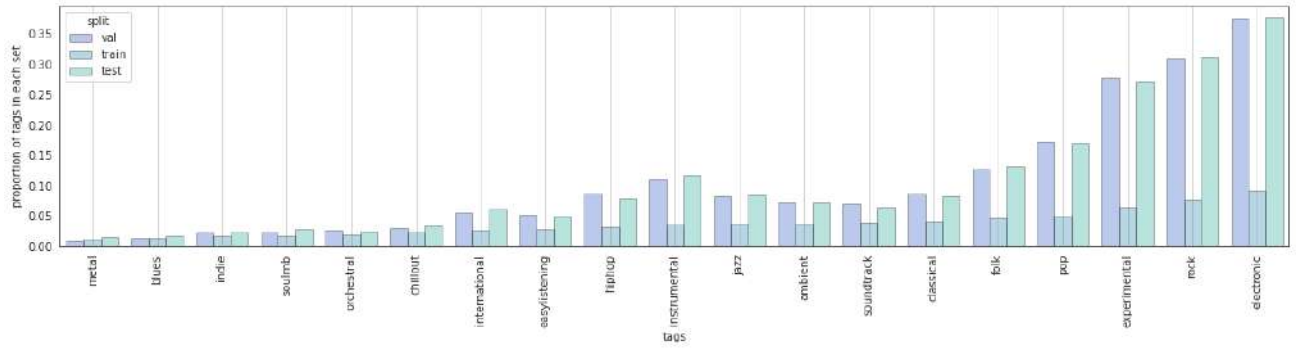

Figure 8.4.: Resampled genres split into train test and validation split. A more equal distribution can be seen in the training set, and an approximately equal distribution in the test and validation set, which is what was sought

9. Preliminary model selection

In section 4, we presented a total of 12 models and their variations which could serve towards the task of automatic music tagging. These models were of variable complexity, size, input length and availability. By availability, we mean that the implementation of some of these models is open-sourced, and some of them are even available pretrained online (e.g MusiCNN and SampleCNN) are available online. Furthermore, an additional constraint is that the tech stack for Groover is mainly based on TensorFlow [26], one of two main python packages for machine learning development along Pytorch [31]. The following criteria are taken into account when selecting the base models we will implement and test on our custom dataset:

- **Complexity of implementation**. If the model is highly complex to implement and not available in open-source implementation, it would take several weeks to implement and to verify its performance by reproducing results, where this work is a benchmark work rather than a results reproduction work.
- **Melgram input representation**. As a conscious choice to simplify the preprocessing pipeline as well as the wrangling pipeline, we choose to focus on models with melgram inputs as our main models. This is with the issue of disk space saving in mind as well as performance shown in state of the art over raw audio models
- **Low input dimension**. Corroborated by state of the art, small input lengths seem to perform better that long chunks (eg FCN and CRNN). This is beneficial from a space saving standpoint, so we choose to prefer models with lower length inputs
- **Open source availability**. This goes in hand with point 1, but an open source reproducible model is always better than rewriting it from scratch. If the model is pretrained, then bonus points.

With these criteria in mind, the following table recaps these criteria for each available model in the state of the art (*Note: Performance on state of the art datasets is important, but as shown in [51]), these performance metrics differ by small margins. As our task dataset has never been used before, and there is no benchmark on it, we remain conservative towards ousting models because they were beaten by a percentage point on a very specific task in state of the art*:

Model	*Input format*	*Input length*	*Open source?*	*pretrained?*	*Implemented in*
FCN	Melgrams	30s	Yes	No	Pytorch
CRNN	Melgrams	30s	Yes	No	Pytorch
MusicNN	Melgrams	3s	Yes	Yes	Pytorch / TF
HarmoniCNN	STFT	3s	Yes	No	Pytorch
SampleCNN	Audio	3s	Yes	No	Pytorch
Ensemble SampleCNN	Audio	3s	No	No	Pytorch
SampleCNN+SE	Audio	3s	Yes	No	Pytorch
ShortChunk	Melgrams	3s	Yes	No	Pytorch
ShortChunk ResNet	Melgrams	3s	Yes	No	Pytorch
CNSSA	Melgrams	15s	Yes	No	Pytorch
Semi-Supervised CNSSA	Melgrams	15s	No	No	Pytorch
SpecTNT	Melgrams	3s	No	No	Pytorch
CALM	Audio	-	Yes	No	Pytorch
CLMR	Audio	-	Yes	No	Pytorch

Table 9.1.: Preliminary model selection

Unfortunately, many models are not available in TensorFlow except for musicNN, which has a different input shape than our own data (due to slightly different input lengths). This and the non-sequential nature of MusiCNN prevents us from using the pretrained version in comparison with the other models. In the above table in green are shown the models that were implemented and trained on our dataset. In light yellow are