

INTRODUCING `DiMCAT` FOR PROCESSING AND ANALYZING NOTATED MUSIC ON A VERY LARGE SCALE

Johannes Hentschel¹

Andrew McLeod²

Yannis Rammos¹

Martin Rohrmeier¹

¹ Digital and Cognitive Musicology Lab, École Polytechnique Fédérale de Lausanne, Switzerland

² Fraunhofer IDMT, Ilmenau, Germany

johannes.hentschel@epfl.ch

ABSTRACT

As corpora of digital musical scores continue to grow, the need for research tools capable of manipulating such data efficiently, with an intuitive interface, and support for a diversity of file formats, becomes increasingly pressing. In response, this paper introduces the Digital Musicology Corpus Analysis Toolkit (`DiMCAT`), a Python library for processing large corpora of digitally encoded musical scores. Equally aimed at music-analytical corpus studies, MIR, and machine-learning research, `DiMCAT` performs common data transformations and analyses using dataframes. Dataframes reduce the inherent complexity of atomic score contents (e.g., notes), larger score entities (e.g., measures), and abstractions (e.g., chord symbols) into easily manipulable computational structures, whose vectorized operations scale to large quantities of musical material. The design of `DiMCAT`'s API prioritizes computational speed and ease of use, thus aiming to cater to machine-learning practitioners and musicologists alike.

1. INTRODUCTION

Given the proliferation of large corpora of digital scores (e.g., [1–4]), the computational challenges of analyzing symbolically encoded staff notation loom large in Digital Musicology and MIR. In principle, any symbolic music encoding is equally amenable to algorithmic processing, to the extent that it is consistent and comprehensive. In practice, however, the *visual* efficiency of staff notation—which conglomerates tonal, rhythmic, metric, articulatory, and other musical parameters in context-dependent and position-dependent symbols—is inversely related to its *computational* efficiency. Analyzing large collections of digital scores is thus hindered not only by the sheer volume of data involved, but also by the intrinsic complexity of the representations comprising musical structures. [5, 6]

To address such challenges, we present the Digital Musicology Corpus Analysis Toolkit (`DiMCAT`), which uses

dataframes [7–9] to disentangle pertinent score features within tabular representations, providing an interface for processing and analyzing large collections of dataframe-structured score data. `DiMCAT` supports MusicXML, MEI, Humdrum, and MuseScore (see Section 3.1), among other formats, and provides an expandable range of music analysis functionalities, including feature extraction, similarity analysis, and visualization. Addressed to Digital Musicology and MIR communities alike, its purpose is to provide a user-friendly interface for “distant-reading” staff-notated score corpora, and for utilizing score data in machine-learning pipelines. Efficiency at scale was among our primary design goals, an aspect which is only growing in importance as corpus sizes have continued to grow (e.g., [3]), with scores, rather than MIDI encodings, increasingly used to train large computational models (e.g., [10]).


In this paper, we describe the design and implementation of `DiMCAT` and argue for its usefulness through trials with various corpora. First, in Section 2, we outline a rationale for the representation of staff notation as dataframes, and for the underlying data-relational mindset. Section 3 presents the library design from a user’s perspective, covering such topics as data loading (Section 3.1) and the slice/group/analysis pipeline (Section 3.2). Evidence of ease-of-use in musicological research is provided in Section 4, and a comparison to extant libraries is made in Section 5.

2. UTILIZING DATAFRAMES TO REPRESENT SCORES

Dataframes were first introduced in 1990 as part of the statistical programming language S [7] and later ported to its descendant R [11]. Since then, they have become ubiquitous in the data science and machine learning communities, with a multitude of supplementary frameworks released across the spectrum of programming languages, often aiming to overcome performance problems associated with large dataframes (e.g., `modin` [12]). The wide adoption of dataframes can be attributed to their versatility, convenience, and operational principles, which resemble those of relational databases, spreadsheets, and nested arrays [9, 11–13]. `DiMCAT` encodes all score information within dataframe objects provided by either `pandas` or `modin`, with support for additional libraries (which we refer to as “backends”) planned in future versions.



© J. Hentschel, A. McLeod, Y. Rammos, and M. Rohrmeier. Licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). **Attribution:** J. Hentschel, A. McLeod, Y. Rammos, and M. Rohrmeier, “Introducing `DiMCAT` for processing and analyzing notated music on a very large scale”, in *Proc. of the 24th Int. Society for Music Information Retrieval Conf.*, Milan, Italy, 2023.



	qstamp	mc	mn	mc_onset	mn_onset	duration	duration_q	timesig	staff	voice	name	midi	tpc	octave
interval	<i>fraction</i>	<i>int</i>	<i>int</i>	<i>fraction</i>	<i>fraction</i>	<i>fraction</i>	<i>float</i>	<i>str</i>	<i>int</i>	<i>int</i>	<i>str</i>	<i>int</i>	<i>int</i>	<i>int</i>
[87.0, 87.25)	87	44	44	1/4	1/4	1/16	0.25	2/4	4	1	F2	41	-1	2
[87.0, 87.25)	87	44	44	1/4	1/4	1/16	0.25	2/4	3	1	A3	57	3	3
[87.0, 87.25)	87	44	44	1/4	1/4	1/16	0.25	2/4	2	1	C4	60	0	4
[87.0, 87.25)	87	44	44	1/4	1/4	1/16	0.25	2/4	1	1	F4	65	-1	4
[87.25, 87.5)	349/4	45	44	0	3/16	1/16	0.25	3/8	1	1	F4	65	-1	4
[87.25, 87.5)	349/4	45	44	0	3/16	1/16	0.25	3/8	2	1				
[87.25, 87.5)	349/4	45	44	0	3/16	1/16	0.25	3/8	3	1				
[87.25, 87.5)	349/4	45	44	0	3/16	1/16	0.25	3/8	4	1				
[87.5, 87.75)	175/2	45	44	1/16	1/4	1/16	0.25	3/8	1	1	D5	74	2	5
[87.5, 88.0)	175/2	45	44	1/16	1/4	1/8	0.5	3/8	2	1				
[87.5, 88.0)	175/2	45	44	1/16	1/4	1/8	0.5	3/8	3	1				
[87.5, 88.0)	175/2	45	44	1/16	1/4	1/8	0.5	3/8	4	1				
[87.75, 88.0)	351/4	45	44	1/8	5/16	1/16	0.25	3/8	1	1	C5	72	0	5

Table 1: Ludwig van Beethoven, *String quartet op. 18/6*, 4th movement (“La Malinconia”), measure number (**mn**) 44. The measure contains the section break after the slow introduction and is composed of two incomplete measure units with counts (**mc**) 44 and 45. The new 3/8 time signature (**timesig**) of the latter is introduced by a 3/16 upbeat, mathematically completing the 2/4 meter of the former. The dataframe represents notes and rests from beat 2 onwards. Its index (bold values on the left) comprises left-closed, right-open intervals which express the start and end points of each event on the score’s timeline, measured in quarter notes. Each column has a name (**bold**) and a data type (*italic*). The first eight columns contain temporal information (see Section 2.1). The columns **staff** and **voice** determine a notational layer. The last four columns express pitch-related information (**tpc** is tonal pitch class, expressed as the distance from C measured in perfect fifths) and are empty for rows representing rests. Special columns are omitted (e.g., ties, tremolos, or grace notes).

2.1 Representing staff notation as dataframes

Most encoding standards symbolically represent staff notation in hierarchical fashion. This includes most non-XML plaintext formats—at least those capable of encoding multiple staves, such as Lilypond, ABC, or Humdrum’s `**kern`—as well as XML-based standards. Table 2 shows a selection of tags in order of hierarchical nesting, from outermost to innermost, for three common XML-based standards. The table reveals that these standards recognize almost the same types of score elements, albeit located at different levels within the document tree. Among these elements are staves, measures, textural layers (‘voices’), chords (understood as groups of notes sharing a stem) and, finally, notes. For certain features, such as tempo and dynamic markings, the choice of hierarchical anchor is to some extent arbitrary: a tempo marking, for instance, might be attached to a specific measure or to a chord within that measure. DiMCAT’s approach to the unified modeling of diverse hierarchical representations consists in traversing them and grouping score elements of the same type in the same dataframe. This obviates mapping the particularities of each standard into a common score model, a process which would either inherit a degree of arbitrariness, or resort to error-prone estimations in order to eradicate it.¹

DiMCAT disentangles the underlying score hierarchy by grouping elements in five distinct categories, which we refer to as “facets”. These are:

- notes and rests (“events”, including ties, tremolos, grace notes, etc.)
- performance details (“control events”; tempo, dynamics, slurs, lyrics, articulation, etc.);
- measures (“flow control”; measure durations, staves, repeat indications, *fine*, etc.);

¹ Such a mapping, employed for example by the `music21` score processing library [14], is rather suitable when a complete model of the score needs to be maintained for further processing.

MuseScore	MEI	musicXML	
<Score>	<music> ...	<score-partwise>	<score-timewise>
<Staff>	(<part> ...)	<part>	<measure>
<Measure>	<measure> ...	<measure>	<part>
<Voice>	<staff> ...		<note>
<Chord>	<layer> ...	<staff>, <voice>	
<Note>	(<chord> ...)	<note>	

Table 2: Synopsis of XML tag hierarchies in three widespread XML-based score models. Models differ mainly in the placement and naming of score elements (<layer> being equivalent to <voice>). In the MEI column, ellipses (...) suggest that any number of hierarchical levels may be nested, and parentheses mark optional layers. MusicXML has two distinct organizational strategies (partwise or scorewise), which converge at the note level.

- analytical annotations (“labels”; chord changes, form labels, algorithmic outputs, etc.);
- metadata.²

A facet is the raw, original representation of a category of score elements from which specific, homogeneous “features” can be derived. In its simplest form, a feature is a subset of a facet in terms of rows and/or columns. For example, the `NotesAndRests` facet (shown in Table 1) comprises the `Rests` feature from which all rows and columns about notes have been removed. Other features offer variants requiring simple transformations. For instance, the `Notes` feature may be requested with tied note heads fused into single note events, with metrical weights added, or with pitches expressed as scale degrees (for an example, see Listing 2). Other features require more substantial computational analysis on a set of features or

² Note that visual details such as beaming are not loaded by default whenever they are deemed irrelevant for a distant-listening setting.

facets, and necessitate the invocation of an `Analyzer` (see Section 3.2.3).

Our approach thus projects different hierarchical score representations into a paradigm similar to that of relational databases. Structural relations previously expressed by the underlying score hierarchy are now expressed via IDs (for example, the columns ‘staff’, ‘voice’, and ‘mc’ in Table 1). In addition, all objects (except metadata) are unambiguously located on the score’s musical timeline by means of timestamps. As Table 1 shows, each facet and feature includes five columns expressing timestamps in three partially redundant ways. Timestamps expressed by means of ‘mc’ (the strict count of measure-like units from the beginning of the piece, regardless of their actual length or displayed measure number), along with ‘mc_onset’ (the location within a measure-like unit, represented as a fraction of a whole note) serve a crucial function. Given the actual durations of the measure-like units, ‘mc’ and ‘mc_onset’ determine ‘qstamp’, an object’s offset from the beginning of the piece in quarter notes. In addition, `DiMCAT` provides ‘mn’, the measure numbers actually found in score engravings, which are in principle non-unique and based on longstanding editorial conventions [15]. Analogously to ‘mc_onset’, these units warrant ‘mn_onset’ positions, required for computing metrical weights consistent with the respective meter (in the column ‘timesig’).

2.2 Operations on `DimcatResource` objects

To facilitate the processing and analysis of potentially large collections of notated music, `DiMCAT` aggregates facets (as well as features or analysis results) drawn from multiple pieces in a single dataframe, a `DimcatResource`. This approach enables vectorized operations on entire datasets, thus achieving higher performance in comparison to an equivalent sequence of single-dataframe operations. For additional speed when using very large corpora, `DiMCAT` can delegate dataframe operations to a distributed-computing backend such as `modin`, which allows for automatic partitioning and parallel processing [12]. `DimcatResources` natively serialize into ZIP archives accompanied by a Frictionless descriptor file [16] allowing for type-safe data validation and loading with external tools. Furthermore, the “frictionless” design allows `DiMCAT` to treat a descriptor file with its included metadata (column descriptions, file genesis, versioning information) as if it was the described resource itself, and to load the actual data into memory no earlier than required. The loaders described in the following section have the purpose of pre-processing the data to be analyzed, and storing it in a self-contained format that can also be easily served on the web.

3. LIBRARY DESIGN

This section describes `DiMCAT`’s design and API,³ which parallel familiar routines of musicological research: con-

³ The complete API and documentation can be found at <https://github.com/DCMLab/dimcat>. In addition to pydocs, we provide Jupyter Notebook-based interactive tutorials.

structing a corpus (loading and filtering, Section 3.1), organizing relevant corpus data (slicing and grouping), and running algorithms on this selection (analyzing, and plotting (Section 3.2)).

3.1 Loading Data

`DiMCAT` defines loaders which parse and store score data for a variety of symbolic encoding standards with the aid of external libraries.⁴ This is typically achieved by discovering the relevant files on disk or (from the web) and producing the homogeneous representation (the dataframes presented in Section 2.1) in parallelized fashion, while also compressing and storing the loaded data on disk for later use. Once data has been pre-processed and stored along with its metadata, `DiMCAT`’s default loader is capable of determining which score features are present, and of “lazily” loading them into memory whenever needed for processing. Apart from decreasing the memory footprint, this principle makes it possible to verify, before proceeding, whether the features required for a processing pipeline are actually available (see below). The extracted `Facet` objects remain by design as faithful as possible to the original data in terms of presence and naming of detected elements. Names and types of facet fields are standardized only in relation to the above-mentioned timeline columns, which are necessary for alignment. `Feature` objects, on the other hand, are comprehensively standardized upon extraction to guarantee type safety. Less specialized analyzers such as `Counters` also allow for the processing of `Facets`, and users who frequently work with custom `Features` drawn from nonstandard elements may contribute appropriate extensions to the codebase in the spirit of community-driven development.

Once loaded, the data is represented internally by the `Dataset` class and its various subclasses (see the complete documentation for details). `Dataset` objects are `DiMCAT`’s main drivers and the object type users interact with the most. They grant centralized access to all available dataframes (`Facets`, `Features`, and `Results`), depending on the current stage of computation. These objects in turn enable type-specific transformations and visualizations.

3.2 The Analysis Pipeline

Conceptually, every action performed by `DiMCAT` is a sequence of `PipelineStep` objects which, having been chained together, accept a `Dataset` object, perform a transformation or analysis on it, and return a new data object. In practice, this chaining is not entirely arbitrary, since some computations require data in specific formats (for example, the `CrossEntropy` analyzer requires equal-shaped probability vectors). All pipeline steps can be expressed as, and instantiated from, associative arrays of type `DimcatConfig` which are stored together with a `Dataset`’s descriptor to make the pipeline generation reproducible.

⁴ These currently include `ms3` [17] and `music21` [14], with `Verovio` [18] planned to be added in the future.

Each step in a `Pipeline` is fundamentally an instance of one of the following three classes⁵: `Slicers` accept data and partition it into chunks of various sizes—for example, sections between repeat signs, segments under the same guitar chord, or 8th-note-long slices. Slices never cross the boundaries of a piece of music. `Groupers` accept data and group it in formally specified categories—for example, segments with the same chord label, or pieces composed in the same decade. Unlike slices, groups often contain information from across the corpus. `Analyzers` are the heart of the library, performing the actual computation once the data has been sliced and grouped.

Many questions in music corpus studies involve comparisons between groups with a degree of commonality, e.g., between groups of pieces by the same composer, or of segments such as sonata development sections [19]. By combining slicers, groupers and analyzers, music researchers will find in `DiMCAT` an intuitive language for addressing pressing questions in the field.

3.2.1 Slicers

`Slicer` objects invoke one particular feature to compute segmentation boundaries. For example, a `NoteSlicer` invokes the `Notes` feature and stores its timestamps (e.g., note onset positions) as slice markers within the resulting `SlicedDataset`, interpreting them as time intervals. The newly returned dataset will slice any facet or feature subsequently requested, inserting an additional index level or column for slice boundaries (any element spanning over a slice boundary will be split or duplicated). In principle, any feature (e.g., double bar lines, dynamic indications, or the results of a key finder) can serve as the slicing criterion. The properties of the feature used as criterion can then be used for grouping the resulting slices (e.g., slicing a dataset using the results of a key-finding algorithm enables the subsequent grouping of slices by mode; see the following section).

3.2.2 Groupers

Applying a `Grouper` to a dataset is tantamount to binning pieces or slices based on a membership criterion. As a result, any facet or feature requested from a `GroupedDataset` is provided with a prepended index level of group identifiers. This enables both choosing a larger unit of analysis (by analyzing entire groups rather than each contained piece or slice, see the following section) and comparing groups of analysis results (for an example, see Section 4.2).

Frequently used groupers include the `CorpusGrouper`, the `StaffGrouper`, and the `ModeGrouper` (grouping pieces, events, and key slices, respectively). Groupers may also use metadata as criterion: for instance, the `YearGrouper` groups pieces based on their composition dates. Grouping is a computationally cheap operation because it is performed using dataframe indices.

⁵ That is without considering auxiliary pipeline steps such as `Writers` which never result in a different dataset type.

3.2.3 Analyzers

`Analyzers` are at the heart of the `DiMCAT` library. Based on its configuration, an analyzer will take one particular or all available features from the `Dataset`, perform the analysis on the minimal unit provided by the `Dataset` (slice or piece), and return an `AnalyzedDataset`. Results can be `Feature` objects (with timestamps) or `Result` objects (without timestamps), both of which are `DimcatResources` (see Section 2.2) and provide suitable methods for retrieving, displaying, transforming, and plotting analysis results. In addition, they allow the combination of piece or piece-slice analysis results into those corresponding to higher units of analysis, e.g., piece results into group results. This makes it possible, by applying several groupers to the same `AnalyzedDataset`, to regroup and recombine individual results. `DiMCAT` currently uses the `plotly` library for creating interactive plots and provides reasonable (but non-binding) defaults for combining grouped results in one figure. The main types of analyzers are `Counters`, `Comparisons` and `ClusterAnalyzers`, `Transformations`, and `RangeAnalyzers`.

The base `Analyzer` class is designed to be easily extensible; additional analyzers can be created by the community without knowledge of the deeper layers of the code. Contributors only need to understand the structure of the features that the new analyzer accepts as input, and select or implement the appropriate result type. Thereafter they implement the new object's serialization `Schema`⁶ and one of the methods that performs the actual analysis on a slice-, piece-, or group-specific dataframe. The method `combine()`, used for aggregating two result objects into one—for example, by adding result vectors—only needs to be implemented if no superclass is available to inherit it from. Optional methods include `check()` (for rejecting a dataset or feature if it doesn't fulfill certain criteria), `pre_process()` (for performing analyzer-specific feature transformations), and `post_process()` (for cleaning up the results object, for example by filling in missing values). A new analyzer constructed in this fashion is guaranteed to work with `DiMCAT`'s pipeline architecture.

The basic `Counter` counts the number of rows of any facet or feature (e.g., notes or chord labels). More versatile counters aggregate counts or durations based on the values contained in a given column (e.g., pitch classes), value combinations between several columns (e.g., pitch class–duration pairs), or n-grams (e.g., pairs of successive dynamic indications). Results can be transformed (e.g., by normalizing), various properties can be calculated and returned (e.g., the distributions' entropies), and plots can be generated.

`Comparison` analyzers perform pairwise comparisons on the slices, pieces, or groups represented by a given feature or result, such as the sliding-window autocorrelation of a feature's inter-onset-intervals, the Jaccard similarity between chord vocabularies, or the cross entropy between key profiles. They typically store their result as a

⁶ See details in the documentation.

confusion matrix, plotted by default as a heatmap. The results of comparisons lend themselves to subsequent application of `ClusterAnalyzers`, which use common algorithms such as k-means to compute groups that can reveal relations between the features under comparison [20, 21].

Transformations apply a function or fit a model in order to translate a feature into a different representation. Examples include analyzers that fit a Gaussian mixture model to a distribution, tokenize pitch events for use in a neural network, or transform pitch-class profiles into Fourier coefficients (as demonstrated in Section 4.1).

`RangeAnalyzers` are useful in cases where only minima and maxima (or the range) of numerical features are relevant. Examples include the line-of-fifths segment covered by a pitch class distribution [22] or the historical timespan covered by a dataset based on composition dates.

Finally, there are many specific analyzers, such as the `PitchClassVectors` analyzer featured in Section 4.1; they perform an analysis or transformation (here, aggregating durations) on one particular feature (here, `Notes`) under a range of specific configuration values (here, for example, type and format of pitch classes). A full list of analyzers is available in the documentation.

4. EXAMPLES

In this section, we present a few examples of musicological questions that can be easily answered using `DiMCAT` (provided the requisite data is available).

4.1 Fourier analysis of pitch class vectors

```

1 D = Dataset.load("debussy_piano")
2 D_analyzed = Pipeline(
3     [WindowSlicer(quarters_per_slice=1.0),
4     PitchClassVectors(),
5     DiscreteFourierTransform()])
6 ).process(D)
7 df = D_analyzed.get_result()
8 df.sample(5)

```

Listing 1: Pipeline for slicing a dataset by quarter-note windows, computing pitch class vectors and applying the Discrete Fourier Transform.

The Discrete Fourier Transform has seen frequent applications to musical structures, in particular pitch-class sets [23–28]. It belongs in a broader class of techniques which require, in our terms, a “slicing” of the score, such as a chordal reduction or a fixed-window segmentation. For example, as part of a corpus study using the Discrete Fourier Transform [29], `DiMCAT` was used to create enharmonic pitch class vectors for all 2-hand piano compositions by Claude Debussy. Listing 1 demonstrates the simplicity with which this analysis can be expressed as a `DiMCAT` pipeline. In the first line, the data is loaded from a local directory. Then the pipeline is created and processed. In the pipeline, a slicer first slices all pieces at every quarter note, then an analyser creates vectors of aggregated pitch class durations for each slice. Finally, the DFT analyser is run on each vector and the result obtained. A sample of

the results, with coefficients 0 through 6 given as complex numbers, is shown in Table 3.

4.2 Evaluating key segments

```

1 D = Dataset.load("dcml_corpora.datapackage.json")
2 D_sliced = Pipeline(
3     [KeySlicer(),
4     ModeGrouper()])
5 ).process(D)
6 F = DimcatConfig("Notes", format=SCALE_DEGREES)
7 D_sliced.get_feature(F).plot_grouped() # plot 1
8 D_grouped = CorpusGrouper().process(D_sliced)
9 D_grouped.plot_grouped() # plot 2

```

Listing 2: Plotting common dataset transformations (plotting parameters omitted). Plots shown in Figure 1.

Given a score dataset with local-key annotations created by human analysts or an automatic key finder, a researcher might wonder how key segments in the major and minor mode are distributed over the corpora contained in the dataset, and how the tonal pitch-class profiles compare between the two modes. This second example relies on a dataset that includes key annotations⁷ and demonstrates the power and ease-of-use of `DiMCAT` pipelines, even without using any analyzers. The `KeySlicer` used in Listing 2 is set up by default to slice the dataset by annotated modulations, and warns the user about pieces for which no local key information is available. The pipeline proceeds by applying the `ModeGrouper` to create one group per mode which, in the present dataset, amounts to a minor-key and a major-key group. Requesting the `Notes` feature from a `Dataset` processed in this fashion, we may retrieve and plot a representation that reflects the grouping, as shown in the upper bar chart in Figure 1. The lower plot demonstrates that the slicing criterion itself may also provide meaningful insights into a dataset. It can be produced by applying a `CorpusGrouper` to the processed `Dataset` and plotting the groups. (Without the corpus grouper, the plot would show the major-minor ratio for the entire dataset—that is, the result of the mode grouper).

5. COMPARISON WITH OTHER LIBRARIES

Other analysis libraries also lend themselves to the analysis of datasets of symbolic music encodings. In this section we compare `DiMCAT` to other open-source libraries which maintain note names (pitch spelling) and support multiple staves and analytical annotations. Several among them likewise utilize dataframes.

The Humdrum Toolkit was one of the first frameworks for computer-aided music analysis, and is still used for the analysis of Humdrum and kern files across the range of programming languages to which it has been ported. The R package `humdrumR` [30] ports the Humdrum Toolkit into R. While it provides support for computationally efficient dataframes, and includes R’s inherent plotting capabilities, like Humdrum itself it cannot import more modern, and arguably more common, symbolic-encoding formats such as musicXML without the use of error-prone converters.

⁷ https://github.com/DCMLab/dcml_corpora

corpus	fname	1.0q_slice	0	1	2	3	4	5	6
debussy_other_piano_pieces	1068_reverie	[351.0, 352.0)	2.00+0.00j	0.32-0.18j	-0.50-0.87j	-1.50+0.50j	-1.00+0.00j	1.18+0.68j	1.00+0.00j
debussy_childrens_corner	1113-03_childrens_serenade	[27.0, 28.0)	2.25+0.00j	0.92-0.47j	0.37-0.22j	0.75-0.50j	-1.12-0.65j	-1.67-0.03j	-0.75+0.00j
debussy_preludes	1123-12_preludes_feux	[176.0, 177.0)	6.62+0.00j	0.53-1.57j	-1.94-0.11j	0.00+4.12j	2.69-2.27j	2.47-3.30j	-2.12+0.00j
debussy_etudes	1136-04_etudes_sixtes	[53.5, 54.5)	4.00+0.00j	-0.12+0.37j	-0.75+0.00j	-0.75-0.25j	-1.25+0.87j	1.62-1.37j	1.50+0.00j
debussy_deux_arabesques	1066-02_arabesques_deuxieme	[137.0, 138.0)	4.00+0.00j	0.25-0.30j	-0.25-1.30j	-2.00-1.00j	1.75+1.30j	0.25+2.30j	2.00+0.00j

Table 3: Sample rows from a dataframe containing the seven first DFT coefficients gained from quarter-note-window pitch class vectors. The first three columns represent a multi-index indicating corpus, file name, and slice interval (expressed as `qstamp`, see Table 1); they make it possible to trace the result back to the score.

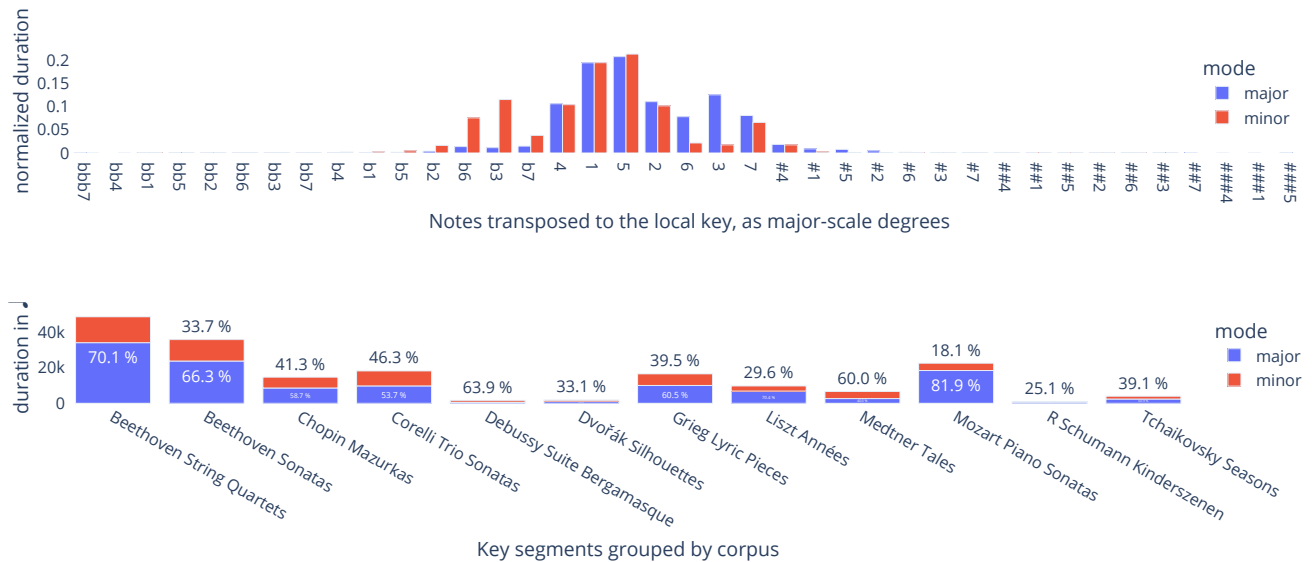


Figure 1: The two plots produced by the code shown in Listing 2.

MUSIC21 [14] is a large Python library capable of importing all relevant music formats, transforming them into a comprehensive hierarchical model of the score. Relying on an elaborate object model, it provides methods for creating and manipulating the elements of a music score. However, its design renders it computationally demanding [30, 31] for large corpora, and it provides only few methods designed specifically for corpus analysis.

Several Python libraries follow a similar approach to DiMCAT’s, analyzing and making available score information in the form of dataframes. These include the VIS-framework [32] and CRIM intervals [33] (both focusing on intervallic successions and sonorities), CAMAT [31] (basic pitch statistics), and `musicif` [34] (with a focus on global features of entire scores). Among them, only [31] introduces its own score parser (for MusicXML), with the remaining ones invoking `music21`. [34] also includes the MuseScore parser `ms3` [17] and therefore exposes an architecture that is as easily extensible as ours.

Although DiMCAT can, in principle, provide any algorithm that operates on successions or sets of pitch events, its focus on “distant listening” makes it less suited for close-reading studies than some of the aforementioned alternatives. Indeed, its distinguishing feature is the newly introduced *Slice-Group-Analyze* paradigm, designed for inquiries in which the corpus, rather than the individual score, is the primary research object. To this end, DiMCAT

provides mechanisms for studying the statistical properties of potentially very large amounts of musical material by iteratively applying sequences of segmentation and grouping algorithms with a high degree of combinatorial freedom. From this point of view, the “slice” serves as an additional operational level between “note” [31–33] and “piece” [34].

6. CONCLUSION

In this paper we have introduced DiMCAT, a Python library capable of parsing, transforming, and analyzing annotated music score data in a range of symbolic-encoding formats, and to do so efficiently at scale. The library stores data as dataframes, a ubiquitous structure in the fields of digital humanities and data science. DiMCAT emphasizes traceability (results can reliably lead to the original score elements) and reproducibility (version identifiers are systematically applied to code and data). Thanks to an interface that masks its inner workings, the functionality of the library is usable and extensible by musicologists with limited programming experience.

DiMCAT is released under the GPL-3.0-or-later License, and we intend to continue adding further music analyzers, inviting feedback, requests, and contributions from the community.

7. ACKNOWLEDGEMENTS

This research was supported by the Swiss National Science Foundation within the project “Distant Listening – The Development of Harmony over Three Centuries (1700–2000)” (Grant no. 182811). This project is being conducted at the Latour Chair in Digital and Cognitive Musicology, generously funded by Mr. Claude Latour.

8. REFERENCES

- [1] C. S. Sapp, “Online database of scores in the humdrum file format,” in *Proceedings of the Proceedings of the 9th International Society for Music Information Retrieval Conference (ISMIR)*, Philadelphia, 2008.
- [2] M. Gotham, P. Jonas, B. Bower, W. Bosworth, D. Rootham, and L. VanHandel, “Scores of Scores: An OpenScore project to encode and share sheet music,” in *Proceedings of the 5th International Conference on Digital Libraries for Musicology - DLfM '18*. Paris, France: ACM Press, 2018, pp. 87–95.
- [3] F. Foscarin, A. McLeod, P. Rigaux, F. Jacquemard, and M. Sakai, “ASAP: A dataset of aligned scores and performances for piano transcription,” in *Proceedings of the 21st International Society for Music Information Retrieval Conference (ISMIR)*, 2020.
- [4] J. Hentschel, Y. Rammos, M. Neuwirth, F. C. Moss, and M. Rohrmeier, “An annotated corpus of tonal piano music from the long 19th century,” *Empirical Musicology Review*, forthcoming.
- [5] J. Devaney, “Using note-level music encodings to facilitate interdisciplinary research on human engagement with music,” *Transactions of the International Society for Music Information Retrieval*, vol. 3, no. 1, pp. 205–217, Oct. 2020.
- [6] F. Foscarin, P. Rigaux, and V. Thion, “Data quality assessment in digital score libraries: The GioQoso Project,” *International Journal on Digital Libraries*, vol. 22, no. 2, pp. 159–173, Jun. 2021.
- [7] J. Chambers, T. Hastie, and D. Pregibon, “Statistical models in S,” in *Proceedings in Computational Statistics*, K. Momirović and V. Mildner, Eds. Heidelberg: Physica-Verlag HD, 1990, pp. 317–321.
- [8] W. McKinney, “Data structures for statistical computing in python,” in *Proceedings of the 9th Python in Science Conference*, Austin, Texas, 2010, pp. 56–61.
- [9] D. Petersohn, “Dataframe systems: Theory, architecture, and implementation,” Ph.D. dissertation, University of California, Berkeley, 2021.
- [10] A. Mcleod and M. A. Rohrmeier, “A modular system for the harmonic analysis of musical scores using a large vocabulary,” in *Proceedings of the 22nd International Society for Music Information Retrieval Conference (ISMIR)*. Online: ISMIR, Nov. 2021, pp. 435–442.
- [11] D. Petersohn, S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran, “Towards scalable dataframe systems,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2033–2046, Aug. 2020.
- [12] D. Petersohn, D. Tang, R. Durrani, A. Melik-Adamyanyan, J. E. Gonzalez, A. D. Joseph, and A. G. Parameswaran, “Flexible rule-based decomposition and metadata independence in modin: A parallel dataframe system,” *Proceedings of the VLDB Endowment*, vol. 15, no. 3, pp. 739–751, Nov. 2021.
- [13] Y. Wu, “Is a dataframe just a table?” in *10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019)*, ser. OpenAccess Series in Informatics (OASISs), S. Chasins, E. L. Glassman, and J. Sunshine, Eds., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, pp. 6:1–6:10.
- [14] M. S. Cuthbert, “Music21: A toolkit for computer-aided musicology and symbolic music data,” in *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR)*, 2010, pp. 637–642.
- [15] E. Gould, *Behind Bars: The Definitive Guide to Music Notation*. London: Faber Music, 2011.
- [16] D. Fowler, J. Barratt, and P. Walsh, “Frictionless data: Making research data quality visible,” *International Journal of Digital Curation*, vol. 12, no. 2, pp. 274–285, May 2018.
- [17] J. Hentschel and M. Rohrmeier, “Ms3: A parser for MuseScore files, serving as data factory for annotated music corpora,” *Journal of Open Source Software*, 2023.
- [18] L. Pugin, R. Zitellini, and P. Roland, “Verovio: A library for engraving MEI music notation into SVG,” in *PugiProceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR)*, 2014, pp. 107–112.
- [19] C. White, *The Music in the Data: Corpus Analysis, Music Analysis, and Tonal Traditions*, 1st ed. New York: Routledge, Nov. 2022.
- [20] R. Cilibrasi, P. Vitanyi, and R. de Wolf, “Algorithmic clustering of music,” *arXiv:cs/0303025*, Mar. 2003.
- [21] E. Anzuoni, S. Ayhan, F. Dutto, A. McLeod, F. C. Moss, and M. Rohrmeier, “A historical analysis of harmonic progressions using chord embeddings,” in *Sound and Music Computing Conference (SMC)*, 2021, pp. 284–291.
- [22] F. C. Moss, M. Neuwirth, and M. Rohrmeier, “The line of fifths and the co-evolution of tonal pitch-classes,” *Journal of Mathematics and Music*, pp. 1–25, Mar. 2022.

- [23] D. Lewin, “Re: Intervallic relations between two collections of notes,” *Journal of Music Theory*, vol. 3, no. 2, pp. 298–301, Nov. 1959.
- [24] I. Quinn, “A unified theory of chord quality in equal temperaments,” Ph.D. dissertation, Eastman School of Music, Rochester, New York, 2004.
- [25] D. Tymoczko, “Set-class similarity, voice eading, and the Fourier transform,” *Journal of Music Theory*, vol. 52, no. 2, pp. 251–272, Sep. 2008.
- [26] J. Yust, “Applications of DFT to the theory of twentieth-century harmony,” in *Mathematics and Computation in Music*, T. Collins, D. Meredith, and A. Volk, Eds. Cham: Springer International Publishing, 2015, vol. 9110, pp. 207–218.
- [27] E. Amiot, *Music through Fourier Space*, ser. Computational Music Science. Cham: Springer International Publishing, 2016.
- [28] J. D. Harding, “Applications of the Discrete Fourier Transform to music analysis,” Ph.D. dissertation, Florida State University, 2021.
- [29] S. Laneve, L. Schaerf, G. Cecchetti, J. Hentschel, and M. Rohrmeier, “The diachronic development of Debussy’s musical style: A corpus study with Discrete Fourier Transform,” *Humanities and Social Sciences Communications*, vol. 10, no. 1, p. 289, Jun. 2023.
- [30] N. Condit-Schultz and C. Arthur, “humdrumR: A new take on an old approach to computational musicology,” in *Proceedings of the 20th International Society for Music Information Retrieval Conference (ISMIR)*, Delft, 2019.
- [31] E. Poliakov and C. R. Nadar, “CAMAT: Computer Assisted Music Analysis Toolkit,” in *Proceedings of the Digital Music Research Network One-day Workshop (DMRN+16)*, 2021, p. 12.
- [32] C. Antila and J. Cumming, “The VIS framework. Analyzing counterpoint in large datasets,” in *Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR)*, 2014.
- [33] R. Freedman, A. Morgan, Gould, O. Shostak, T. Dang, A. Janco, D. Russo-Batterham, E. Leon, and H. West, “CRIM intervals,” 2023. [Online]. Available: <https://github.com/HCDigitalScholarship/intervals>
- [34] A. Llorens, F. Simonetta, M. Serrano, and Á. Torrente, “Musif: A Python package for symbolic music feature extraction,” in *Sound and Music Computing Conference (SMC)*. arXiv, Jul. 2023.