Mol Cell Proteomics Papers in Press. Published on December 18, 2018 as Manuscript RP118.001070

psims - A declarative writer for mzML and mzIdentML for Python

Joshua Klein[1] and Joseph Zaia[1,2,*]

[1]Program for Bioinformatics, Boston University
[2]Department of Biochemistry, Boston University
[*]Corresponding author: Joseph Zaia, jzaia@bu.edu

December 17, 2018

Running Title: psims - A writer for mzML and mzIdentML for Python

Abstract

Summary: mzML and mzIdentML are commonly used, powerful tools for representing mass spectrometry data and derived identification information. These formats are complex, requiring non-trivial logic to translate data into the appropriate representation. Most published implementations are tightly coupled to data structures. The most complete implementations are written in compiled languages which cannot expose the complete flexibility of the implementation to external programs or bindings. To our knowledge, there are no complete implementations for mzML or mzIdentML available to scripting languages like Python or R. We present psims, a library written in Python for writing mzML and mzIdentML. The library allows writing either XML format using built-in Python data structures. It includes a controlled vocabulary resolution system to simplify the encoding process and an identity tracking system to manage entity relationships. The source code is available at https://github.com mobiusklein/psims, and through the Python Package Index as psims, licensed under the Apache 2 common license.

1 Introduction

The proliferation of data processing and identification methods in mass spectrometry has led to ever increasing complexity for tools that need to describe their results. Over the last decade and a half, the community-driven XML standards for representing spectral data, mzML [1], and peptide/protein identification, mzIdentML [2], have become core to computational methods development [3]. These formats combine a complex XML schema for defining the structure of the information contained with a flexible vocabulary of terms for describing the contents [4].

While this combination makes these tools powerful, it comes at the cost of increased complexity in implementation. There are a small number of comprehensive C/C++ implementations for these formats, the dominant two being `ProteoWizard` [5] and `OpenMS` [6], and Java implementations including `ms-data-core-api/jmzML` and `MSFTBX` [7, 8], among other staticly-typed compiled languages. In higher-level languages, readers of these formats are often implemented directly on top of generic XML parsing libraries [9, 10] or as bindings for the implementation in a lower level language [11, 12]. When readers are implemented, often writers are implemented as being templated on some read-in data, or not implemented at all. These libraries cannot be used to generate files "from native data", which means that users cannot easily write tools to produce these file formats because the data to produce are not immediately connected with an input template. For implementations using bindings on lower level languages, there are limits on the range of constructs that can be expressed, such as the inability to describe a new complex structure which is not part of the lower level library's bindings or type system. This may be due to a violation of the single responsibility principle, or intentionally covering only those parts of the specification which are considered useful by the implementors. To our knowledge, the only template-free writers are found in the above listed C++ libraries and in several published and unpublished Java or C# implementations. To write mzML and mzIdentML files from native data using a scripting language like Python or R, one must use bindings `OpenMS` through `pyOpenMS`[11] and `ProteoWizard` through `mzR`[12] respectively. This adds unecessary complexity to the creation and deployment of mass spectral data processing tools.

We implemented a declarative writer for mzML 1.1 and mzIdentML 1.2 in Python, with a wider range of expression than previously available implementations for the language[11, 12]. The document writers use context managers to express the logical nesting of components, dynamically translates common Python types, and tracks referential integrity of the document for the user. The document is generated as a stream of enclosing contexts which can only influence actions taken inside themselves, making document construction

referentially transparent. Our writers automatically resolve controlled vocabulary terms by name or accession number, including unit information where appropriate. These writers store only the identity map for maintaining referential integrity, decoupling document size from the bounds of available memory by writing the document incrementally as elements are attached to the document writer. This library includes the option to perform validation against the appropriate XML schema, but validation is not made mandatory. This allows its users to provide additional information to downstream applications if both have an agreed upon structure to parse, extending the document format used, if needed. The decision to step outside the defined schema should not be made lightly, as this may make the file unusuable by other software. Additionally, semantic validation has not been implemented, as this functionality is available through stand-alone tools in OpenMS [6] as well as those provided with the mzML and mzIdentML format specification [1, 2].

The library, `psims` was written using `lxml`, a powerful XML handling library [13] which implements incremental document writing, NumPy [14] to handle binary data arrays, and SQLAlchemy [15] for relational schema representation in some cases where the semantic term graph was not applicable. We demonstrate examples using Pyteomics [9] and it is used extensively in our test suite, providing readers for both mzML and mzIdentML. `psims` is available under the Apache 2 common license and its source code is freely available at https://github.com/mobiusklein/psims as is its documentation. The library is compatible with both Python 2 and Python 3 across platforms.

The only alternative to this library within Python is to use `pyOpenMS`, Python bindings generated for the excellent C++ `OpenMS` library. While the underlying data model in `OpenMS` may be as flexible within the standard as ours, the Python-level API was only able to expose some of this after release 2.2. Even with this, the user cannot extend the format, being able to only represent those entities which have a mapping to an `OpenMS` data structure.

2 Methods

2.1 Novel Input Data Preparation

To demonstrate the facility of the library, we used previously published data from PXD003498. We downloaded `AGP-tryp_016_1.raw` and `AGP-tryp_016_1.mgf` from PRIDE. We converted `AGP-tryp_016_1.raw` to mzXML using Proteowizard [5] with peak picking. Next we used `MS-Deconv` [16] to process the mzXML file, using its text output format including MS1 information. Using the example script `convert_msdeconv.py` to convert the `MS-Deconv` output into mzML format.

We used `IdentiPy`[17] (commit b89a19e) to search `AGP-tryp_016_1.mgf` against the Human proteome downloaded from UniProt Release 2017_09 [18] with fixed Carbamidomethyl at Cysteine and variable Oxidation at Methionine. The search was done with trypsin with two missed cleavages. We used 10 ppm precursor mass accuracy and 0.1 Da production mass accuracy. The output was written to pickle format. We used the example `identipy_mzid.py` to convert the pickle file into mzIdentML format.

2.2 Generality Test

To demonstrate generality, we constructed two scripts for reading in mzML and mzIdentML files from and output the same document semantically unchanged. We detected material differences in the generated XML by performing a recursive comparison of the parsed data from each source file and its duplicated pair. We assessed semantic difference manually from any detected material differences.

To test the mzML writer, we included a script `transform_mzml.py` (see supplementary information) which read in an mzML document using `pyteomics`, then translated the extracted data back into mzML, followed by XSD validation and a test for content equivalence. We ran this test on `small.mzML`, an mzML file produced by ProteoWizard from the original mzML specification test data, and on `only_ms2_mzml.mzML` produced by Bruker's `CompassXport`. This translation process included appropriate updating of the `softwareList` and `dataProcessingList` to include the additional processing stepsincluded in the translation procedure. Validation was performed using OpenMS v2.3.0 [6].

To test the mzIdentML writer, we included a script `transform_mzid.py` (see supplementary information) which read in an mzIdentML document using `pyteomics`, then translated the extracted data back into mzIdentML followed by XSD validation and a test for content equivalence. We ran this test on several mzIdentML 1.1 and 1.2 files from both the mzIdentML specification example collection and from commercial tools, listed in Table 1. This includes sample files which contain cross-linked peptide identifications and multiple search engines to demonstrate support for new features in the 1.2 version of the standard. Validation was performed using the official mzIdentML Validator [2].

Example File Name	Format	Producer	File Source
small.mzML[1]	mzML 1.1.0	ProteoWizard	proteowizard.sourceforge.net/
only_ms2_mzml.mzML	mzML 1.1.0	Bruker CompassXport	
combined_fdr_1.2.mzd[1]	mzIdentML 1.2.0	mzidLib	HUPO-PSI/mzIdentML
MSGFplus_tandem.pia.1.2.mzid	mzIdentML 1.2.0	PIA	HUPO-PSI/mzIdentML
xiFDR-CrossLinkExample_single_run.mzid	mzIdentML 1.2.0	xi	HUPO-PSI/mzIdentML
Mudpit_170119_O2_P8_SG_StM_G1_1.mzid	mzIdentML 1.1.0	Scaffold	PXD008280
2017-10-03-MEM-AGP-0016.raw_20171220_Byonic.mzid[1]	mzIdentML 1.1.0	Byonic	
AGP_Proteomics2.mzid	mzIdentML 1.1.0	PEAKS Studio	

[1] Original File Validated

Table 1: Generality Validation Sources

2.3 Vocabulary Resolution

To enable automatic derivation of controlled vocabulary terms, we first loaded each CV from either a URI specified by the user during document initialization, or a static file included with the library's source code during packaging if the network request fails. The library includes a configurable caching mechanism to choose to store the vocabulary source files on the file system after first retrieving them from their providers or from the installed package. With the exception of Unimod [19], we parsed each CV in OBO format [20], constructing a term graph in memory. We parsed Unimod's database from the XML export of their relational database. The URIs used to obtain the files included in the packaged distribution are shown in Table 3. Next, we indexed each term graph by accession number i.e. `"MS:1000016"` or `"UNIMOD:4"`, by human readable name case insensitively, and by synonym case insensitively. We next identified three idiomatic ways to specify a controlled vocabulary term:

1. By single identifier for a term which has no associated value, such as `"MSn spectrum"` or `"ms-ms search"`. This naturally maps to string value.
2. By paired identifier and a value which has no unit, such as `"ms level"` and an integer or `"peptide sequence-level e-value"` and a float. This naturally maps to a `Mapping` with a single key-value pair like a dictionary, or a `Sequence` of two values like a tuple.
3. By a fully specified term including an explicit name, value, CV of origin, and optionally a unitidentifier and the unit's CV of origin, such as `"scan start time"` with a float denoting the value, a quantity of time and an identifier of the unit, such as `"minute"` or `"UO:0000031"`. This maps most naturally to a `Mapping` type with as many relevant keys specified as possible.

Whenever the document writer would encounter an argument to be interpreted as either a `<cvParam/>` or `<userParam/>`, it first determined which of the three idioms was being used, and then queried the controlled vocabulary index in descending order of specificity for each CV, first accession number, then natural name, then synonym until a match was found or no indices remained. See Table 2 for example term mappings. If a match was found, a controlled vocabulary term was resolved and a `<cvParam/>` tag would be rendered to the document, otherwise a `<userParam/>` would be rendered including with all the fields the user specified. To support mzML's `<referenceableParamGroup/>` feature, we included a final case, if the provided argument explicitly matched the identifier of a previously registered parameter group, a `<referenceableParamGroupRef/>` would be rendered instead.

2.3.1 Unit Resolution

When the document writer resolved a controlled vocabulary term, if the term definition specified that the term had a unit through the relationship `has_unit`, the unit would also be resolved. If only one unit was specified in the controlled vocabulary and the user provided data describing the term did not provide any unit information, the term's defined unit would be used automatically. If multiple units were permitted, as is the

ID	Name	Synonyms
MS:1000128	profile spectrum	continuous mass spectrum, Continuum Mass Spectrum
MS:1000422	beam-type collision-induced dissociation	HCD
UNIMOD:7	Deamidation	Deamidated, Citrullination, phenyllactyl from N-term Phe
UNIMOD:41	Hex	Hexose, Glucose, Galactose, Mannose, Fructose

Table 2: Controlled Vocabulary Synonym Mapping Examples. Any term in a row would map to the same entity case-insensitively.

Controlled Vocabulary Name	File Name	URI
PSI-MS	psi-ms.obo	https://raw.githubusercontent.com/HUPO-PSI/psi-ms-CV/master/psi-ms.obo
UNIT	unit.obo	http://ontologies.berkeleybop.org/uo.obo
PATO	pato.obo	http://ontologies.berkeleybop.org/pato.obo
XLMOD	XLMOD.obo	https://raw.githubusercontent.com/HUPO-PSI/mzIdentML/master/cv/XLMOD.obo
UNIMOD	unimod_tables.xml	http://www.unimod.org/xml/unimod_tables.xml

Table 3: Controlled Vocabulary Sources

case for quantities such as time or signal intensity, instead a warning would be generated and the first unit option would be used.

2.4 Component Definition

To avoid locking the document writer into a fixed system forcing an element to be mapped to a single class, XML elements could be created as free tags using the writer's `element` method which creates a context manager wrapping a light weight element object, or using a component type. Component types are fully specified classes, laying out attributes, inter-element relationships, and derived property detection. A component is bound to a document context providing it with access to controlled vocabularies and the document's identity map. Each component constructs one containing element to enclose its contents, but may create multiple inner elements or components automatically, bound to the same document context. Both element and component types implement context managers so they may be arbitrarily nested and mixed.

2.5 Referential Integrity

Each time the document writer constructs an element, it attempts to create an identifier for it for the `id` attribute. If an element was constructed with am integer identifier, in order to ensure that the identifier is unique, it instead assigned an identifer `[TAG_NAME]_[INTEGER]`. If the provided identifier was not an integer, it would be used verbatim. If the element was part of the top-level element of an component, it would register a mapping between the user provided identifier and the generated identifier specific to that component type. When constructing an component which references another component, the user would provide the same identifier from their application, and the document writer would map this to the generated identifier, maintaining referential integrity.

If a provided identifier has not been registered prior to being referenced, the document writer would emit a warning to indicate that the reference is missing, though this may be made an error during writer configuration. To avoid generating warnings caused by document ordering elements referencing other elements that have not been written yet, identifiers may be registered independent of element construction with the `register` method of the document writer. Both `register` and requesting a missing identifier update the referential integrity map using the same procedure as described during automatic registration, maintaining the same behavior independent of whether an identifier is referenced first or registered first. This makes the registration and query process stateless, and serves only as an error checking service.

2.6 Streaming and Referential Transparency

Each component object used by the document writer was defined by a class. Each object upon initialization would configure itself and the XML element it represented, register with the referential integrity map of the document, and then await serialization. The component types may call their `write` method on an XML file stream to serialize themselves and all their child elements, just as in other XML serialization techniques, but this requires first constructing all child elements in memory prior to serialization. Components may also be used as context managers with an XML file stream, writing the component's opening tag and any content appropriate to their type before yielding control to any nested statements, and finally writes any trailing content such as a list of `<cvParam/>`s and the component's closing tag. This allows callers to load data into memory incrementally, construct an XML component associated with that data, serialize it, and then releasethat data from memory.

This pattern also encourages writing code that is referentially transparent, where components are constructed and immediately begin their context, any child elements are incrementally loaded, begun, and ended, before ending themselves. This makes the component effectively immutable, as previously stated when a component is construct and it queries the referential integrity map no tangible state is changed. It also conveys the relationship that any logic within the context manager opened by a component is within that component without constructing the sequence of child components before opening that context. The file writing process we described is declarative by virtue of its referential transparency, it is not monadic, and file operations are performed immediately. Furthermore, the caller is allowed to use component objects imperatively, mutating their state repeatedly prior to writing them to the output stream or beginning their context, though the guarantee remains that once begun the component will remain unchanged.

[Suggested Location for Figure 1]

3 Results and Discussion

To demonstrate the facility of making these formats producible by scripting, we translated the result of `MS-Deconv` [16] from their tab-delimited text format into mzML, preserving as much source file metadata as possible while retaining the deisotoped and charge deconvoluted peak masses and intensities (Figure 1). We also retained `MS-Deconv`'s recalculated precursor monoisotopic m/z and charge fields which can correct the precursor monoisotopic peak errors reported by the instrument. The generated mzML file retained the metadata read from the original mzXML file that was passed to `MS-Deconv` and can be viewed using any tool which can read mzML, such as SeeMS [5]. Additionally, during the translation process, we included a set of calculations to look for specific signature ions and include a `<userParam/>` tag on each MS^2 scan which contains a desired m/z value.

The `convert_msdeconv.py` script (see supplementary information) for the workflow described in Figure 1 also demonstrates the controlled vocabulary resolution process, recognizing partial specifications by name or accession, and packing the structured data into fully specified `<cvParam/>` tags. In some cases `<cvParam/>` were automatically derived from other parameters, such as the redundancy between `"MSn spectrum"` and `"ms level"`. Finally, the script validates the generated mzML document against the schema definition to ensure each element has all required information and contain the expected elements in the correct order.

We also show that this library can be useful for more complex applications written in Python themselves, including the `identipy_mzid.py` script defining a process for writing the results of `IdentiPy` to an mzIdentML file (see supplementary information). This script exercises the identifier resolution system to keep track of entities which have no natural identifier or universally unique identifier, and further demonstrates the controlled vocabulary resolution process. As `IdentiPy` does not use external controlled vocabulary names for post-translational modifications, we include a modification identity resolution procedure to infer Unimod names from the user-provided mass and amino acid specificities, and only emit `"unknown modification"` when no mapping may be found. `IdentiPy` does not perform protein inference, so no `<ProteinDetection>` or related elements were produced. As with many mzIdentML writers, during the document writing process, the current locations and identifiers of connected external data were recorded in the `<Inputs>` element, recording the database and spectral data file used. The native identifiers of the spectra as recorded by `IdentiPy` were included in the `<SpectrumIdentificationResult>` element's `spectrumID` attribute. It is compatible with data integration tools like those provided by PRIDE for reconstructing identifications.

We demonstrate generality of the representation and serialization process by parsing data files in mzML 1.1, mzIdentML 1.1, and mzIdentML 1.2 format and re-writing that data back to disk unchanged. In all cases where the original document was structurally and semantically valid the resulting file were alsovalid. Where manual assessment was necessary, it was caused by the original file leaving an attribute indeterminant or unspecified while we reserialized it explicitly. We assert this shows that a wide variety of workflows and data can be serialized by this library.

4 Conclusion

We presented `psims`, a library for writing HUPO-PSI standard mzML and mzIdentML and for interacting with the associated controlled vocabularies in Python. We briefly demonstrated its utility for writing an mzML file from tabular data produced by an external tool that consumes raw spectra but cannot return the data to a metadata-rich format, permitting the re-integration of these methods into environments which depend upon this extra information to interpret the described data appropriately. With the increasing number of peptide and protein identification tools including `IdentiPy` and `Ursgal`[21] being written directly in Python, it may be advantageous for future tools to directly write their results to mzIdentML.

`psims` does not use a DOM-like architecture but it fully supports the published XML schemata and can use them to validate its own generated XML. The library does not strictly enforce the XSD validation at runtime due to its streaming nature, so the user must explicitly request it. We also wanted to provide a flexible system that enables users to write documents that meet their needs, which may not yet be covered by the standard. The ability to experiment with alternative constructs might also lead to better proposals.

5 Acknowledgments

This work was supported by National Institute of Health Grant U01CA221234

References

[1] Lennart Martens, Matthew Chambers, Marc Sturm, Darren Kessner, Fredrik Levander, Jim Shofstahl, Wilfred H Tang, Andreas Römpp, Steffen Neumann, Angel D Pizarro, Luisa Montecchi-Palazzi, Natalie Tasman, Mike Coleman, Florian Reisinger, Puneet Souda, Henning Hermjakob, Pierre-alain Binz, and Eric W Deutsch. mzML–a community standard for mass spectrometry data. *Molecular & cellular proteomics : MCP*, 10(1):R110.000133, 2011.

[2] Juan Antonio Vizcaíno, Gerhard Mayer, Simon R Perkins, Harald Barsnes, Marc Vaudel, Yasset Perez-Riverol, Tobias Ternent, Julian Uszkoreit, Martin Eisenacher, Lutz Fischer, Juri Rappsilber, Eugen Netz, Mathias Walzer, Oliver Kohlbacher, Alexander Leitner, Robert J Chalkley, Fawaz Ghali, Salvador Martínez-Bartolomé, Eric W Deutsch, and Andrew R Jones. The mzIdentML data standard version 1.2, supporting advances in proteome informatics. *Molecular & cellular proteomics : MCP*, page mcp.M117.068429, may 2017.

[3] Eric W. Deutsch, Sandra Orchard, Pierre Alain Binz, Wout Bittremieux, Martin Eisenacher, Henning Hermjakob, Shin Kawano, Henry Lam, Gerhard Mayer, Gerben Menschaert, Yasset Perez-Riverol, Reza M. Salek, David L. Tabb, Stefan Tenzer, Juan Antonio Vizcaíno, Mathias Walzer, and Andrew R. Jones. Proteomics Standards Initiative: Fifteen Years of Progress and Future Work. *Journal of Proteome Research*, 16(12):4288–4298, 2017.

[4] Gerhard Mayer, Andrew R Jones, Pierre-Alain Binz, Eric W Deutsch, Sandra Orchard, Luisa Montecchi-Palazzi, Juan Antonio Vizcaíno, Henning Hermjakob, David Oveillero, Randall Julian, Christian Stephan, Helmut E Meyer, and Martin Eisenacher. Controlled vocabularies and ontologies in proteomics: overview, principles and practice. *Biochimica et biophysica acta*, 1844(1 Pt A):98–107, jan 2014.

[5] Darren Kessner, Matt Chambers, Robert Burke, David Agus, and Parag Mallick. ProteoWizard: Open source software for rapid proteomics tools development. *Bioinformatics*, 24(21):2534–2536, nov 2008.

[6] Hannes L Rost, Timo Sachsenberg, Stephan Aiche, Chris Bielow, Hendrik Weisser, Fabian Aicheler, Sandro Andreotti, Hans-Christian Ehrlich, Petra Gutenbrunner, Erhan Kenar, Xiao Liang, Sven Nahnsen, Lars Nilse, Julianus Pfeuffer, George Rosenberger, Marc Rurik, Uwe Schmitt, Johannes Veit, Mathias Walzer, David Wojnar, Witold E Wolski, Oliver Schilling, Jyoti S Choudhary, Lars Malmstrom, Ruedi Aebersold, Knut Reinert, and Oliver Kohlbacher. OpenMS: a flexible open-source software platform for mass spectrometry data analysis. *Nat Meth*, 13(9):741–748, 2016.

[7] Yasset Perez-Riverol, Julian Uszkoreit, Aniel Sanchez, Tobias Ternent, Noemi Del Toro, Henning Hermjakob, Juan Antonio Vizcaíno, and Rui Wang. Ms-Data-Core-Api: An open-source, metadata-oriented library for computational proteomics. *Bioinformatics*, 31(17):2903–2905, 2015.

[8] Dmitry M Avtonomov, Alexander Raskind, and Alexey I Nesvizhskii. BatMass: a Java Software Platform for LC MS Data Visualization in Proteomics and Metabolomics. *Journal of Proteome Research*, 2016.

[9] Anton A Goloborodko, Lev I Levitsky, Mark V Ivanov, and Mikhail V Gorshkov. Pyteomics–a Python framework for exploratory data analysis and rapid software prototyping in proteomics. *Journal of the American Society for Mass Spectrometry*, 24(2):301–4, feb 2013.

[10] M Kösters, J Leufken, S Schulze, K Sugimoto, J Klein, R P Zahedi, M Hippler, S A Leidel, and C Fufezan. pymzML v2.0: introducing a highly compressed and seekable gzip format. *Bioinformatics*, jan 2018.

[11] Hannes L. Röst, Uwe Schmitt, Ruedi Aebersold, and Lars Malmström. pyOpenMS: A Python-based interface to the OpenMS mass-spectrometry algorithm library. *Proteomics*, 14(1):74–77, 2014.

[12] Matthew C. Chambers, Brendan MacLean, Robert Burke, Dario Amodei, Daniel L. Ruderman, Steffen Neumann, Laurent Gatto, Bernd Fischer, Brian Pratt, Jarrett Egertson, Katherine Hoff, Darren Kessner, Natalie Tasman, Nicholas Shulman, Barbara Frewen, Tahmina A. Baker, Mi Youn Brusniak, Christopher Paulse, David Creasy, Lisa Flashner, Kian Kani, Chris Moulding, Sean L. Seymour, Lydia M. Nuwaysir, Brent Lefebvre, Frank Kuhlmann, Joe Roark, Paape Rainer, Suckau Detlev, Tina Hemenway, Andreas Huhmer, James Langridge, Brian Connolly, Trey Chadick, Krisztina Holly, Josh Eckels, Eric W. Deutsch, Robert L. Moritz, Jonathan E. Katz, David B. Agus, Michael MacCoss, David L. Tabb, and Parag Mallick. A cross-platform toolkit for mass spectrometry and proteomics. *Nature Biotechnology*, 30(10):918–920, 2012.

[13] The lxml Development Team. lxml, 2006–. [Online; accessed 08-06-2017].

[14] Travis E. (19..-....). Oliphant. *Guide to NumPy*. Continuum Press, 2015.

[15] The SQLAlchemy Development Team. SQLAlchemy, 2005–. [Online; accessed 08-06-2017].

[16] Xiaowen Liu, Yuval Inbar, Pieter C. Dorrestein, Colin Wynne, Nathan Edwards, Puneet Souda, Julian P. Whitelegge, Vineet Bafna, and Pavel A. Pevzner. Deconvolution and database search of complex tandem mass spectra of intact proteins: a combinatorial approach. *Molecular & cellular proteomics : MCP*, 9(12):2772–2782, sep 2010.

[17] Lev I Levitsky, Mark V Ivanov, Anna A Lobas, Julia A Bubis, Irina A Tarasova, Elizaveta M Solovyeva, Marina L Pridatchenko, and Mikhail V Gorshkov. IdentiPy: an extensible search engine for protein identification in shotgun proteomics. *Journal of Proteome Research*, page acs.jproteome.7b00640, 2018.

[18] The UniProt Consortium. UniProt: a hub for protein information. *Nucleic Acids Research*, 43(D1):D204–212, oct 2014.

[19] David M Creasy and John S Cottrell. Unimod: Protein modifications for mass spectrometry. *Proteomics*, 4(6):1534–6, jun 2004.

[20] Barry Smith, Michael Ashburner, Cornelius Rosse, Jonathan Bard, William Bug, Werner Ceusters, Louis J Goldberg, Karen Eilbeck, Amelia Ireland, and J Christopher. The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. *Nature Biotechnology*, 25(11), 2007.

[21] Lukas P M Kremer, Johannes Leufken, Purevdulam Oyunchimeg, Stefan Schulze, and Christian Fufezan. Ursgal, Universal Python Module Combining Common Bottom-Up Proteomics Tools for Large-Scale Analysis. *Journal of Proteome Research*, 15(3):788–794, 2016.

List of Figures

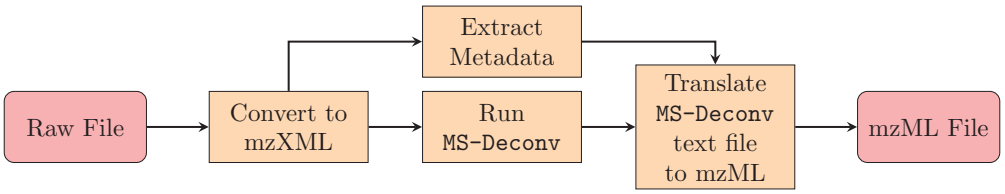

Figure 1: Process for MS-Deconv translation