7. ACE XML: File formats for expressing MIR data

7.1 Overview of MIR data mining file formats and ACE XML

Much of the cross-institutional efficiency of MIR research hinges on the ability of researchers to share data effectively with one another. Information such as ground-truth annotations, for example, can be very expensive to produce, and a great deal of repeated effort is avoided if researchers are able to share such information efficiently. Similarly, training and testing datasets themselves can be expensive to acquire, and since they cannot typically be distributed directly because of legal limitations, the ability to share representative feature values efficiently can be very valuable. The communication of more abstract information, such as class-label ontologies or characteristics of features themselves, can also be very useful.

Well-constructed, flexible and expressive standardized file formats are essential for distributing all of these types of information efficiently and fully. Furthermore, in order to encourage adoption as a standard, such file formats must be simple for both humans and machines to understand, parse and write. The absence of such standardized formats can pose an obstacle to the sharing of research information, with the result that each lab has a greater tendency to generate its own in house data, which results in both wasteful repeated effort and, in general, lower quality data.

Although there are a variety of widely used general-purpose data mining and classification file formats in existence, none of them meet the very specific needs of MIR research. The Weka ARFF format (Witten and Frank 2005), for example, is currently the de facto standard in MIR research, but as is shown in **Sections 7.3.2** and **7.4,** it has severe restrictions with respect to the requirements of realistic MIR research. Formats such as ARFF impose serious limitations on the types of information that can be represented and, accordingly, on the quality of research that can be performed based on this information. To give just one example, most such formats, including ARFF, only allow instances to be labelled with just one class label at a time, something that fundamentally limits the sophistication and realism of research in areas such as genre classification.

Sections 7.2 and **7.3** respectively review existing file format technologies and the particular formats that are currently the most prominent in MIR research. **Section 7.4**

presents a critical analysis of the limitations of existing formats, and introduces a corresponding list of design priorities that can be used to guide the development of new file formats designed specifically for use in music data mining and classification research.

The primary focus of this chapter is on the original ACE XML file formats. These formats were designed based on the design priorities emphasized in **Section 7.4**, and are intended for use in music data mining and classification research of any kind. Possible applications of these formats include genre classification, artist classification, track segmentation, pitch tracking, instrument identification and so on. The ACE XML formats may be used equivalently well with respect to audio, symbolic and cultural data.

ACE XML has recently been chosen for use in the NEMA (Networked Environment for Music Analysis) project,[1] a large-scale multinational and multidisciplinary effort to create a general music information processing infrastructure. NEMA is funded by the Scholarly Communications program of the Andrew W. Mellon Foundation, and involves research groups from McGill University, University of Illinois at Urbana-Champaign, University of Southampton, University of Waikato and Goldsmiths and Queen Mary at University of London.

ACE XML is the native format used by all jMIR components to communicate with one other.[2] The ACE project also includes a general API for parsing, writing and processing ACE XML files so that ACE XML functionality can be easily incorporated into other software as well. The ACE GUI prototype also includes functionality for manually generating, displaying, editing and saving ACE XML files.

There are four primary types of ACE XML files, which may be used individually or integrated together:

- **Feature Value:** These files express feature values extracted from specific instances.
- **Feature Description:** These files express abstract information about features. To give a few examples, this format can be used to express specific details about the processes used to extract feature values that are expressed in Feature Value files,

[1] nema.lis.uiuc.edu

[2] jMIR components can also read and write ARFF for compatibility reasons, although it is recommended that ACE XML be used instead.

to publish information about the features that can be extracted by a new feature extraction application, to express updates to existing feature extraction algorithms, and so on.

- **Instance Label:** These files express labelled annotations of specific instances. This format can be used to notate ground-truth labels or the predicted labels output by a classification system, for example.
- **Class Ontology:** These files express relationships between different classes. This can be used to simply specify candidate class labels, or for more sophisticated purposes such as expressing hierarchical taxonomical relationships between classes that can be taken advantage of by specialized machine learning techniques.

There are two versions of ACE XML, namely ACE XML 1.1 and ACE XML 2.0. Version 1.1 (McKay et al. 2005) is the stable version that is currently implemented in all of the jMIR components, including the ACE API. For the purposes of this publication, 1.1 is the official version of ACE XML. **Section 7.5** describes the four ACE XML 1.1 formats in general, and **Sections 7.6** to **7.9** focus on each of them individually. The ACE API is discussed in **Section 7.10**.

The newer ACE XML 2.0 is ultimately intended for candidacy as a standardized format for MIR research in general, without any inherent links to jMIR. It builds upon the ACE XML 1.1 formats to add even more expressivity and functionality. Prototypes for the updated versions of each of the four main ACE XML formats are introduced in **Section 7.11**. As discussed in **Section 7.11.1**, version 2.0 also includes the new ACE XML Project file and ACE XML ZIP file types, which can be used to more conveniently associate and potentially package different ACE XML files together. Additional potential future improvements are also discussed in **Section 7.13**.

The ACE XML 2.0 formats are presented here for review and improvement by the MIR community at large. The actual implementation of ACE XML 2.0 awaits amendment and finalization of the formats based on community feedback, so the ACE XML 1.1 formats, which are presently fully finalized and implemented, continue to serve as the standard formats used by the jMIR components at the time of this publication.

A review of the original research contributions of this chapter is provided in **Section 7.12**. **Section 7.14** also provides artificially generated samples of each of the ACE XML 1.1 and 2.0 formats in order to more clearly illustrate how the file formats can be used.

7.2 Background information

This section provides background information on the principal fundamental concepts and technologies that are relevant to the representation of information related to automatic music classification research. The contents of this section are presented in order to ensure that the reader is familiar with the concepts that are necessary to fully appreciate and compare the ACE XML file formats outlined later in this chapter with the existing alternative formats.

7.2.1 ASCII and Unicode text files

The term *text file* refers to a simple kind of computer file that holds basic textual data. Text files conform to simple standards for representing text, and are thus highly portable. Most text files that are referred to as such are *plain text* formats, which is to say that they do not include any provisions for formatting codes other than very simple markers such as *end of line, end of file* and tab markers. The lack of formatting in plain text files is both an advantage and a disadvantage, in that this generally results in smaller files, but also results in less expressivity.

The primary advantage of text files is that they follow a simple standard that can be parsed on essentially any computer running essentially any operating system. Applications called *text editors* are typically used to read, write and edit text files, although word processors and many other types of applications are compatible with text files as well.

ASCII (American Standard Code for Information Interchange) is perhaps the most common text file format. It is widely used enough to be considered platform independent, and ASCII files are often given a *.txt* extension, although this extension is also sometimes used for other types of text files as well. ASCII files use one byte to encode each character, with one bit reserved as a parity bit to aid in the detection of data corruption. This means that 128 characters may be represented in ASCII, 33 of which are mostly obsolete control characters. The remaining 95 characters include the lower-case and

upper-case letters, numbers and punctuation marks associated mainly with English and related European languages, as well as a few mathematical characters, accents and other miscellaneous characters.

This ASCII limit of 95 printable characters is much too small to deal with all of the characters that even an English speaker might wish to use, much less someone writing in a language that uses a different alphabet. As a result, there are many other standards, usually chosen based on the default locale setting on a user's computer. An example is the technically obsolete but still often used *ISO 8859-1* encoding used for many European languages.

Limiting the space needed to store a single character to a single byte was reasonable in the past when data storage and transmission was expensive. Such rationing is no longer as much of a priority, however, and as a result the much more expressive *Unicode* standard is replacing ASCII as the preferred standard.

The multilingual full Unicode standard includes more than 100,000 characters, and has proven to be very valuable in the internationalization of computer software. It has been adopted by the XML standard, by Java and by the Micrsoft .NET framework, among many others.

There are a variety of Unicode encodings in existence. One of the most common of these is the variable-length *UTF-8* encoding, which uses one byte for all ASCII characters and up to three additional bytes for all other characters. UTF-8 has the significant advantage of being backwards-compatible with ASCII.

UTF-16 is the most common alternative Unicode encoding. UTF-16 is also a variable-length encoding, and it uses up to four bytes. UTF-16 can sometimes be more space-efficient than UTF-8, but the reverse can also be true, depending on the particular characters that need to be encoded. Both UTF-8 and UTF-16 may be used to encode any Unicode character.

7.2.2 Escape characters and delimiter-separated values

Some applications make use of reserved combinations of characters to effectively add additional characters or formatting instructions to text files that are not directly permitted by their encodings. This is often done via the use of *escape characters,* such as a backslash, that indicate that the following character is to be interpreted in a special way.

Although files that use this approach are of course still text files, they nonetheless lose some portability and human-readability, as the software or individual parsing the files must be aware of the rules governing these special codes in order to properly interpret them.

It is often desirable to store data in text files in some structured way, such as in the case of tables of values. One simple way of doing this is to use *delimiters,* which is to say special characters that are reserved to separate values, such as a list of names. Commas, tabs and end-of-line characters are three of the most commonly used delimiters. This approach is useful as a quick and easy solution, but once again involves a loss of portability, since it requires that parsing software be aware of the particular delimiters that are being used.

There are a variety of standardized delimited text file formats. The comma-delimited CSV standard is perhaps the best-known example.

7.2.3 XML

It is often desirable to be able to express textual data in structured ways that are more sophisticated and general than is possible with simple delimited text files. *XML (eXtensible Markup Lanuage)* files, which are encoded in UTF-8 by default, provide one particularly flexible and convenient way of doing this.

XML is an example of a *markup language,* which is to say that it is an artificial language that uses annotations to impose structure and formatting on text. HTML is perhaps one of the most famous markup languages, due its role as the traditional format in constructing web pages. An essential difference between XML and HTML is that XML allows users to specify how data is to be structured, whereas HTML requires that data be formatted in rigidly pre-defined ways.

XML attempts to strike a balance between permitting data to be represented in such a way that it is conveniently structured for machine processing, and requiring that it be stored in simple text files in ways that are relatively easily human readable. The ability for humans to read XML files directly is augmented by functionality in many web browsing or text editing applications to display data stored in an XML files in an easily human-parsible way that is consistent with the structuring specified in the XML file,

while at the same time hiding the infrastructure that specifies this structuring. Specialized software such as Altova XML Spy[3] provides even greater functionality in this respect.

XML data essentially consists of two parts: one that specifies the structuring and formatting that stored data must conform to, and the other storing this data using the specified infrastructure. The XML specification permits a variety of ways of performing this first task, the oldest, least powerful and simplest of which is known as a *Document Type Definition (DTD).* A DTD typically consists of a separate file or header preceding the actual data that is stored in an XML file. Alternative XML *schemas* allow added expressivity over DTDs, but at the cost of increased complexity.

The formatting of the data stored in XML files must conform to the rules laid out in the DTD or schema. There are many software packages and web services (e.g., validator.w3.org) that can be used to verify that this is the case for any given document. This is referred to as checking whether an XML document is *well-formed.*

The majority of XML documents consist of clauses of information denoted using *elements.* Each element has a *start tag* and an *end tag* as well as, often, some *content* between the tags. The tags indicate the kind of information that the content expresses, typically in the form of a field label, and the content indicates the value for the field.

For example, the element *<authour_name>Alexander Solzhenitsyn</authour_name>* includes start and end name tags, indicating that the content of the tags is an authour's name, and the content itself specifies the name of the specific authour, *Alexander Solzhenitsyn* in this case. Start and end tags of elements are always each contained within < and > signs, and end tags always have the same name as their matching start tag, but have a / sign added at their beginning.

Elements can be organized hierarchically, which is to say that the content of an element can itself contain one or more other elements. The elements that may appear and the rules governing their structuring are declared in the DTD or schema of each XML file. **Figure 7.1** provides an example of how elements can contain other elements.

[3] www.altova.com

```
<my_books>
    <book>
        <title>Gulag Archipelago</title>
        <authour_name>
               <first_name>Alexander</first_name>
               <last_name>Solzhenitsyn</last_name>
        </authour_name>
    </book>
</my_books>
```

Figure 7.1 Sample hierarchically organized XML elements. This could be used, for example, to store a database of the books that one owns. The *my_books* element could be used to hold multiple *book* elements, each of which refer to one separate book, although only one is listed here. The *book* element contains a *title* element and an *authour_name* element. The *authour_name* element itself contains two further elements. One would expect there to be one *book* element for every book that is owned. The names of these particular kinds of elements and the relationships between them must be specified in a DTD header or other XML schema.

Elements may also contain *attributes* that provide further information. The rules governing the nature of the attributes and which elements may contain particular attributes are also specified in the DTD or other XML schema. Attributes are noted in the start tag of each element by following the name of the tag with a space, the name of the attribute, an equal sign and, finally, the value of the attribute enclosed within double quotes. **Figure 7.2** shows how attributes might be used in an expanded version of the data shown in **Figure 7.1**.

```
<my_books>
    <book staus="owned">
        <title>Gulag Archipelago</title>
        <authour_name alive="no">
               <first_name>Alexander</first_name>
               <last_name>Solzhenitsyn</last_name>
        </authour_name>
    </book>
</my_books>
```

Figure 7.2 A modified version of the *book* element from **Figure 7.1**. Two attributes are added, namely a *status* attribute for the *book* tag and an *alive* tag for the *authour_name* tag. The *status* attribute could be used to indicate whether the book is *owned*, *read but not owned* or *not read*, for example, and the *alive* attribute would be used to indicate if the authour is currently alive.

In most cases, information expressed in attributes could alternatively be expressed using subordinate elements. For example, the *alive* attribute in **Figure 7.2** could just have easily been a child element of *authour_name,* just as *last_name* is. Although which approach one uses is mostly a matter of style, a general rule of thumb is to use an attribute if only restricted values are possible (e.g., yes or no) and to use an element if arbitrary values are possible. To return to the example if **Figure 7.2**, an alternative architecture might be to make *status* a child element, but leave *alive* as an attribute, as shown in **Figure 7.3**.

```
<my_books>
    <book>
        <title>Gulag Archipelago</title>
        <authour_name alive="no">
               <first_name>Alexander</first_name>
               <last_name>Solzhenitsyn</last_name>
        </authour_name>
        <status>owned</status>
    </book>
</my_books>
```

Figure 7.3 Alternative architecture to that of **Figure 7.2**, where the *status* field is now a field instead of an attribute.

As noted above, which elements are permitted and how they may be structured is defined in a DTD header or other schema. A DTD is essentially a statement that consists

of *<!DOCTYPE fileype […]>*, where *filetype* is a code that is chosen to identify the type of XML file that the DTD is defining (e.g., *my_books* might be used to identify a file format being used to store one's book collection). The *[…]* holds a separate declaration for each element and attribute type that is permitted. Each element declaration is a statement consisting of *<!ELEMENT elementname (elementdeclarationtype)>*. The *elementname* variable specifies the text of the start tag of the element and the *elementdeclarationtype* variable specifies which child elements are permitted, if any, which attributes are permitted, if any, and what kind of content data (e.g., character data) is permitted for the element. The order that the element declarations appear in the DTD overall and in each individual element declaration type constrain the order that the tags may be used in the document if it is to be well-formed. **Figure 7.4** provides an example of a sample DTD.

```
<!DOCTYPE my_books [
    <!ELEMENT my_books (book+)>
    <!ELEMENT book (title, authour_name, status?)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT authour_name (first_name, last_name)>
    <!ATTLIST authour_name alive CDATA "yes">
    <!ELEMENT status (#PCDATA)>
    <!ELEMENT first_name (#PCDATA)>
    <!ELEMENT last_name (#PCDATA)>
]>
```

Figure 7.4 A possible DTD for the data formulation used in **Figure 7.3**. This could be stored in an external file or could be included as a header in the same file that stores the data about the book(s) themselves. Note that in this particular DTD there must be one or more books (because of the *+* sign), the status element is optional for each authour (because of the *?* sign and the default value for the *alive* attribute is specified as *yes*.

The DTD also specifies how attributes are used, as can also be seen in **Figure 7.4**. The *<!ATTLIST elementname attributename1 CDATA "default1" attributename2 CDATA default2 …>* declaration provides a list of all possible attributes for the *elementname* element, each of which is given the name *attributename#* and the default value of *default#,* which will be used if a particular entry does not specify a value for the attribute. The *CDATA* code simply means that the value for the attribute will be in the form of normal text.

Note that the *<!ATTLIST authour_name alive CDATA "yes">* declaration in **Figure 7.4** does not constrain the possible choices that may be used in a well-formed file, so an instance of the *alive* tag might be given reasonable values such as *yes, no* or *unknown* as well as unreasonable values such as *dafdasfd*. This flexibility may be desirable in some cases, but not in others. It is possible to constrain the valid values for an attribute, such as in the following example: *<!ATTLIST authour_name alive (yes | no | maybe) "yes">,* which requires that the attribute be assigned values of either *yes, no,* or *maybe,* with a default of maybe if the attribute value is omitted for a particular element.

As an alternative to specifying default values for attributes, it is also possible to use the *#IMPLIED* keyword in the attribute declaration to make the attribute optional for the element, or the *#REQUIRED* keyword to require that a value for the attribute be specified whenever the associated element appears. Alternatively, the *#FIXED* keyword followed by a value in quotation marks may be used to specify that the attribute will always have the given value. For example, an *<!ATTLIST authour_name alive CDATA #IMPLIED>* statement in a DTD would make the *alive* attribute optional, and an *<!ATTLIST authour_name alive CDATA #REQUIRED>* statement would require that it be specified whenever the *author_name* element is used.

As is apparent from the discussion above and from **Figure 7.4**, there are also a number of codes that may be used in DTD element declarations. These are explained in **Table 7.1**.

Element Declaration Code	Description
(#PCDATA)	Character data
(#PCDATA)*	Zero or more characters
(anelementname)	One instance of an element
(anelementname?)	Zero or one instances of an element
(anelementname*)	Zero or more instances of an element
(anelementname+)	One or more instances of an element
(anelementname1, anelementname2)	One instance of one element and one instance of another element
(anelementname1 | anelementname2)	One instance of one element or one instance of another element

Table 7.1 Some of the most common codes used in the element declaration codes of DTDs. This data would go in the *elementdeclarationtype* section of an *<!ELEMENT elementname (elementdeclarationtype)>* declaration. These codes can be combined so that, for example, one might have an element declaration code of *(#PCDATA, anelement1, anelement2?, anelement3+).*

There are many other options offered by XML DTDs, and still more that are made available by alternative schemas. This sub-section only describes those parts of XML that are specifically used in ACE XML, however. Whitehead, Freidman-Hill and Vander Veer (2002) provide a much more detailed description of XML, with a particular emphasis on using Java code to manipulate XML documents.

It is clear that XML allows simple, human-readable and extremely flexible document formats to be constructed. Further adding to its appeal, there is a great deal of free, open-source and well-documented code that is available for facilitating the structured parsing of XML files. Apache Xerces[4] is an example of such a parsing library that is available in a variety of programming languages. Such XML parsers typically provide two types of APIs for accessing XML data in convenient ways: *SAX* APIs allow the data to be accessed in a sequentially structured way and *DOM* APIs allow the data to be accessed in a hierarchically structured way.

When writing XML files, or manually parsing them, it is important to remember that single-byte Unicode is the default encoding, and that corresponding codes must be used, an issue which is especially important for special characters that do not have ASCII equivalents. Fortunately, many programming languages include libraries for

[4] xerces.apache.org

automatically translating plain text data appropriately. Java, for example, includes the *java.net.URLEncoder* and *java.net.URLDecoder* core classes, which can be used to ensure that all text, including special characters, is appropriately encoded and decoded.

XML also includes its own special provisions for certain special characters. XML parsers all automatically understand basic character substitutions for reserve characters that have special meanings in XML, as specified in **Table 7.2**.

XML Code	Corresponding Character
&lt;	<
&gt;	>
&amp;	&
&quot;	“
&apos;	‘

Table 7.2 Default XML character substitutions for characters that have special meanings in the XML specification.

Special characters can also be manually referred to using their decimal or x-prefixed hexadecimal Unicode code point preceded by the *&#* characters and followed by a semicolon. So, for example, the Euro symbol (€) can be referred to in an XML character field as *€* or *€*. It is typically preferable to simply pre-process all strings read from or written to an XML file by classes such as *java.net.URLEncoder* and *java.net.URLDecoder* rather than performing such manual encodings, however.

7.2.4 RDF

RDF, or *Resource Description Framework,* refers to a family of W3C[5] syntax specifications for representing relationships between different entities, or *resources*. In essence, RDF provides an abstract model for describing how anything can be related to any other thing. The result is effectively a labelled directed multi-graph.

RDF uses subject-predicate-object *triples* to make statements about the relationships between resources. The *subject* denotes a resource and the *predicate* denotes characteristics of the resource and its relationship with another resource, the *object*. For example, the statement *So What is a song that belongs to the genre of Modal Jazz* is a

[5] W3C, or the *World Wide Web Consortium,* is the primary international standards organization overseeing the World Wide Web.

triple specifying that the subject *So What* is related to the object *Modal Jazz* by the predicate *is a song that belongs to the genre.*

Resources are often referred to using *Uniform Resource Identifiers,* or *URIs,* to make them accessible via the Internet. Further information on individual resources can thus be acquired by accessing the data stored at their respective URIs, a process called *dereferencing* in RDF terminology. However, linking to resources using URIs is not an obligatory requirement of RDF. Resources can in fact potentially be abstract entities that do not actually exist anywhere on the Internet or elsewhere. In order for RDF producers and consumers to be in agreement on the semantics of resource identifiers, it is often useful to externally define certain controlled vocabularies, such as the Dublin Core[6] metadata set, which is partially mapped to a URI space for use in RDF.

A process called *reification* can be used to achieve further expressiveness using triples, or to deduce measures of confidence about triples. This involves assigning a URI to each triple so that it can itself be treated as a resource about which other triples can be formulated.

There are a variety of specific formats, called *serialization formats,* that each provide different syntaxes for representing RDF data models. XML is often used as one way to provide a structured representation of RDF models, but there are also a variety of alternative serialization formats as well, such as Notation 3.[7] There are also several query languages designed for use with RDF graphs, the most common of which is an SQL-like language called SPARQL.[8]

A mechanism for describing relationships between resources, such as that offered by RDF, is an important component of the Semantic Web. This is because such a mechanism allows software to automatically store, exchange and otherwise use machine-readable information distributed on the Internet. This direct machine usability of RDF graphs is considered to be one of the main goals and advantages of RDF.

The generality, simplicity and abstract nature of RDF are both its keys strengths and its key weaknesses. Although these characteristics allow it to be used to describe a wide range of information in flexible ways, it can also introduce computational disadvantages

[6] dublincore.org
[7] www.w3.org/DesignIssues/Notation3
[8] www.w3.org/TR/rdf-sparql-query/

and ambiguities. Similarly, the broad structural framework of RDF offers greater flexibility and extensibility than a particular given XML schema might, for example, but also does not incorporate the ability to quickly and easily define strict and sophisticated structures when in might be useful to do so, as one can in unrestricted XML.

7.2.5 OWL

OWL,[9] or the *Web Ontology Language,* is a family of languages for representing *ontologies,* which is to say formal representations of sets of concepts within some defined domain and the relationships between the concepts. Ontologies can be useful in the Semantic Web, as well as in other domains like machine learning.[10] *OWL Full,* one of the variants of OWL, provides partial compatibility with RDF.

The data contained in an OWL ontology is represented as a set of *individuals* that are related to one another via *property assertions.* OWL individuals can be collected into sets, called *classes,* whose properties are constrained by sets of *axioms.* Semantics can be inferred from explicitly defined axioms when appropriate.

7.2.6 Binary files and serialized objects

A *binary file* is a computer file consisting of 1's and 0's that may be used to encode any type of data. Although computer text files are therefore technically binary files, the term *binary file* is typically used to refer specifically to computer files that are not encoded in plain text formats.

Storing data such as extracted features in binary form can have a number of advantages over storing it in text files. For example, storing numbers even in a relatively character-sparse format such as ASCII requires at least a full byte per digit, a much greater amount of space than if the data were stored in binary. Parsing and processing data stored as text can also carry greater processing overhead.

Storing data in text files can also have important advantages over binary storage, however. One of the greatest advantages is that text files are human-readable, something that can be very convenient during debugging or other situations where direct human inspection of data is appropriate or convenient. Text files can also be parsed in an

[9] www.w3.org/TR/owl-features/
[10] Sections 9.1 and 9.2 discuss ontologies in the context of machine learning.

application and platform-independent way, while binary files are generally application-specific, and therefore much less portable. Furthermore, standard text compression techniques can be used to reduce text file sizes significantly.

Advantages such as these, combined with consistently cheaper storage, data transmission and processing power, as well as the popularity of flexible text-based data formatting protocols like XML have led to an increasing use of text as the preferred choice for data storage. For example, binary file formats in Microsoft Office have recently been replaced with compressed XML-based formats.

Many programming languages, including Java, include functionality for saving any objects in memory to files as binary *serialized objects.* This can be highly convenient for programmers, as there is no need to implement any specialized parsing or saving functionality. As might be expected, the main disadvantage of serialized objects, aside from the lack of human readability, is the common lack of portability once these objects are written to disk. Even serialized objects from relatively portable languages such as Java can sometimes fail to be properly read when accessed from newer releases of the language than they were saved under. For example, serialized Java Swing objects are often not portable across different versions of the Java Virtual Machine.

7.3 File formats used by existing MIR systems

Although most MIR research has traditionally used specialized in-house file formats for storing information such as feature values and instance labels, most of which have been either binary dumps or simple delimited text files, there has been an increasing push in recent years to use more standardized file formats so that data can be shared between different research groups. This section describes some of the best-known and most general file formats in use by the MIR community. Both this section and **Section 7.4** stress some of the strengths and weaknesses of these file formats.

7.3.1 Matlab binaries and Java serialized objects

MathWorks Matlab[11] is a numerical computing environment and programming language that is particularly popular among some MIR researchers because of the variety

[11] www.mathworks.com

of its associated stable and effective toolboxes implementing digital signal processing and machine learning functionality. Matlab can easily save information such as extracted features and learned models to binary .mat files, with the result that researchers that use Matlab tend to favour the .mat file format.

Unfortunately, .mat binaries suffer from the same limitations as all binaries, as discussed in **Section 7.2.6**. Although there are a variety of scripts written in different languages for parsing Matlab binaries, the issue remains that Matlab is commercial software that uses a proprietary file format that is subject to the design decisions of MathWorks, which may not coincide with the needs of the MIR community. Furthermore, .mat files are not directly human-readable, nor do they allow data to be as easily structured in useful ways as some of the alternative file formats.

Any serialized Java object can be saved directly to disk and parsed by the Java Virtual Machine. The ease with which this can be done has made the use of Java serialized objects attractive to some researchers, just as has been the case with Matlab .mat files, but once again, concerns about the portability, stability, structural expressivity and lack of human-readability are serious concerns.

Nonetheless, one is sometimes forced to use serialized objects when dealing with third-party software. An example of this is the saving of trained Weka models.[12]

7.3.2 Weka ARFF

Weka (Witten and Frank 2005) is a well-known set of open-source machine learning libraries implemented in Java. Weka includes several front ends, including graphical user interfaces, as well as the core libraries and their associated APIs. The breadth and ease of use of Weka have caused it to be adopted by many MIR research labs and, as a result, its ARFF file format is likely the most commonly used format in the MIR community for storing extracted feature values and providing them to machine learning algorithms. As the closest thing to a standardized file format that there is in MIR, the ARFF format will be given special attention here. More information on ARFF is available in Witten and Frank's book as well as on the Weka ARFF Sourceforge page.[13]

[12] Although information on instances themselves can be saved using the text-based Weka ARFF format, as discussed in **Section 7.3.2**.
[13] weka.wiki.sourceforge.net/ARFF

Throughout the description of the ARFF format that follows, several limitations of the format may become evident to the reader in relation to the specific domain of MIR research, something that is to be expected of any format as generally applicable as ARFF. The focus of this particular section is on a purely objective description of the ARFF specification, so these issues will not be discussed here explicitly. However, the weaknesses of the ARFF format with respect to MIR are discussed in some detail in **Section 7.4**.

ARFF files are basic text files that specify feature values and class labels associated with individual instances. All ARFF files consist of a *Header* section that outlines the available features and class names, followed by a *Data* section holding the feature values and, potentially, class names for each instance. Comments may also be included on lines starting with a percentage sign.

The first non-comment line of an ARFF file must begin with a single @*relation* declaration of the form:

```
@RELATION <relation-name>
```

The <*relation-name*> is a string providing a name for the basic relationship or type of information that the ARFF file represents. For example, in the case of an artist identification task, it might be *artist_classification* or *artist_identification*. As with other types of ARFF data, <*relation-name*> strings with spaces or percent signs in them must be enclosed in quotation marks. Also, as with other ARFF keywords, @*relation* statements are case insensitive.

The rest of the header consists of @*attribute* declarations of the form:

```
@ATTRIBUTE <attribute-name> <datatype>
```

<*attribute-name*> and <*datatype*> are strings specifying, respectively, the name of a feature,[14] in the form of a string, and the data type of this feature. If a feature value is present in one or more instances then it must be declared in an @*ATTRIBUTE* statement in the header. The following data types may be specified for a feature in @*ATTRIBUTE* statements:

- numeric: Numeric data that may be an integer or a real number.
- integer: Integer-only numeric data.

[14] Weka refers to *features* as *attributes.*

- real: Numeric data that is in the form of real numbers.
- string: A string of text data. Strings containing spaces or percent signs must be wrapped in quotation marks.
- <nominal-specification>: A string of text data that must, for the feature value of any given instance, correspond to one of a set of eligible strings specified for the feature in the *@ATTRIBUTE* declaration. Such declarations are in the form: *{<nominal_name_1>, <nominal_name_2>, <nominal_name_3>, ...}*.
- date: A date, which may be formatted in a variety of ways. The default is yyyy-MM-dd'T'HH:mm:ss.
- relational: A format implemented only in the most recent developer versions of Weka that allows a simple hierarchical relationship to be specified for features so that multi-instance classifiers can be used.

The final *@ATTRIBUTE* declaration in the header usually specifies the possible class names that an instance may have. The <*attribute-name*> is specified *class,* and the <*datatype*> must be a list of strings in the <*nominal-specification*> form.

The Data section of an ARFF file is specified once all features (and class names) have been declared in the Header section. The Data section is begun by placing a single-line *@DATA* statement after the last *@ATTRIBUTE* statement.

The feature values for each instance are then specified on a single line for each instance. The feature values must each be separated by a comma, and the feature values must be listed in the same order that the features were themselves declared in the Header with *@ATTRIBUTE* statements. Unknown feature values may be denoted by using a single question mark as a place holder. String and nominal feature values are case sensitive, even though Weka keywords are not case sensitive.

Figure 7.5 provides a complete example of an example Weka file.

```
% TITLE: Music, applause, speech, silence discriminator.
%
% This artificial data, intended for demonstration purposes,
% specifies possible feature values extracted from windows of
% audio that could be used for segmenting the audio into
% sections of music, applause, speech and silence.
%
% Since this is a simple demonstration, only the basic
% features of spectral centurion and the duration of the
% recording from which the window was extracted are specified.

@RELATION music_applause_speech_silence_discriminator

@ATTRIBUTE spectral_centroid     NUMERIC
@ATTRIBUTE duration              NUMERIC
@ATTRIBUTE class                 {music, applause, speech, silence}

@DATA
0.0,250.0,silence
440.0,250.0,music
526.0,250.0,applause
0.0,372.5,applause
220.0,372.5,music
115.0,372.5,music
115.0,372.5,applause
854.6,960.3,?
```

Figure 7.5 A complete sample Weka file. The first lines all begin with percentage signs, which indicates that they are simply comments. The *@RELATION* statement indicates the beginning of the feature declarations and specifies a name for the relationship represented by the Weka file. Three features are specified using *@ATTRIBUTE* statements, namely *spectral centroid, duration* and *class.* In practice, *class* is not really a feature, but rather a *nominal-specification* of the candidate classes that instances can have. This is how classes are always declared in Weka. The actual features are listed for eight instances after the *@DATA* declaration, with the class name specified as the last attribute for each instance. Note that the class name is not specified for the last instance, since it is replaced by a question mark. This convention can be used for feature values as well if they are unknown.

Weka also allows a slightly modified alternative to standard ARFF files called *sparse ARFF files.* These are essentially the same as standard ARFF files, except that they do not explicitly specify zero-value feature values,[15] and feature values are linked with index values associating them with particular features.

[15] Values are assumed to be zero if they are not specified.

As of Weka 3.5.8 (a developer version), weights can be associated with instances. This is done by enclosing them in curly braces and appending the weight to the end of the line, as in the following example corresponding to the first instance from **Figure 7.5**:

```
0.0,250.0,silence, {4}
```

A weight of 4 is assigned here to this particular instance.

As a final note on Weka data formats, it should be noted that ARFF files cannot be used to represent trained models. Weka instead stores these as Java serialized objects.

7.3.3 SDIF

SDIF (Wright et al. 1999; Schwartz and Wright 2000), or the *Sound Description Interchange Format,* is a standardized file format for describing sound that was jointly developed by IRCAM and CNMAT. It is popular in the signal processing community. SDIF files consist of a basic fixed framework, and also include an extensible set of description types, including time-domain descriptions, frequency-domain models and sinusoidal models. SDIF files make use of XML.

The primary purpose of the SDIF format is the storage of audio for use in signal analysis and synthesis. The storage of feature values and other MIR-oriented data mining information is not the primary emphasis of the SDIF format, although features can certainly be represented as well. For example, one of the most fundamental structures of the SDIF format is a series of *frames,* each consisting of a four-byte *Frame Type ID,* a four-byte integer *Frame Size* and the frame data itself. Each frame must be a multiple of 64 bits. It is clear that this sort of structuring is not ideally suited for most MIR-oriented tasks.

Although SDIF can be extended to store features for the purposes of MIR applications (Burred et al. 2008), the emphasis is still on efficiently representing audio data, not on representing features in ways that are as flexible and clear as would be desirable in an ideal MIR-oriented format, nor on dealing with non-audio data such as symbolic or cultural features. The SDIF format is also not as convenient as one would ideally like for representing other MIR-relevant information such as sophisticated ontological class structures.

So, while SDIF is one of the best formats available for use in the audio signal processing research community, it is not as ideally suited for MIR. Although it is true that

SDIF can be adapted for MIR research, using SDIF to meet the ideal needs of an MIR standardized file format, as described in **Section 7.4**, can be awkward.

7.3.4 Music Ontology

Music Ontology (Raimond et al. 2007; Raimond and Sandler 2008; Raimond 2009; www.musicontology.com) is a framework designed for dealing with music-related data on the Semantic Web, with the particular needs of MIR applications in mind. It is designed to be very flexible, and takes advantage of RDF to facilitate connections between resources and to make it possible to access further information about resources by dereferencing them. Music Ontology is divided into three main areas that respectively deal with *editorial information* (e.g., track names, musicians, record labels, etc.), *production workflow information* (e.g., arrangements, compositions, etc.) and *event decompositions*, (e.g., specifying that a particular musician played in a particular key at a particular time).

Music Ontology makes use of several existing ontologies, including the *Timeline, Event* and *Functional Requirements for Bibliographic Records* ontologies. The Timeline ontology, which is based on OWL, is used for representing a variety of types of temporal information, and can represent instances in time, intervals in time and references to defined timelines.

The Event ontology is used to represent particular musical or other events that can be localized in both time and space. Music Ontology Events can also have *factors* (e.g., a musical instrument), *agents* (e.g., a performer playing the instrument) and *products* (e.g., the physical sound produced by the musician playing the instrument). Complex Events can also be related to subordinate *Sub-Events,* such as a large Event consisting of an ensemble of musicians playing music made up of Sub-Events each consisting of a particular musician playing particular notes.

The Functional Requirements for Bibliographic Records ontology includes *Works* (abstract artistic creations such as musical compositions), *Manifestations* (physical embodiments of Works, such as CDs in general) and *Items* (an instance of a Manifestation, such as a particular CD). Music Ontology also makes use of other existing ontologies, such as the social networking-oriented *FOAF* (Friend Of A Friend) ontology and its concepts of *Persons* and *Groups*.

Music Ontology is designed using an object-oriented approach that emphasizes inheritance. For example, the *Key* ontology, *Instrument* taxonomy and *Genre* taxonomy are all sub-classes of Events.

Of particular interest with relation to machine learning, Music Ontology includes an *Audio Features* ontology[16] intended for the expression of features extracted from audio signals. Events are interpreted in this context to be regions of time corresponding to particular features called *FeatureEvents,* and each FeatureEvent may have a number of *Feature* factors that each represent a feature such as a musical key or a set of MFCCs.

A number of data resources have already been converted to Music Ontology, including Musicbrainz musical metadata[17] and data from the DBTune music repositories.[18] Music Ontology is also being used as the main representation format in the large OMRAS2 project, and is used by high-quality MIR-oriented software such as Sonic Visualiser (Cannam et al. 2006) and some of its associated Vamp plugins.[19]

A number of advantages and disadvantages of Music Ontology relative to ACE XML will become apparent to readers as the details of ACE XML are presented later in this chapter. Overall, Music Ontology has both the relative strength and weakness that it is very general, and is intended for MIR-oriented applications that are much wider in scope than the music classification focus of ACE XML. This enables Music Ontology to represent a much greater range of information more conveniently than can be done in ACE XML, but does not have all of the structural advantages of the ACE XML formats that make them particularly useful for music classification. XML in general tends to be much better suited to representing well-structured data than RDF. Although the Audio Features ontology subclass of Music Ontology does bring a greater focus on the music classification domain, this ontology as it is described in motools.sourceforge.net/doc/audio_features.html is not as flexible and convenient as ACE XML with respect to feature values, instance labels, feature descriptions and class interrelationships. Also, the Audio Features ontology is designed particularly with respect to audio features, whereas ACE XML treats audio, symbolic and cultural features equally

[16] motools.sourceforge.net/doc/audio_features.html
[17] zitgist.com
[18] purl.org/dbtune/
[19] www.vamp-plugins.org

and equivalently. Finally, the Audio Features ontology is still in relatively preliminary design stages as of the time of this writing.

Music Ontology has the advantage over ACE XML that it is explicitly designed to facilitate the referencing of external resources. Although it is certainly possible to specify URIs in ACE XML identifier fields, ACE XML does not have an explicit RDF framework that makes it particularly convenient to do so. Furthermore, the RDF framework makes it possible to use existing tools such as SPARQL to query Music Ontology data.

The choice to use basic XML rather than RDF in ACE XML does carry a number of advantages, however. For example, ACE XML files are typically much more human readable than Music Ontology's RDF files, which are more oriented towards machine readability. ACE XML files are also simpler and more obviously structured, with fewer varieties. This makes them very easy for users to learn and write code for. The majority of researchers in the MIR community and many of its associated disciplines tend to, in general, be much more familiar with XML than they are with RDF, with the consequence that they will likely be more willing to adopt an XML-based standard.

Of course, file parsing and writing code libraries have already been implemented for both ACE XML and Music Ontology, but the simple and clear structuring of ACE XML makes it easier for MIR researchers to quickly inspect the simple and self-contained ACE XML DTDs, learn them and write code for them, something that is essential for file formats intended for adoption as standards. This is particularly true for key areas of MIR research that are more oriented to the humanities than to technical applications. A related advantage is that ACE XML only relies on simple XML parsing, and does not require the installation of packages for parsing additional standards such as OWL, for example.

Also, although the implicit ease of linking disparate resources offered by RDF can certainly be a strong advantage in many contexts, it can also be a disadvantage when different linked documents are inconsistent or missing, which can often be an issue in MIR, at least in its current state. RDF-based approaches by their nature tend to rely on the accessibility of potentially widely distributed network resources. This can be a significant disadvantage if one does not have network access at a particular moment, or if a remote resource is removed, renamed or moved. If even one resource is eliminated it is possible

that one will not only lose access to it, but potentially to all of the resources that it refers to as well. Self-contained XML files, on the other hand, do not carry this risk.

Furthermore, the ability to easily store information such as feature values and instance labels locally if desired can be an important advantage when dealing with many gigabytes of feature values. Limitations such as slow network connections and monthly bandwidth caps can pose serious obstacles when dealing with widely distributed network ontologies, but are less of a problem with file types that store the most essential information in a self-contained way and that can be downloaded or uploaded when convenient, such as ACE XML. Although RDF-based ontologies do certainly have additional important advantages of their own, they are perhaps better suited to relatively small amounts of textual data than to the very large feature associated with many typical MIR use cases and the even larger datasets that are likely to arise in the future as MIR research scales up to include larger quantities of music. ACE XML 2.0 files can also, of course, be made accessible on networks and linked to external ontologies if desired, but the ability to easily use them purely locally can be an important advantage.

ACE XML seeks to reach a compromise between the strong encouragement of distributed linked files promoted by RDF and the more conventional approach of having all data, including feature values and class labels, contained in a single file. ACE XML does this by using four different file formats to express different types of information, and makes it relatively easy to merge files of both the same and different types. The ability to merge separate files when convenient into single self-contained units when needed for the purposes of convenience and robustness is emphasized in the ACE 2.0 ZIP format described in **Section 7.11.1**.

ACE XML provides a good compromise between the simplicity of ARFF and the generality of Music Ontology and RDF as a whole. ACE XML is strongly and consistently structured, with a special eye to flexibility, so that features and labels of any kind can be specified in well-understood but extensible ways. Much more useful information can be represented with ACE XML than is possible with ARFF, but a much stronger structure is imposed on the data than in RDF, with the result that all of the data that is typically used by music classification researchers or is likely to be used by them in the foreseeable future can be represented in ways that can be clearly and consistently

parsed using simple and standardized software. Furthermore, this diverse information can all be expressed in a fully self-contained way, without dependencies on distributed resources that are potentially fragile in terms of both their accessibility and longevity.

With respect to ACE XML 2.0, users have the option of using only the basics of the ACE XML specification if they wish. This means that the files are very simple and easy to learn for new users and for those implementing new software that uses ACE XML. ACE XML 2.0 also includes the ability to represent more sophisticated information if needed, and special attention has been paid to providing handles that can be used to link ACE XML 2.0 files to external resources in a variety of ways if this is a particular need, including RDF resources, thereby providing accessibility to the benefits of the RDF world.

In general, it can be said that Music Ontology is likely a preferable format for general representation and for the linking of musical data on-line, and that ACE XML is likely a preferable format in the specific domain of music classification, particularly with respect to feature extraction and instance labeling. Music Ontology can certainly be used for such purposes as well, however. In the long-term, it is certainly possible that RDF-based approaches and the semantic web in general will be able to truly demonstrate and take advantage of the power of their generality. This potential has yet to be reached, however, despite efforts dating back well over a decade. In the meantime, more strongly structured approaches such as ACE XML have significant practical advantages for use cases associated with most MIR classification research, both academic and commercial.

7.3.5 RapidMiner

RapidMiner (Mierswa et al. 2006), which was formerly known as YALE *(Yet Another Machine Learning Environment),* is an environment for performing machine learning and data mining experiments. Such experiments can be defined using nestable operators, which can be described in XML files.

Although the RapidMiner XML files can be very useful in embedding RapidMiner functionality in other applications, or in communicating experimental setups to other research groups, the particular emphasis is on communicating experimental configurations rather than on communicating data itself. This is limiting for MIR researchers who might wish to use their own algorithms developed under frameworks that

are not related to RapidMiner at all. Furthermore, RapidMiner is a general machine learning environment that is not intended specifically for music, and as such has many of the same weaknesses as Weka ARFF files when applied specifically to MIR-oriented data, as described in **Section 7.4**.

7.3.6 M2K and MIREX

M2K (Downie et al. 2005), or *Music-to-Knowledge,* is a graphical feature extraction and classification framework based on the D2K parallel data mining and machine learning system. M2K has most famously been used as the primary framework for carrying out the various MIREX[20] comparisons of different algorithms. Unfortunately, at least from the perspective of developing a standardized MIR file format, the individual committees of participants organizing each of the MIREX tasks have generally tended to choose a range of differing file formats for each task, rather than coordinating to all use the same file formats for communicating information such as feature values, instance labels and class ontologies. Most of the file formats that have been used have been either simple Java serialized objects or delimited text files, and the more sophisticated data structuring made possible by frameworks such as XML or RDF has not been taken advantage of.

7.3.7 Marsyas

Marsyas (Tzanetakis and Cook 2000) is a set of software tools for analyzing and processing audio. It was one of the first major open-source MIR systems, and has been widely used for feature extraction, among other tasks. Although Marsyas does have a few basic text file formats, such as the .mtl *Marsyas Timeline* files, the main emphasis in Marsyas is on interoperability (Tzanetakis et al. 2008), with the result that Marsyas promotes the use of existing formats like Weka ARFF files and Matlab files.

7.3.8 CLAM

CLAM (Amatrain, Arumi and Ramirez 2002) is another prominent set of signal processing software tools which, among other things, can be used to extract audio features. Although oriented more towards signal modification than specifically MIR,

[20] www.music-ir.org/mirex/2009/

features can be saved to either basic XML files or SDIF files. Although certainly useful for the purposes for which CLAM is intended, these files are not sufficiently sophisticated, expressive or flexible for the ideal needs of MIR, as specified below.

7.3.9 CrestMuseXML

CrestMuseXML (Kitahara 2008; www.crestmuse.jp/index-e.html) is an extensible framework for describing music. CrestMuse XML is part of the CrestMuse project, which emphasizes the integration of different XML formats. Although CrestMuse XML remains to be widely experimented with by the international MIR community, it does hold significant potential for integrating the benefits of different formats.

7.4 Limitations of existing formats and resultant design priorities

There are a number of important shortcomings that are each found in all or most of the existing file formats that are currently used in MIR classification research. This section discusses some of the most significant of these limitations. Taking these problems into account, a number of design priorities are then proposed for consideration in the implementation of any future file formats intended for MIR-oriented data mining applications.

Since there is insufficient space here to provide a detailed analysis of all file formats that have been used in MIR, a special focus will be placed on describing the most important shortcomings of the Weka ARFF format (see **Section 7.3.2**) in particular, as it appears to be the most commonly used format in MIR and also illustrates many of the common shortcomings of other formats. Note that this is not in any way meant to denigrate ARFF files, which are in fact so popular precisely because they are one of the best general formats available. ARFF files are intended for general data mining research, and as such certainly cannot be expected to meet the special application-specific needs of MIR.

One serious limitation of ARFF files is that there is no convenient way to assign more than one class to a given instance.[21] This is not a serious shortcoming for most pattern

[21] There are, however, two possible workarounds. The is to break one multi-class problem into many binary classification problems, so that there is a separate ARFF file for every class, with all instances classified as either belonging or not belonging to each class. Alternatively, one could create a separate class for every

recognition tasks, which typically require classification into one and only one class. However, there are many MIR research domains where the limitation of one class label per instance is a very problematic limitation. For example, any genre classifier dealing with even a moderate genre ontology would be unrealistic if it could not assign multiple labels to individual pieces of music. Similarly, a performer classification system should be able to assign multiple labels to pieces, particularly in the case of pieces with prominent soloists or with musical groups whose members have also had prominent solo careers (e.g., Cream songs should likely also be labeled with Eric Clapton, many of the pieces on the *Kind of Blue* albums should be labeled with both Miles Davis and John Coltrane, at the very least, etc.). Many MIR areas involve certain inherent ambiguities relating to class labels, and the imposition of only one class membership at the fundamental file format level is an unacceptable limitation for any realistic MIR classification system.

A second problem with ARFF files is that they do not permit any logical grouping of features. ARFF files treat each feature as an independent entity with no relation to any other feature. In contrast to this, one often encounters multi-dimensional features in music, and it can be useful to maintain logical relationships between the components of such features. Power spectra, MFCCs, bins of a beat histogram and a binary list of instruments present are just a few examples of music related multi-dimensional features. Maintaining a logical relationship between the values of multi-dimensional features allows one to perform classifications in particularly fruitful ways that take advantage of their interrelatedness, particularly with respect to classifier ensembles. Training one neural net on MFCCs, for example, and using another classifier for one-dimensional features such as RMS or spectral centroid can prove much more fruitful than mixing the MFCCs in with the other features. To give another example, it can also be useful for other reasons to group features that are derived from one another, such as in the case of the average value, average derivative and standard deviation of a particular feature.

A third problem is that ARFF files do not allow any labeling or structuring of instances. Each instance is stored only as a collection of feature values and a class label,

---

possible combination of classes, with a resulting exponential increase in the total numbers of classes. Unfortunately, both of these workarounds are inconvenient, and implicitly require classifier configurations that are much less than ideal.

with no identifying metadata. In music, it is often appropriate to extract features over a potentially overlapping time series of windows for each musical piece, something that results in sets of related ordered sub-sections of individual pieces. This is absolutely essential in applications such as automatic recording segmentation or structural analysis, and is convenient in a wide variety of MIR-oriented tasks. Furthermore, some features may be extracted for each window, some only for some windows and some only for each recording as a whole. ARFF files provide no way of associating features extracted from a window with the recording that the window comes from, nor do they provide any means of identifying recordings or of storing time stamps associated with each window. This means that this information must be stored, organized and processed by some external software using some additional unspecified and non-standardized file format.

A fourth problem is that there is no way to provide any metadata about features in ARFF files, other than unstructured information in comments. This can be a problem if data is to be shared amongst different groups, who may wish to use it to train their own systems, for example. In cases such as this, it is necessary to know details about the features and the particular parameters with which they were extracted (e.g., the roll-off point for the Spectral Roll-Off feature) if features are to be extracted from new instances to be classified based on the features provided in the original dataset.

A fifth problem is that there is no way of imposing a structure on class labels in ARFF files. One often encounters hierarchical or other ontological structures in music, such as in the cases of genre categories or structural analyses. Weka treats each class as distinct and independent. This means that there is no native way to use classification techniques that make use of structured ontologies, such as hierarchical tree classifiers, for example. This also means that there is no way to use weighted misclassification training strategies that penalize misclassifications into dissimilar classes more severely than misclassifications into similar classes.

The following list, based on the above analysis of ARFF files and on additional general observations, outlines a set of (sometimes by necessity contradictory) requirements that are proposed for consideration in the design of any new standardized file formats intended for general MIR classification research:

- File formats should be as simple and easy to understand as possible. This makes it easier for users to learn the formats and adopt them. It also decreases the probability of unforeseen conflicts and inconsistencies.
- File formats should be as flexible and expressive as possible, within the constraint of avoiding excessive complexity and redundancy.
- The data stored in the files should be easily human readable. This is important for purposes of debugging and general utility. It is can also be useful for the purposes of allowing humans to write files manually when the design of annotation software would be inappropriate or unnecessarily time consuming.
- The data stored in the files should be easily machine readable. If a file format is difficult to write parsers with or is parsed into inconvenient data structures then it will be difficult to convince users to adopt it as a standard.
- The data should be stored as efficiently as possible, in order to avoid excessively large files, within the constraint of maintaining human readability.
- Some widely accepted and well-known existing standard technology, such as XML, should be used. This increases the likelihood that new file formats will be themselves adopted as standards because they will be based on a proven technology, because users are likely to be already be at least somewhat familiar with the technology and because parsing libraries will already be available.
- File formats should rely on as few external technologies as possible. Each external technology that is present increases the probability that a given programming language used to develop a particular application may not include parsing libraries for that technology, that a parsing library does not function under a given operating system or that a component of the system will become obsolete in the future.
- The fundamental types of information that need to be represented are: feature values extracted from instances, class label annotations of instances, abstract descriptions of features and their parameters, and ontological structuring of candidate class labels.

- It should be possible to express features extracted from audio, symbolic and cultural sources of information, and treat these features equivalently so that they can easily be combined.
- It should be easy to reuse files, such as in the case of the same set of audio feature values being used for both genre classification and artist identification. Similarly, it could be convenient to reuse the same model classifications with different sets of features. For example, one could classify a given corpus of audio recordings and then later perform the same task on symbolic versions of the same corpus using the same model classifications.
- It is useful to emphasize a clear separation between the feature extraction and classification tasks. This is in contrast to formats such as Weka ARFF, which combine feature values with class labels. A separation between these two types of data is important because individual researchers may have reasons for using particular feature extractors or particular classification systems. The file format should therefore make it possible to use any feature extractor to communicate features of any type to any classification system. This portability makes it possible to process features generated by different feature extractors with the same classification system, or to use a given set of extracted features with multiple classification systems.
- It should be a simple matter to combine files of the same type, such as in the case of features extracted during different feature extraction sessions.
- It should be a simple matter to package files expressing related types of information (e.g., feature values extracted from particular instances and abstract information about the features themselves) together when appropriate, but also to separate them out when convenient. This helps to ensure data availability, integrity and accessibility, as well as flexibility.
- It should be possible to use files to reference external sources of information, but in such a way that doing so does not introduce dependencies on external

information that may no longer be available in the future or that changes unexpectedly.

- It should be possible to assign an arbitrary number of class labels to each instance. It should also be possible to express relative weightings for each of these labels.
- It should be possible to group the dimensions of multi-dimensional features and to logically associate related features and their values in general with one another.
- It should be possible to assign identifying metadata to instances (e.g., recording titles or time stamps) that will associate meaning and context for the instances so that they can be identified, both internally and externally. In general, users should be free to specify whatever metadata fields they wish, and the file format should not limit them to specific fields. However, it may be useful to supply sample templates of particular schemas.
- It should be possible to specify relationships between specific instances, in both ordered and hierarchical ways. In the case of the former, this could be a time series of analysis windows, for example. In the case of the latter, it could be a hierarchical ranking of, from bottom to top, features extracted from individual analysis windows of a recording, compared to features extracted for recordings as a whole, compared to features extracted for a performer as a whole, etc. In any case, it should be possible to express both feature values and class labels for both overall instances and ordered sub-sections of them.
- For time-series data, it should be possible for analysis windows to overlap with one another and for section labels to overlap with one another.
- For time-series data, it should be possible for analysis windows to have variable sizes, rather than requiring them all to be the same size.
- It should be possible for feature values to be present for some instances and/or sub-instances, but not others. For example, some features may be extracted for all analysis windows, some only for some windows and some only for each recording as a whole.

- Related to this, it should be possible to abstractly specify the appropriateness of different features for different contexts. For example, some features might only be appropriate to extract for a whole recording, not its analysis windows, or some features might only be possible to extract once one or more windows have already been calculated (e.g., spectral flux).
- It should be possible to specify general metadata about features, including identifying feature names, descriptions and extraction parameters.
- It should be possible to specify relationships between different class labels, both hierarchical and otherwise. This is important for the specification of class ontologies that can be taken advantage of by machine learning strategies such as hierarchical learning algorithms or weighted misclassification penalization during training. It should also be possible to relatively weight the associations between class labels.

7.5 Overview of the ACE XML file formats

This section provides an overview of the ACE XML 1.1 file formats and provides motivations for some of the general design decisions behind them. In all cases, the guidelines described in **Section 7.4** guided the design of these file formats. The proposed ACE XML 2.0 formats, as described in **Section 7.11**, go further still in meeting the **Section 7.4** guidelines.

Sections 7.6 to **7.9** provide more detailed individual descriptions of each of the four ACE XML 1.1 formats, including their DTDs. There is also a sample full ACE XML 1.1 file for each of the four ACE XML file types provided in the **Section 7.14** appendix. For the sake of comparison, these sample files are constructed such that they express the same base data as that described by the sample Weka ARFF file in **Figure 7.5**, but take advantage of the increased expressivity offered by the ACE XML file formats.

ACE XML 1.1 is the current stable version of ACE XML, and is supported by all of the jMIR software components. It is the first published version of ACE XML (McKay et al. 2005), and is a minor update over the never published ACE 1.0 test prototype. **Section 7.11** describes ACE 2.0, which is a proposed update of ACE that is not yet finalized or implemented in software.

As implied by their name, ACE XML files are all XML-based. XML was chosen because it is not only a standardized format for which parsers are widely available, but is also an extremely flexible format. It is a verbose format, with the consequence that it is less space efficient than formats such as ARFF, but this verbosity has the corresponding advantage that it allows humans to read and write the files relatively easily.

It was decided to use XML DTDs rather than another XML schema to specify the structures used by ACE XML files. Although other schemas can in general be more expressive than DTDs, DTDs are nonetheless sufficiently expressive for the purposes of ACE XML. They also have the significant advantages of being simpler and easier to understand, thereby making the ACE XML formats more attractive to new users and making the work of those implementing custom ACE XML parsers and writers easier. This is a particular issue given the wide variety of alternative schemas that are available. The average member of the MIR community is much less likely to be familiar with any particular one of these schema languages, particularly in the cases of those specialized schemas that provide enough increased expressivity to arguably have advantages over the simple DTD approach. Furthermore, DTDs tend to be much simpler and straight-forward, and are therefore much easier to learn for those users who might not know any XML at all. The ACE XML DTDs are specified in each ACE XML file along with the file's data, which means that each ACE XML file is packaged with an explanation of its formatting.

As briefly discussed in **Section 7.1**, There are four different types of ACE XML files: *Feature Value* files that express feature values extracted from instances, *Feature Description* files that describe features abstractly, *Instance Label* files that allow labels to be associated with instances and allow the specification of metadata about instances, and *Class Ontology* files that specify relationships between different candidate class labels.

Each of these four XML file types may be used independently, or they may be associated with one another and logically merged in software using unique identifying keys, such as matching *data_set_id* fields found in both Feature Value and Instance Label files, for example. Multiple files of the same type can also be merged using the ACE software. To give just a few examples: two Feature Value files containing the same features for different instances could be merged into one file, two different Feature Value files containing different features for the same instances could be merged, an Instance

Label file containing only sub-section labels and an Instance Label file containing only overall instance labels could be merged, etc.

It is not in any way necessary to provide all four ACE XML file types for any application if this is not appropriate or needed. For example, if a classifier is already trained and is to be used to classify unknown patterns, then there is certainly no need for an input Instance Label file, although one may be output during processing to express the predicted classes. The ACE software and its code libraries will automatically construct implied data for missing file types in a way that is effortless and transparent to the user. For example, if only a Feature Value file and an Instance Label file are specified, the software will automatically construct a flat class ontology based on the labels present in the Instance Label file and will also automatically generate feature descriptions based on the characteristics of the features present in the Feature Value file (e.g., the dimensionality of each of the features).

The decision to use four different file types rather than the more typical single file type is unorthodox, and therefore requires some justification. As discussed in **Section 7.4**, it is useful to incorporate a separation between feature values and instance labels. This is important for data reusability, such as in a case where one might extract features once from a large number of recordings, and then reuse this single resulting Feature Value file for multiple purposes, such as classification of artist, composer, genre and geographical point of origin. If there were only one ACE XML file type, then features would have to be repeated for each of these applications, but with the multiple file type approach the Feature Value file can remain unchanged, and be reused with a different Instance Label file for each classification task. Similarly, one can imagine a case where the same model classifications contained in one Instance Label file are used for separate sets of features extracted from symbolic, cultural and audio data respectively contained in three different Feature Value files.

Feature descriptions and class ontologies are each distributed in separate files as well in order to emphasize their independence from particular instances. For example, a Feature Description file could be published on its own to demonstrate general features that can be extracted by a particular feature extraction application in general, or a Feature Description file could be published on its own that contains specific extraction parameters

that were found to be effective for a particular research domain, or a Feature Description file could be packaged with a Feature Value file containing feature values extracted for particular instances, as appropriate. Similarly, class ontologies can be published in a way that is independent of particular instances and features, or even of a particular data set. Such file type separations emphasize the abstract nature of many types of data that are useful in music classification, and allow them to be distributed and used either independently or together, as appropriate, rather than artificially forcing links where they may not always be appropriate.

The use of separate file formats also has advantages with respect to data longevity and convenience when updating the data. For example, if new features become available after a Feature Value file has been generated, it would only be necessary to update the Feature Value and Feature Description files since the data stored in the other two file types could be reused unmodified. Similarly, if a genre ontology changed over time, it would not be necessary to update already existing Feature Value or Feature Description files.

Overall and most importantly, the separation of different types of data into four different file types makes it possible to distribute and use one type of file for arbitrary purposes without needing to impose one's own choices with respect to the types of data described by the other three file types. Also, the separation into multiple files types makes it easier to conceptualize and represent sophisticated arrangements of information with a divide and conquer approach.

jMIR includes several software tools for facilitating the use of the ACE XML file formats in general, including use outside of the scope of the principal jMIR software components themselves. Although ACE XML files may certainly be manually read, created and edited with text editors or XML editors such as XML Spy, the best way to perform such manual operations is to use the prototype ACE GUI, which displays data from single or multiple ACE XML files in particularly convenient ways.

A separate Java application called jMIRUtilities is also included as part of jMIR for, among other things, performing a number of functions that facilitate the general use of ACE XML files. Aspects of jMIRUtilities include a simple GUI for batch annotating files into an Instance Label file, functionality for generating Instance Label files based on simple tab delimited text files, functionality for accessing data from iTunes XML files so

that it can then be imported into ACE XML files, and functionality for merging features contained in separate ACE XML Feature Value files.

7.6 The ACE XML 1.1 Feature Value file format

ACE XML Feature Value files are used to express feature values that have been extracted from instances and sub-sections of instances. **Figure 7.6** specifies the DTD for Feature Value files. A sample complete Feature Value file is shown in **Code Sample 7.1** in the **Section 7.14** appendix.

```
<!ELEMENT feature_vector_file (comments,
                               data_set+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT data_set (data_set_id,
                    section*,
                    feature*)>
<!ELEMENT data_set_id (#PCDATA)>
<!ELEMENT section (feature+)>
<!ATTLIST section start CDATA ""
                  stop CDATA "">
<!ELEMENT feature (name, v+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT v (#PCDATA)>
```

Figure 7.6 The XML DTD for ACE XML 1.1 Feature Value files. This DTD precisely defines the information that may appear in Feature Value files and how it must be formatted. See **Section 7.2.3** for an explanation of XML DTDs. A sample file based on this DTD is shown in **Code Sample 7.1**.

It can be seen from the Feature Value DTD that feature values can be expressed for overall instances, called *data_sets,* that are each named using the *data_set_id* element. Each *data_set_id* refers to a unique identifier, such as a file path or a URI. Each *data_set* may or may not have sub-sections, which are each specified using the *section* element. For example, a *data_set* might correspond to an audio recording and its sub-sections might correspond to analysis windows of the recording, although there is nothing about the Feature Value specification that requires this particular arrangement.

Each *data_set* sub-section must have *start* and *stop* stamps in order to indicate what portion of the *data_set* that it corresponds to. These stamps may or may not overlap and they may or may not be of equal sizes. This makes it possible to have, for example, overlapping analysis windows of arbitrary and potentially varying sizes. There is nothing

about the *start* and *stop* attributes that requires them to denote time, however, and they could just as easily be used to denote a range of pixels in an image of album art, for example.

Features may be expressed for either a *data_set* as a whole or for individual sub-sections. In either case, the *feature* element is used to denote a new feature value or feature vector, and the particular feature is named using a *name* element in *each* feature clause.

Each *data_set* or *section* may have an arbitrary and potentially differing number of features in an arbitrary and potentially differing order. This makes it possible to omit features from some data sets or sub-sections if appropriate or if they are unavailable. Each feature may also have an arbitrary and potentially varying number of values, each denoted with a *v* element, in order to allow multi-dimensional features that may vary in dimensionality based on context.[22]

More information on instances described in a Feature Value files, such as class labels or identifying metadata, may be accessed by linking instances in the Feature Value file to instances in an ACE XML Instance Label file (see **Section 7.8**) by using *data_set_id* values that match across the two files.

Similarly, more information on the features themselves (as opposed to their values) can be specified by linking a Feature Value file to an ACE XML Feature Description file (**see Section 7.7**) by using *name* values in *feature* clauses in the Feature Value file that correspond to *name* values in *feature* clauses in the Feature Description file. However, it is not necessary to include Feature Description files with Feature Value files, since software such as ACE can automatically implicitly deduce information such as the dimensionality of features or whether particular features are to be extracted for overall instances or their sub-sections.

As a final note, it should be mentioned that a *feature_vector_file* tag is used in the DTD specification rather than a *feature_value_file* tag. This is for the purpose of legacy compatibility, since Feature Value files were called Feature Vector files in the deprecated ACE XL 1.0 specification.

[22] The dimensionality of a given feature type may alternatively be fixed in a Feature Description file, if desired.

7.7 The ACE XML 1.1 Feature Description file format

ACE XML Feature Description files are used to express abstract information about features themselves. Feature Description files do *not* specify feature values or other information related to specific instances, as this information is instead specified in Feature Value files (see **Section 7.6**). **Figure 7.7** specifies the DTD for Feature Description files. A sample complete Feature Description file is shown in **Code Sample 7.2** in the **Section 7.14** appendix.

Feature Description files make it possible to specify information about features in a general sense in a way that is independent from particular feature extractions. This makes it possible to publish a self-contained Feature Description file describing the features that a particular feature extraction application can extract, for example, or to publish a list of features and associated parameters that were found to be useful for different research application, such as instrument classification and pitch classification.

```
<!ELEMENT feature_key_file (comments,
                            feature+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT feature (name,
                   description?,
                   is_sequential,
                   parallel_dimensions)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT is_sequential (#PCDATA)>
<!ELEMENT parallel_dimensions (#PCDATA)>
```

Figure 7.7 The XML DTD for ACE XML 1.1 Feature Description files. This DTD precisely defines the information that may appear in Feature Description files and how it must be formatted. See **Section 7.2.3** for an explanation of XML DTDs. A sample file based on this DTD is shown in **Code Sample 7.2**.

It can be seen from the Feature Description DTD that information on each feature is expressed in a separate *feature* clause. Each such clause includes a *name* element uniquely identifying the feature and an optional *description* element that can be used to include textual metadata about the feature.

The *is_sequential* element for each feature specifies whether or not the feature can be extracted from sub-sections of an instance. A value of *true* means that it can, and a value

of *false* means that the feature can only be extracted from instances as a whole, not from their sub-sections. So, for example, a feature such as Most Common Pitch might have an *is_sequential* value of true for a MIDI file that is broken into analysis windows and the most common pitch is calculated for each individual window, but a feature such as Overall Key might have an *is_sequential* value of false because it would only be extracted for the MIDI file as a whole.

The *parallel_dimensions* element specifies the vector size of extracted features. This value will be 1 unless the feature is a multi-dimensional feature. So, for example, a multi-dimensional Pitch Histogram feature with 128 pitch bins would have a *parallel_dimensions* value of 128, but the one-dimensional Most Common Pitch feature would only have a *parallel_dimensions* value of 1.

Feature Description files can be linked with Feature Value files if it is desirable to describe the features used in a particular feature extraction run on a particular dataset. It can often be helpful to do this, as there are often many variable implementation details about features that are not apparent from examinations of actual feature values, so the distribution of feature details with extracted feature values makes it much easier to extract new feature values from new instances in ways that are compatible with earlier feature extractions.

The linking of a Feature Description file and a Feature Value file can be achieved by using *name* elements in *feature* clauses in the Feature Description file that correspond to matching *name* elements in *feature* clauses in the Feature Value file.

As a final note, it should be mentioned that a *feature_key_file* tag is used in the DTD specification rather than a *feature_description_file* tag. This is for the purpose of legacy compatibility, since Feature Description files were called Feature Key files in the deprecated ACE XL 1.0 specification.

7.8 The ACE XML 1.1 Instance Label file format

ACE XML Instance Label files are used to specify class labels and miscellaneous metadata about instances and sub-sections of instances. A typical use of this file type would be to express ground-truth model classifications or predicted classifications, but there are certainly other possible uses as well. **Figure 7.8** specifies the DTD for Instance

Label files. A sample complete Instance Label file is shown in **Code Sample 7.3** in the **Section 7.14** appendix.

```
<!ELEMENT classifications_file (comments,
                                data_set+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT data_set (data_set_id,
                    misc_info*,
                    role?,
                    classification)>
<!ELEMENT data_set_id (#PCDATA)>
<!ELEMENT misc_info (#PCDATA)>
<!ATTLIST misc_info info_type CDATA "">
<!ELEMENT role (#PCDATA)>
<!ELEMENT classification (section*,
                          class*)>
<!ELEMENT section (start,
                   stop,
                   class+)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT start (#PCDATA)>
<!ELEMENT stop (#PCDATA)>
```

Figure 7.8 The XML DTD for ACE XML 1.1 Instance Label files. This DTD precisely defines the information that may appear in Instance Label files and how it must be formatted. See **Section 7.2.3** for an explanation of XML DTDs. A sample file based on this DTD is shown in **Code Sample 7.3**.

It can be seen from the Instance Label DTD that class labels can be expressed for overall instances, called *data_sets,* that are each named using the *data_set_id* element. Each data_set_id should refer to a unique identifier, such as a file path or a URI. Each *data_set* may or may not have sub-sections, which are each specified using the *section* element. For example, a *data_set* might correspond to an audio recording and its sub-sections might correspond to analysis windows of this recording, although there is nothing about the Instance Label specification that requires this particular arrangement.

Each *data_set* instance may have pieces of metadata associated with it via the *misc_info* element. Each *misc_info* clause may be associated with an *info_type* attribute that specifies the type of metadata (e.g., title, album, composer, etc. field names) that the *misc_info* clause specifies.

The optional *role* element can be used to specify the purpose for which an instance is to be used. Values such as *training, testing* and *predicted* are typically used for this field

to specify the role of the instance with respect to machine learning, but any string may be provided here if desired. This can be useful for purposes such as specifying pre-determined cross-validation folds, for example, and can be particularly useful for bookkeeping when multiple Instance Label files are merged.

Class labels are assigned to overall instances and/or their sub-sections using the *class* element. Multiple labels may be assigned to each instance or sub-section, or the label may be left unspecified if a particular label is unknown.

Each sub-section of an instance must have *start* and *stop* stamps in order to indicate the portion of the instance that it corresponds to. These stamps might often be used to specify time intervals, although there is nothing requiring that they be related specifically to time. The resultant sub-section intervals may or may not overlap and they may or may not be of equal sizes.

This arrangement permits two partially overlapping regions, where each region is labelled with a different class name, and the overlapping portion is associated with both labels. Such an occasion might occur, for example, in the ground-truth for a music/applause discriminator where the applause in a live performance begins before the music ends. Such a situation could be expressed as either two sections with one label each overlapping in time or as three non-overlapping consecutive sections where the outer sections have one label each and the central section has two labels, whichever is more convenient.

Information on specific feature values extracted from instances referred to in an Instance Label file may be accessed by linking the Instance Label file to an ACE XML Feature Value file (see **Section 7.6**) by using *data_set_id* values that match across the two files. Similarly, more information on the class labels used to label instances in an Instance Label file can be accessed by linking the Instance Label file to a Class Ontology file (see **Section 7.9**) by using *class* values that match across the two files.

As a final note, it should be mentioned that a *classifications_file* tag is used in the DTD specification rather than a *instance_label_file* tag. This is for the purpose of legacy compatibility, since Instance Label files were called Classification files in the deprecated ACE XL 1.0 specification.

7.9 The ACE XML 1.1 Class Ontology file format

ACE XML Class Ontology files are used to list candidate class labels for a particular classification domain and to specify hierarchical structures that connect different classes. As discussed in **Section 7.11**, the ACE XML 2.0 version of the Class Ontology format extends the purview of Class Ontology files to general weighted ontologies, but the ACE XML 1.1 version only permits extended tree-based taxonomical class structuring,[23] which at least is significantly more than the simple flat class structures used in the majority of current MIR research.

Class Ontology files do *not* specify the class labels of any specific instances, as this information is instead annotated in Instance Label files (see **Section 7.8**). **Figure 7.9** specifies the DTD for Class Ontology files. A sample complete Class Ontology file is shown in **Code Sample 7.4** in the **Section 7.14** appendix.

The ability to specify hierarchical class structuring has several important benefits. From a musicological perspective, it provides a simple machine readable way of specifying meaningful structuring of classes. From a machine learning perspective, it has the dual advantages of enabling the use of potentially very powerful hierarchical classification methodologies that take advantage of this structuring (e.g., McKay 2004) as well as the use of learning schemes utilizing weighted penalization, such that misclassifications during training into related classes are penalized less severely than misclassifications into unrelated classes.

[23] The tree-based structuring permitted by Class Ontology files is referred to as "extended" because the Class Ontology format and its associated ACE data structures allow the possibility of any given class being descended from multiple parent classes, something that is not permitted in standard tree structures. This can be implemented by specifying the same class name in multiple *parent_class* or *sub_class* clauses.

```
<!ELEMENT taxonomy_file (comments,
                         parent_class+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT parent_class (class_name,
                        sub_class*)>
<!ELEMENT class_name (#PCDATA)>
<!ELEMENT sub_class (class_name,
                     sub_class*)>
```

Figure 7.9 The XML DTD for ACE XML 1.1 Class Ontology files. This DTD precisely defines the information that may appear in Class Ontology files and how it must be formatted. See **Section 7.2.3** for an explanation of XML DTDs. A sample file based on this DTD is shown in **Code Sample 7.4**.

It can be seen from the Class Ontology DTD that flat class structures can be specified simply by listing a set of *parent_class* clauses, each with a *class_name* element used to specify the name of a class. This simple approach can be useful in communicating a list of candidate class labels to an annotator, for example, or for combining a Class Ontology file with a set of labelled instances contained in an Instance Label file in order to ensure that it does not use any unexpected class labels.

The structural aspect of Class Ontology files become apparent when the *sub_class* element is used to specify hierarchical structures of class labels under *parent_class* level classes. Each *parent_class* clause may contain an arbitrary number of *sub_classes,* and each *sub_class* may itself also contain an arbitrary number of *sub_classes,* with the result that a hierarchical class tree of arbitrary depth can be built under each *parent_class*. Each class with no descendants, be it in a *parent_class* or *sub_class* clause, can be referred to as a *leaf class,* and can be used to label instances in Instance Label files.

A Class Ontology file can be linked to an Instance Label file for use during training, or for other reasons, by using *class_name* values in the Class Ontology file that correspond to the *class* values in the Instance Label file.

As a final note, it should be mentioned that a *taxonomy_file* tag is used in the DTD specification rather than a *class_ontology_file* tag. This is for the purpose of legacy compatibility, since Class Ontology files were called Taxonomy files in the deprecated ACE XL 1.0 specification.

7.10 Using ACE XML with new and existing non-jMIR software

A key factor in the effectiveness of any effort to encourage researchers to adopt new standardised file formats is the ease with which they can parse and write to them in their own existing and new software. The simplicity of Weka ARFF files and the ample data structures and processing functionality offered by the Weka code base have certainly contributed to its broad adoption, for example. Although all jMIR software components are of course able to read and write all relevant ACE XML formats, this in itself is not sufficient to encourage the use of ACE XML in other software platforms.

jMIR therefore includes open-source Java libraries in the ACE code package that implement parsing and writing functionality for each of the ACE XML file types, as well as convenient data structures and general processing methods for dealing with the data that is parsed from files. This data can be used and manipulated directly within these libraries, or it can be exported to individual developers' own data structures. ACE's parsing and data structure libraries may be used entirely independently of the ACE meta-learning software itself if desired.

Functionality has also been implemented to automatically convert data in ACE XML data structures into Weka data structures, and vice versa, in order to take advantage of the convenient and well-established functionality built into Weka. This also makes it possible to use Weka data structures as intermediaries for conversion to yet other formats. jMIR also includes utilities for directly translating back and forth between Weka ARFF and ACE XML files, although data that fundamentally cannot be represented in ARFF files is lost when doing so.

As discussed in **Section 7.13**, there are plans to implement ACE XML parsing and processing functionality in other programming languages and to build custom modules for other well-established MIR systems. For the moment, however, the Java implementation of the ACE XML processing functionality makes these libraries as accessible as libraries implemented in any single language can be. Java is platform independent, and the only third-party software used by the ACE XML processing libraries is the open-source Apache Xerces[24] XML-parsing library and the Weka library[25]

[24] xerces.apache.org/xerces-j/
[25] www.cs.waikato.ac.nz/ml/weka/

(and this only if Weka functionality is used), both of which are also implemented in Java. This means that the ACE XML libraries can be easily accessed under any common operating system, with the further advantage that many systems such as Matlab include functionality for accessing Java externals. Furthermore, even if one is for some reason unable to access the ACE XML code libraries, general XML parsers are available in virtually all modern programming languages.

For the purpose of clarity, the overall structure of the Java classes used to parse and process ACE XML files is briefly outlined here. This is only a basic overview, however, and those wishing more details are referred to the ACE API and the well-documented code itself, available at jmir.sourceforge.net.

An important point to note before proceeding to the architectural details of the ACE XML Java classes is that the jMIR components themselves make use of the ACE classes described below when dealing with ACE XML files. This means that any bug fixes or updates to the ACE XML standard only need to be implemented once in these classes to be automatically updated in all of the jMIR components as well.

The parsing code for all ACE XML file formats is contained in the *ace.xmlparsers* Java package. Although changes to the ACE XML standard must be implemented here, users in general should never need to reference these classes directly when incorporating ACE XML functionality into their own code. This is because all of the parsing functionality can be accessed more conveniently from the higher-level classes in the *ace.datatypes* Java package. Having noted this, those users who do wish to have low-level access to the parsed data may wish to examine the *ParseClassificationsFileHandler, ParseDataSetHandler, ParseFeatureDefinitionsHandler* and *ParseTaxonomyFileHandler* classes in the ace.xmlparsers package for parsing Instance Label, Feature Value, Feature Description and Class Ontology ACE XML files respectively.

As noted above, the significant majority of users will prefer to use the ace.datatypes classes, however, which represent the data parsed from ACE XML files at a higher level and include access to file reading, automatic error and consistency checking, file writing, file merging, data translation and data processing functionality, as well as access to the basic data structures used to hold data once it is parsed from ACE XML files. A number of methods are designed specifically with the intention of making relevant data available

in forms convenient to external software, and this data can often be simply imported over to users' own software in the form of simple and well-established data structures.

The overarching Java class in the ace.datatypes package is the *DataBoard*, which provides access to information relating to any of the four ACE XML file types, interpreted either independently or in conjunction with one another. The DataBoard also provides direct access to the *SegmentedClassification, DataSet, FeatureDefinition* and *Taxonomy* Java classes, each of which relates specifically to information stored in Instance Label, Feature Value, Feature Description and Class Ontology ACE XML files, respectively. The ace.datatypes package also contains other Java classes, but these relate more to either ACE XML 2.0 functionality, such as ACE ZIP files (see **Section 7.11.1**), or to ACE machine learning functionality that is not directly relevant to accessing data stored in ACE XML files.

7.11 Current developments: Proposed update to ACE XML 2.0

The ACE XML 1.0 file formats were developed at the beginning of the jMIR project, before any of the jMIR software components had themselves been completed. Minor changes were introduced in version 1.1, the stable version described in the sections above, but the file formats were frozen at this version after the publication of the first jMIR component (McKay et al. 2005). This was necessary because ACE XML is intended for use as a standard, which precludes the incorporation of changes that would render existing software that uses the previous format obsolete.

Of course, certain areas of potential improvement became apparent as more jMIR components were completed. The need for an update to the ACE XML specification also became increasingly apparent as the use of ACE XML as a standardised format for use in the inter-university NEMA[26] project became probable, something that would necessitate a number of changes for the sake of compatibility with other NEMA systems.

As a result, it was decided to design an overhauled version of ACE XML called ACE XML 2.0, which is described in the following sub-sections. It should be stressed that the specified ACE XML 2.0 formats are only proposals, and that the finalization and implementation of these formats is beyond the scope of this document. Given that ACE

[26] nema.lis.uiuc.edu

XML 2.0 is proposed as a standardized set of file formats intended for general use, it is appropriate to first publish proposed changes to the MIR research community for potential modification before finalizing and implementing the changes. ACE XML 1.1 is already implemented and established in all jMIR components and is still the "official" jMIR format at the time of publication of this document.

Some of the changes proposed for ACE XML 2.0 are based on the changing needs of the MIR community since ACE XML 1.1 was established. Others are simply for the sake of improved clarity, simplicity and improved consistency across ACE XML formats. The most fundamental changes, however, are based on the needs imposed by the NEMA project. The NEMA researchers at Queen Mary, University of London are invested in the Music Ontology format (see **Section 7.3.4**), and the NEMA researchers at the University of Illinois at Urbana-Champaign are building the NEMA infrastructure using Meandre,[27] both of which require specific modifications to ACE XML for the sake of full compatibility. For example, the addition of optional URI and other fields to the ACE XML files makes it possible to add RDF handles to ACE XML files if it is necessary to integrate them with file formats such as Music Ontology, while at the same time maintaining the advantages of essentially self-contained structured XML files.

As a consequence of these diverse needs, many of the characteristics of ACE XML 2.0 are the result of compromises between the often competing priorities of different researchers and research groups. The changes implemented in ACE XML 2.0 are intended to meet both these specific needs as well as the original design philosophy of ACE XML as much as possible. The main priority, however, has remained the implementation of formats that are as flexible and general as possible within the specific sphere of MIR classification research.

Many of the changes to ACE XML proposed in the sections below result from very helpful conversations, criticisms and suggestions from other researchers. These include, at McGill University, J. Ashley Burgoyne, Rebecca Fiebrink, Ichiro Fujinaga, Daniel McEnnis and Jessica Thompson, among others. Jessica Thompson in particular deserves special credit for her central role with respect to the ACE ZIP format, as well as for ideas relating to ACE XML in general. And, of course, Ichiro Fujinaga's input was essential

[27] seasr.org/meandre/

throughout the process, from beginning to end. Many other researchers outside McGill University have also contributed very helpful insights, including, Mert Bay, Douglas Eck, Andreas F. Ehmann, Ben Fields, Ian Knopke, Amit Kumar, Paul Lamere, Cyril Laurier, Kevin Page, Yves Raimond, Allen Renear, Mark Sandler, David Tcheng, Karen Wickett and, especially, J. Stephen Downie and Kris West.

It is important to stress that as much effort as possible needs to be made to keep ACE XML 2.0 backwards compatible with ACE XML 1.1, especially in terms of the data structures that the file formats support.

7.11.1 ACE XML 2.0 ZIP files and ACE XML 1.1 and 2.0 Project files

One of the first problems that became apparent with ACE XML was that large groups of XML files associated with a particular research project could be unwieldy to deal with collectively, and could potentially be confusing to new users. Since it is undesirable to combine the four formats into a single format, for reasons discussed in **Sections 7.4** and **7.5**, it was necessary to find a solution that would allow the continued use of multiple discrete files while at the same time simplifying the logistics related to using groups of them together.

The first solution was to devise an ACE XML Project file format that could be used to associate related files together for a given project. This format allows users of an application such as ACE, for example, to simply specify a single Project file, and then rely on the application to itself automatically open all of the files referred to by this Project file, thus increasing user convenience significantly.

A prototype Project file format was developed under the ACE XML 1.1 framework and implemented in the ACE Java package, but never finalized as a standard or implemented in the other jMIR components. The DTD of this prototype ACE XML 1.1 Project file is shown in **Figure 7.10**, and a revised ACE XML 2.0 version is shown in **Figure 7.11**. A sample ACE XML 1.1 Project file is shown in **Code Sample 7.5** in **Section 7.14**, and a sample ACE XML 2.0 Project file is shown in **Code Sample 7.6**.

```
<!ELEMENT ace_project_file (comments,
                            taxonomy_path,
                            feature_definitions_path,
                            feature_vectors_path,
                            model_classifications_path,
                            gui_preferences_path,
                            classifier_settings_path,
                            trained_classifiers_path,
                            weka_arff_path)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT taxonomy_path (#PCDATA)>
<!ELEMENT feature_definitions_path (path*)>
<!ELEMENT feature_vectors_path (path*)>
<!ELEMENT model_classifications_path (path*)>
<!ELEMENT gui_preferences_path (#PCDATA)>
<!ELEMENT classifier_settings_path (#PCDATA)>
<!ELEMENT trained_classifiers_path (#PCDATA)>
<!ELEMENT weka_arff_path (#PCDATA)>
<!ELEMENT path (#PCDATA)>
```

Figure 7.10 The XML DTD for the prototype ACE XML 1.1 Project file format. This DTD precisely defines the information that may appear in Project files and how it must be formatted. See **Section 7.2.3** for an explanation of XML DTDs. A sample file based on this DTD is shown in **Code Sample 7.5**.

The *feature_vectors_path, feature_definitions_path, model_classifications_path* and *taxonomy_path* elements allow references to be made to external ACE XML Feature Value, Feature Description, Instance Label or Class Ontology files, respectively. Zero to many files of each type may be referred to, except in the case of Class Ontology files, for which only zero or one files may be referenced. Weka ARFF files can also be referred to using the *weka_arff_path* element if ACE XML files are unavailable for a certain dataset.

Preference files (in as of yet unspecified formats) for the ACE meta-learning application can also be specified using the *gui_preferences_path* and *classifier_settings_path* elements. Trained classification models can be referenced via the *trained_classifiers_path* element.

```
<!ELEMENT ace_xml_project_file_2_0 (comments?,
                                    feature_value_id,
                                    instance_label_id,
                                    class_ontology_id,
                                    feature_description_id,
                                    weka_arff_id?,
                                    trained_model_id?,
                                    uri?)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT feature_value_id (path*)>
<!ELEMENT instance_label_id (path*)>
<!ELEMENT class_ontology_id (#PCDATA)>
<!ELEMENT feature_description_id (path*)>
<!ELEMENT weka_arff_id (#PCDATA)>
<!ELEMENT trained_model_id (#PCDATA)>
<!ELEMENT uri (path*)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT path (#PCDATA)>
```

Figure 7.11 The proposed ACE XML 2.0 update to the DTD of the Project file format. See **Section 7.2.3** for an explanation of XML DTDs. A sample file based on this DTD is shown in **Code Sample 7.6**.

As can be seen by a comparison of **Figures 7.10** and **7.11**, most of the changes are in the specific terminology used in the element tags and in the order in which they appear. These changes are proposed for purposes of clarity and consistency with other ACE XML file formats. One of the few fundamental changes is the removal of the *ace_preferences_id* and *classifier_settings_id* elements. This was done because it is in general desirable to separate the ACE XML file formats from the ACE meta-learning application or any other particular jMIR components. A new *uri* element is also added so that references can be made to external resources of any type. This *uri* element includes an optional *predicate* attribute in order to make it possible to specify RDF-like triples, such that the contents of the *uri* clause indicate the object, the clause containing the *uri* sub-clause is the subject, and the *predicate* attribute specifies the relationship between the two.

Another change that has been considered but rejected, at least for the moment, is the ability to list multiple Class Ontology files. The was not done since the merging of different ontologies could lead to significant inconsistencies if not supervised carefully, and it would probably be safer to require that such rare operations be performed manually.

Although the Project file does make the use of multiple ACE XML files together significantly more convenient, it is an imperfect solution. Users must still maintain the individual files, and must be careful not to delete, rename or move them without making appropriate changes in the Project file.

In order to fully address this problem, it was decided to devise an ACE XML ZIP file format. This format is inspired by the Microsoft Office 2007 Open XML File Format,[28] which stores each Microsoft Office "document" as sets of XML files packaged into a single ZIP file. Similarly, ACE XML ZIP files consist of sets of ACE XML (or other) files that are packaged together into a single ZIP file.

This approach retains the advantages of maintaining separate files, as discussed in **Sections 7.4** and **7.5**, since each ACE XML file stored in an ACE XML ZIP file remains self-contained and can be easily extracted from the ZIP file and used on its own or with other projects. At the same time, this approach enables multiple related ACE XML files to be packaged into a single ZIP file, so no housekeeping of external files is required. ZIP files in particular are an especially appropriate format because there are many free applications and code libraries that can be used to access or store data in them.

Another significant advantage of using ZIP files is that they are compressed formats, which means that they can dramatically reduce space and bandwidth requirements. This is significant, as ACE XML files can be quite large, particularly in cases when many windowed features are extracted from large collections of data. Compression rates as high as 83% have been observed when compressing Feature Value files, although the amount of compression depends on the particular data that is being compressed.

Open-source code for using ACE ZIP files has already been implemented by Jessica Thompson. This code is integrated into the ACE code package, and can be accessed via the ACE command line interface and general API. Work on incorporating this functionality into the ACE GUI as well is also currently underway.

Each ACE XML ZIP file is associated with a single Project file, which is either specified by the user when the ACE ZIP file is created, or auto-generated by the ACE ZIP processing code when ACE XML files are added to an existing ACE ZIP file. So, although an ACE ZIP file can hold many files of any type, each ACE ZIP file always has

[28] msdn.microsoft.com/en-us/library/aa338205.aspx

exactly one default ACE XML Project file contained within it and only one or zero Class Ontology files specified by this Project file. In order for this to work properly, each ACE ZIP file has a simple hidden file called *project.sp* automatically generated and added to it that specifies the name of the default ACE XML Project file for the ZIP file. Users do not ever interact directly with this file, however, or need to be aware of it.

An ACE XML ZIP file may be automatically generated from an ACE XML Project file using the ACE ZIP processing software. This software automatically packages all files referred to in the Project file, along with the Project file itself, into the ZIP file. Then, when the ZIP file is later opened, the Project file is automatically decompressed and parsed so that the files contained in the ZIP file can themselves be automatically accessed, decompressed and properly interpreted by the ACE code. When the project and other files contained in the ZIP file are decompressed into a new directory, the ACE zip-processing software automatically updates the Project file to reflect their new file paths.

Alternatively, users may specify a list of files or simply a directory containing ACE XML files. The ACE software will then automatically package the appropriate files into a new ACE ZIP file, along with an ACE XML Project file that is auto-generated based on the specified ACE XML files. If desired, the ACE API and command line can also be used to add or extract individual or all files from ACE ZIP files.

ACE XML ZIP and Project file functionality will be incorporated into the jMIR components other than ACE once the ACE XML 2.0 specification is finalized via consultation with the NEMA researchers and the MIR community.

7.11.2 Proposed ACE XML 2.0 updates to Feature Value files

The DTD of the current ACE XML 1.1 Feature Value file format is shown in **Figure 7.6**, and the proposed updated DTD for the ACE XML 2.0 Feature Value format is shown in **Figure 7.12**. A sample ACE XML 2.0 Feature Value file is shown in **Code Sample 7.7** in the **Section 7.14** appendix. The *comments* clause of this sample file provides descriptive instructions on how to use the file format. The proposed changes to the ACE

XML 2.0 Feature Value format relative to the current ACE XML 1.1 version are as follows:[29]

- The *feature_vector_file* element is renamed to *ace_xml_feature_value_file_2_0* for the purpose of distinguishing between the ACE XML 2.0 and 1.1 versions.
- The *data_set* element is renamed to *instance* for the purposes of clarity and generality.
- The *data_set_id* element is renamed to *instance_id* for the purposes of consistency, clarity and generality.
- The *feature* element is renamed to *f* in order to reduce file size.
- The *name* element is renamed to *id* in order to reduce file size.
- The *section* element is renamed to *s* in order to reduce file size.
- The *start* and *stop* attributes are renamed *b* and *e* (abbreviations for *beginning* and *end*) in order to reduce file size, and are now obligatory in *s* elements in order to enforce proper file construction.
- Features that correspond to a precise time (or other) coordinate value in an instance rather than intervals of time (or other) coordinates or instances as a whole can be specified with the new *precise_coord* element and its associated *coord* attribute.
- The new optional *coord_units* element can be used in each instance to specify the units used for the coordinate indicators for both sub-sections of and precise coordinates in instances.
- The new optional *extractor* element can be used to specify the name of the feature extraction software used to extract each feature for an instance. A separate *extractor* clause is used for each feature. The contents of an *extractor* clause

[29] Several of the tags in the ACE XML 2.0 Feature Value file have been abbreviated. Feature Vector files in particular have a tendency to grow very long, especially when there are instances with many analysis windows and when many features are extracted. So, although in general abbreviations have been avoided in ACE XML for the sake of clarity, some abbreviations have been used in Feature Value files for those tags that are likely to be repeated often in order to avoid exorbitant file sizes.

indicate the name of the feature extractor, and the obligatory *fname* attribute indicates the name of the feature that is to be associated with this extractor. This arrangement allows scenarios where different feature extractors are used to extract the same feature for different instances, as well as scenarios where different features are extracted by different feature extractors for the same instance.

- Feature values consisting of arrays with an arbitrary number of dimensions may now be expressed, as compared to the ACE XML 1.1 limitation to only one-dimensional vectors of feature vales. Sparse arrays, or arrays missing some entries, are now supported as well, something that can be important for space efficient and flexible data representation. ACE XML 2.0 currently supports four alternative approaches to representing feature values and arrays, each with its own relative strengths and weaknesses with respect to the number of dimensions that can be represented, file size, ability to efficiently represent sparse data and human readability:
    - In the case of feature values consisting of only one value or feature vectors consisting of only one dimension, a methodology similar to the one that was used in ACE XML 1.1 (i.e. *f* clauses containing the *v* element) may be used in ACE XML 2.0 as well.
    - Arrays with any number of dimensions may be expressed using JSON (*JavaScript Object Notation*)[30] array notation. JSON is a well-established and relatively human readable text-based data interchange format for representing simple data structures. JSON arrays are expressed using simple square bracket notation, enclosed in *vj* clauses in ACE XML 2.0. So, for example, a feature vector of size three consisting of the numbers one, two and three would be represented as *<vj>[1,2,3]</vj>*. JSON arrays can be nested in order to represent arrays of arbitrary dimensionality. So, for example, a table with two identical rows each containing the values one, two and three would be represented as *<vj>[[1,2,3],[1,2,3]]<vj>*. A similar approach could have been achieved

[30] json.org

by using nested XML elements, but the JSON representation is more compact and more human readable for large arrays. There are also JSON parsing libraries available in many languages that can parse such arrays quickly, which offloads some of the work from that would otherwise need to be performed by an ACE XML parser. A disadvantage of the JSON approach, however, is that JSON is not ideally suited to efficiently representing sparse arrays. Also, JSON is less human readable than some of the alternative approaches. Ultimately, however, it is a compromise that enables potentially very large arrays to be represented relatively compactly while still being relatively readable, at least compared to binary data.

- Explicitly indexed arrays may be used as an alternative representation in the case of feature values consisting of only one number, feature vectors consisting of one dimension or feature arrays consisting of two to ten dimensions. This approach involves specifying coordinates using the *d0* to d*9* attributes in *vd* clauses, as an alternative to *v* clauses. If there is only one coordinate (i.e., a feature vector), then only the *d0* attribute would be used, if there is a three-dimensional array then the d0, d1 and d2 attributes would be used, and so on. This approach is moderately space efficient, can represent sparse arrays and is easily human readable. There is a limitation to only ten dimensions, but each of these may consist of vectors of any size, and very few features used in MIR need arrays with more than ten dimensions. Although it would be ideal to have such an approach for N dimensions, it is not possible to specify an arbitrary number of dimension attributes in an XML DTD schema. To give an example, the *JSON* feature vector of *[1,2,3]* would be represented as *<vd d0="0">1</vd><vd d0="1">2</vd><vd d0="2">3</vd>*, and the JSON array of *[[1,2,3],[1,2,3]]* would be represented as *<vd d0="0" d1="0">1</vd><vd d0="0" d1="1">2</vd><vd d0="0" d1="2">3</vd><vd d0="1" d1="0">1</vd><vd d0="1" d1="1">2</vd><vd d0="1" d1="2">3</vd>*.

- The final option permitted by ACE XML 2.0 is to represent arrays with any number of dimensions using *vs* clauses. Each *vs* clause contains one *d* element for each dimension, and this *d* element is used to specify the coordinate value its corresponding dimension. Each *vs* clause also contains a single *v* clause to specify the feature value for the array at the corresponding coordinate. To give an example, the element of the *JSON* feature array *[[1,2,3],[4,5,6]]* with a value of 6 would be represented as *<vs><d>1</d><d>2</d><v>6</v></vs>*. This approach has the advantage that it can be used to express arrays with any number of dimensions and, unlike the JSON approach, can also efficiently represent sparse arrays as well as represent data in a more human readable way. It is significantly less compact than the JSON approach for complete arrays, however, and the ACE XML encoder must ensure that the correct number of *d* elements are present for each value and that they consistently appear in the correct order.

- Data types may now optionally be explicitly specified for each feature via the optional *type* attribute of the *f* (formerly *feature*) element. Types of *int, double, float, complex* and *string* are permitted. Although this typing is not necessary for the jMIR components, it is sometimes necessary for other applications, so it is useful to incorporate it into the ACE XML formats so that it can be used when needed. The type will typically be assumed to be *double* if it is not specified in any given *f* clause, but this not an intrinsic assumption of the Feature Value format. It is important to note that feature types may also be specified in an associated Feature Description file, in which case the feature types in the Feature Value file should either correspond to the types specified in the Feature Description file or, for the sake of brevity, be omitted in the Feature Value file.

- The *comments* element is now optional.

- It is now possible to specify other files that are related to the Feature Value file using the *related_resources* element. These can either be explicitly referenced Feature Value, Feature Description, Instance Label, Class Ontology or Project

files, respectively referenced via *feature_value_file, feature_description_file, instance_label_file, class_ontology_file* or *project_file* elements, or they can be resources of arbitrary types referenced with *uri* elements. Note that references referred to via a *related_resources* element are for informal informational purposes only from the perspective of the jMIR components, and are *not* substitutes for references in ACE XML Project (see **Section 7.11.1**).

- Optional *uri* elements (and their associated *predicate* attributes) may also be used within any instance (*instance*), section (*s*), precise coordinate (*precise_coord*) or feature (*f*) clauses. These are not intended for use by the jMIR components, but may be used by other software to access external resources whenever appropriate.

```
<!ELEMENT ace_xml_feature_value_file_2_0 (comments?, related_resources?,
                                          instance+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file*,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT instance (instance_id,
                    uri*,
                    extractor*,
                    coord_units?,
                    s*,
                    precise_coord*,
                    f*)>
<!ELEMENT instance_id (#PCDATA)>
<!ELEMENT extractor (#PCDATA)>
<!ATTLIST extractor fname CDATA #REQUIRED>
<!ELEMENT coord_units (#PCDATA)>
<!ELEMENT s (uri*,
             f+)>
<!ATTLIST s b CDATA #REQUIRED
            e CDATA #REQUIRED>
<!ELEMENT precise_coord (uri*,
                         f+)>
<!ATTLIST precise_coord coord CDATA #REQUIRED>
<!ELEMENT f (fid,
             uri*,
             (v+ | vd+ | vs+ | vj))>
<!ATTLIST f type (int | double | float | complex | string) #IMPLIED>
<!ELEMENT fid (#PCDATA)>
<!ELEMENT v (#PCDATA)>
<!ELEMENT vd (#PCDATA)>
<!ATTLIST vd d0 CDATA #REQUIRED d1 CDATA #IMPLIED d2 CDATA #IMPLIED
             d3 CDATA #IMPLIED d4 CDATA #IMPLIED d5 CDATA #IMPLIED
             d6 CDATA #IMPLIED d7 CDATA #IMPLIED d8 CDATA #IMPLIED
             d9 CDATA #IMPLIED>
<!ELEMENT vs (d+,
              v)>
<!ELEMENT d (#PCDATA)>
<!ELEMENT vj (#PCDATA)>
```

Figure 7.12 The proposed ACE XML 2.0 update to the DTD of the Feature Value file format. See **Section 7.2.3** for an explanation of XML DTDs. A sample file based on this DTD is shown in **Code Sample 7.7**.

7.11.3 Proposed ACE XML 2.0 updates to Feature Description files

The DTD of the current ACE XML 1.1 Feature Description file format is shown in **Figure 7.7**, and the proposed updated DTD for the ACE XML 2.0 Feature Description format is shown in **Figure 7.13**. A sample ACE XML 2.0 Feature Description file is also shown in **Code Sample 7.8** in the **Section 7.14** appendix. The *comments* clause of this sample file provides descriptive instructions on how to use the file format. The proposed changes to the ACE XML 2.0 Feature Description format relative to the current ACE XML 1.1 version are as follows:

- The *feature_key_file* element is renamed to *ace_xml_feature_description_file_2_0* for the purpose of distinguishing between the ACE XML 2.0 and 1.1 versions.
- The *name* element is renamed to *fid* to make it consistent with the Feature Value format.
- The *is_sequential* element is replaced by the *scope* element. The old *is_sequential* element could only be used to specify whether a feature could be extracted only over a whole instance or only over sub-sections of an instance. The new *scope* element makes it possible to specify that a feature may be extracted for an instance as a whole, for sub-sections of an instance, from only a precise point in an instance (e.g., a moment in time) or from any combination of these. This information is expressed via the *overall, sub_section* and *precise_coord* attributes respectively, which may each have values of either *true* or *false*. This information must all be specified for all features. It is possible to enter comment data in the *scope* clause, but this data will have no technical meaning.
- The *parallel_dimensions* element is replaced by the *dimensionality* element in order to accommodate the new ability to represent feature value arrays of arbitrary dimensionality in ACE XML 2.0 Feature Value files, as opposed to the ACE XML 1.0 limitation to one-dimensional feature vectors. The old *parallel_dimensions* element was only used to specify the size of feature vectors, but the *dimensionality* element is used to specify the number of different dimensions of the coordinate system for the feature (e.g., one for a feature vector, two for a table structure, etc.) as well as the size of each of the dimensions. The

orthogonal_dimensions attribute indicates the former, and *size* clauses within the *dimensionality* clause are used to indicate the size of each of these (e.g., one size clause each for the number of rows and the number of columns in a table structure). The *dimensionality* element may also be omitted if a particular feature can have variable number of coordinate dimensions, and *size* clauses may be omitted as well if they also vary.

- The optional *data_type* element and its *type* attribute are added in order to allow the specification of the particular data type for a given feature. Specifically, the permitted types are *int, double, float, complex* and *string,* and one of these must be specified in the *type* tag. Although this typing is not necessary for the jMIR components, it is sometimes necessary for other applications, so it is useful to incorporate the option of using it into the ACE XML formats so that it can be used when needed. The type will typically be assumed to be *double* if it is not specified in any given *feature* clause, but this not an intrinsic assumption of the Feature Description format. Although data types may be specified in Feature Value files, it is preferable to do so in Feature Description files, which take priority. Note that comments may be entered in the *data_type* clause itself, but they do not have any technical meaning.

- It is now possible to specify feature parameters using the optional *parameter* element and its associated *parameter_id, description* and *value* elements. This could be used for specifying the roll-off point for the Spectral Roll-off feature, for example. A separate *parameter* clause is used for each parameter of a feature, the *parameter_id* element is used to identify the parameter uniquely, the *description* element can be used to describe the parameter in general, and the *value* element can be used to express numerical parameter values.

- Global parameters may also be specified for all features in the Feature Description file using the *global_parameter* element. This is useful for specifying overall pre-processing of audio files before features are extracted, for example, such as down sampling or normalization. The mechanics of the global_parameter are the same as those of the *parameter* element.

- A new optional *related_feature* clause may be used to specify other features that are related to any given feature. This could be used, for example, to note that one feature is an alternative implementation of another. The *fid* element in the related feature clause should be used to specify the name of a feature specified in the *fid* clause of another feature. The *relation_id* element can be used to specify a specific externally defined type of relationship, and the *explanation* element can be used to provide a qualitative description of the relationship.
- The *comments* element is now optional.
- It is now possible to specify other files that are related to the Feature Description file using the *related_resources* element. This is implemented in a fashion identical to that described for Feature Value files in **Section 7.11.2**.
- Optional *uri* elements (and their associated *predicate* attributes) may also be used within any *feature, dimensionality, parameter* or *related_feature* clause. These are not used by the jMIR components, but may be used by other software to access external resources if appropriate.

```
<!ELEMENT ace_xml_feature_description_file_2_0 (comments?,
                                                related_resources?,
                                                global_parameter*,
                                                feature+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file*,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT feature (fid,
                   description?,
                   related_feature*,
                   uri*,
                   scope,
                   dimensionality?,
                   data_type?,
                   parameter*)>
<!ELEMENT fid (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT related_feature (fid,
                           relation_id?,
                           uri*,
                           explanation?)>
<!ELEMENT relation_id (#PCDATA)>
<!ELEMENT explanation (#PCDATA)>
<!ELEMENT scope (#PCDATA)>
<!ATTLIST scope overall (true|false) #REQUIRED
                sub_section (true|false) #REQUIRED
                precise_coord (true|false) #REQUIRED>
<!ELEMENT dimensionality (uri*,
                          size*)>
<!ATTLIST dimensionality orthogonal_dimensions CDATA #REQUIRED>
<!ELEMENT size (#PCDATA)>
<!ELEMENT data_type (#PCDATA)>
<!ATTLIST data_type type (int | double | float | complex | string) #REQUIRED>
<!ELEMENT global_parameter (parameter_id,
                            uri*,
                            description?,
                            value?)>
<!ELEMENT parameter (parameter_id,
                     uri*,
                     description?,
                     value?)>
<!ELEMENT parameter_id (#PCDATA)>
<!ELEMENT value (#PCDATA)>
```

Figure 7.13 The proposed ACE XML 2.0 update to the DTD of the Feature Description file format. See **Section 7.2.3** for an explanation of XML DTDs. A sample file based on this DTD is shown in **Code Sample 7.8**.

7.11.4 Proposed ACE XML 2.0 updates to Instance Label files

The DTD of the current ACE XML 1.1 Instance Label file format is shown in **Figure 7.8**, and the proposed updated DTD for the ACE XML 2.0 Instance Label format is shown in **Figure 7.14**. A sample ACE XML 2.0 Instance Label file is also shown in **Code Sample 7.9** in the **Section 7.14** appendix. The *comments* clause of this sample file provides descriptive instructions on how to use the file format. The proposed changes to the ACE XML 2.0 Instance Label format relative to the current ACE XML 1.1 version are as follows:

- The *classifications_file* element is renamed to *ace_xml_instance_label_file_2_0* for the purpose of distinguishing between the ACE XML 2.0 and 1.1 versions.
- The *data_set* element is renamed to *instance* for the purposes of clarity and generality.
- The *data_set_id* element is renamed to *instance_id* for the purposes of consistency, clarity and generality.
- The *info_type* atribute is renamed to *info_id* for the purpose of consistency with other element and attribute identifiers. Also, *misc_info* clauses now contain *info_id* and *info* elements, and there are no longer any attributes for the *misc_info* element. This is to enable the addition an arbitrary number of *uri* annotations to *misc_info* clauses, as noted below.
- For similar reasons, class labels are now specified within a *class_id* element contained in a *class* clause.
- The *role* element is now an optional attribute of the *instance* element instead of an element itself. This makes it possible to explicitly constrain its possible values to *training, testing* or *predicted.*
- A new optional *related_instance* clause may be used to specify other instances that are related to any given instance. This could be used, for example, to note that one recording is a cover of another. The *instance_id* element in the related instance clause should be used to specify the name of an instance specified in the *instance_id* clause of another feature. The *relation_id* element can be used to

specify a specific externally defined type of relationship, and the *explanation* element can be used to provide a qualitative description of the relationship.

- The *start* and *stop* elements for denoting coordinate ranges within an instance are replaced with the *begin* and *end* attributes of the *section* element. This change is in order to maintain consistency with the *b* and *e* attributes of ACE XML 2.0 Feature Value files.

- Class labels that correspond to a precise time (or other) coordinate rather than intervals of time (or other) coordinates or instances as a whole can be specified with the new *precise_coord* element and its associated *coord* attribute.

- The new optional *coord_units* element can be used in each instance to specify the units used for the coordinate indicators for both sub-sections of and precise coordinates in instances.

- The *classification* element is removed for the sake of improving file simplicity. Section labelling and overall instance labelling now simply occur directly within an *instance* clause rather than within a *classification* clause within an *instance* clause.

- The *class* element has a new *weight* attribute that can be use to specify proportional support for a class relative to other classes when more than one class apply. So, for example, a given musical recording might be labelled with the *Blues* genre with a specified weight of *2* as well as with the *Jazz* genre with a specified weight of *1*. Depending on context, this could be intended to mean either that the recording is a member of both the *Blues* and *Jazz* genres, but the influence of the former is twice that of the latter, or it could mean that a classifier is unsure whether the piece is *Blues* or *Jazz,* but believes that the former label is twice as likely as the latter. If the *weight* attribute is not specified for a class, it is assigned a value of *1* by default. All weight values are proportional, so the absolute value of a weight has no meaning other than its value relative to the weights of other class labels with the same scope.

- The optional *source_comment* attribute may now be used with the class element. This permits the specification of whether an instance was labeled by a machine, human expert, survey, etc.
- The *comments* element is now optional.
- It is now possible to specify other files that are related to the Instance Label file using the *related_resources* element. This is implemented in a fashion identical to that described for Feature Value files in **Section 7.11.2**.
- Optional *uri* elements (and their associated *predicate* attributes) may be added to any *instance, related_instance, misc_info, section, precise_coord* or *class* clause. These are not used by the jMIR components, but may be used by other software to access external resources when appropriate.

```
<!ELEMENT ace_xml_instance_label_file_2_0 (comments?,
                                           related_resources?,
                                           instance+)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file *,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT instance (instance_id,
                    misc_info*,
                    related_instance*,
                    uri*,
                    coord_units?,
                    section*,
                    precise_coord*,
                    class*)>
<!ATTLIST instance role (training | testing | predicted) #IMPLIED>
<!ELEMENT instance_id (#PCDATA)>
<!ELEMENT related_instance (instance_id,
                            relation_id?,
                            uri*,
                            explanation?)>
<!ELEMENT relation_id (#PCDATA)>
<!ELEMENT explanation (#PCDATA)>
<!ELEMENT misc_info (info_id,
                     uri*,
                     info)>
<!ELEMENT info_id (#PCDATA)>
<!ELEMENT info (#PCDATA)>
<!ELEMENT coord_units (#PCDATA)>
<!ELEMENT section (uri*,
                   class+)>
<!ATTLIST section begin CDATA #REQUIRED
                  end CDATA #REQUIRED>
<!ELEMENT precise_coord (uri*,
                         class+)>
<!ATTLIST precise_coord coord CDATA #REQUIRED>
<!ELEMENT class (class_id,
                 uri*)>
<!ATTLIST class weight CDATA "1">
<!ATTLIST class source_comment CDATA #IMPLIED>
<!ELEMENT class_id (#PCDATA)>
```

Figure 7.14 The proposed ACE XML 2.0 update to the DTD of the Instance Label file format. See **Section 7.2.3** for an explanation of XML DTDs. A sample file based on this DTD is shown in **Code Sample 7.9**.

7.11.5 Proposed ACE XML 2.0 updates to Class Ontology files

The DTD of the current ACE XML 1.1 Class Ontology file format is shown in **Figure 7.9**, and the proposed updated DTD for the ACE XML 2.0 Class Ontology format is shown in **Figure 7.15**. A sample ACE XML 2.0 Class Ontology file is also shown in **Code Sample 7.10** in the **Section 7.14** appendix. The *comments* clause of this sample file provides descriptive instructions on how to use the file format. The proposed changes to the ACE XML 2.0 Class Ontology format relative to the current ACE XML 1.1 version are as follows:

- The *taxonomy_file* element is renamed to *ace_xml_class_ontology_file_2_0* for the purpose of distinguishing between the ACE XML 2.0 and 1.1 versions.
- The *class_name* element is renamed to *class_id* in order to make it consistent with the naming conventions used in the other ACE XML 2.0 formats.
- The *parent_classs* element is removed because it made an implied hierarchical organization of classes obligatory, which is not always appropriate. Information about each class is now contained in a *class* clause, regardless of the presence or absence of a hierarchical structure.
- Since it is sometimes useful to be able to specify hierarchical class structuring, the optional *sub_class* element is repurposed so that it can be used to reference one or more other classes that are hierarchical subordinates to the class whose clause contains the *sub_class* element. Such sub-classes must now also be separately declared in their own *class* clauses. Tree structures can be built by referring to subordinate classes using *sub_class* clauses, then referring to further subordinate classes at the next depth level of the tree in the *sub_class* clauses of these classes, and so on.
- As an alternative to hierarchical class structuring, the ACE XML 2.0 Class Ontology format now allows more general ontological relationships to be specified between classes using the *related_class* element. A relationship specified from one class to another with this element is unidirectional, unless the same relationship is explicitly specified from the second class to the first class as well in

the second class' declaration. The *relation_id* element may be used to specify the meaning of the relationship using some externally defined keyword if desired.

- Weights may be assigned to both *related_class* and and *sub_class* elements using the *weght* attribute, which defaults to 1 unless specified. Relating to this, the global *weights_relative* attribute of the *ace_xml_class_ontology_file_2_0* element must be specified as either *true* or *false.* If it is true, then the weights for each class will be normalized upon parsing, if not they will be interpreted as is.
- The *explanation* element can be used to provide qualitative explanations of any class connections in the class ontology.
- The *misc_info* element may now be used to specify miscellaneous metadata about each class. Each *misc_info* clause contains *info_id* and *info* elements to specify some externally defined unique keyword for the metadata field and the metadata itself, respectively.
- The *comments* element is now optional.
- It is now possible to specify other files that are related to the Class Ontology file using the *related_resources* element. This is implemented in a fashion identical to that described for Feature Value files in **Section 7.11.2**.
- Optional *uri* elements (and their associated *predicate* attributes) may also be used within any *class, related_class* or *sub_class* clause. These are not used by the jMIR components, but may be used by other software to access external resources when appropriate.

```
<!ELEMENT ace_xml_class_ontology_file_2_0 (comments?,
                                           related_resources?,
                                           class+)>
<!ATTLIST ace_xml_class_ontology_file_2_0 weights_relative (true|false)
                                                            #REQUIRED>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT related_resources (feature_value_file*,
                             feature_description_file *,
                             instance_label_file*,
                             class_ontology_file*,
                             project_file*,
                             uri*)>
<!ELEMENT feature_value_file (#PCDATA)>
<!ELEMENT feature_description_file (#PCDATA)>
<!ELEMENT instance_label_file (#PCDATA)>
<!ELEMENT class_ontology_file (#PCDATA)>
<!ELEMENT project_file (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ATTLIST uri predicate CDATA #IMPLIED>
<!ELEMENT class (class_id,
                 misc_info*,
                 uri*,
                 related_class*,
                 sub_class*)>
<!ELEMENT class_id (#PCDATA)>
<!ELEMENT misc_info (info_id,
                     uri*,
                     info)>
<!ELEMENT info_id (#PCDATA)>
<!ELEMENT info (#PCDATA)>
<!ELEMENT related_class (class_id,
                         relation_id?,
                         uri*,
                         explanation?)>
<!ATTLIST related_class weight CDATA "1">
<!ELEMENT relation_id (#PCDATA)>
<!ELEMENT explanation (#PCDATA)>
<!ELEMENT sub_class (class_id,
                     relation_id?,
                     uri*,
                     explanation?)>
<!ATTLIST sub_class weight CDATA "1">
```

Figure 7.15 The proposed ACE XML 2.0 update to the DTD of the Class Ontology file format. See **Section 7.2.3** for an explanation of XML DTDs. A sample file based on this DTD is shown in **Code Sample 7.10**.

7.12 Summary of original contributions

This chapter provides a critical analysis and overview of existing file formats that are used by MIR researchers for data mining and music classification. It also provides an original and previously lacking set of design priorities for consideration when devising new file formats in this domain.

The four original ACE XML 1.1 file formats are presented as an implementation of these design priorities. The jMIR components can all use these file formats to communicate with each other, and a general API is provided as part of the ACE package so that ACE XML functionality can be easily integrated into other software. General ACE XML file processing utilities are also provided in the jMIRUtilities software.

Original prototypes for the four ACE XML 2.0 file formats are also presented to the MIR research community for general discussion and comment. These formats expand upon the ACE XML 1.1 by adding further expressive power and flexibility, and it is hoped that the MIR community will collaborate to improve upon them and eventually adopt them.

7.13 Future research

The clear priority for future research is to collect input from the MIR community on changes and improvements to the ACE XML 2.0 formats.

One specific issue that needs to be addressed is that Feature Value files can end up being very large, particularly when many features are extracted for small windows over many instances. Although ACE ZIP packaging does address this issue to an extent, it in turn introduces additional processing overhead when compressing and decompressing the files. One solution would be to represent feature values, including arrays, in binary rather than in text. Unfortunately, this would entirely undermine ACE XML's design philosophy of permitting human readability. Furthermore, there are many alternative ways of representing values and arrays in binary, which could cause incompatibilities if different users encode binary in different ways and then distribute the Feature Value files to others. This could also pose problems with respect to data longevity. One final related issue is that XML validation processing cannot be applied to binary representations, thereby increasing the work that must be performed directly by ACE XML parsers. All of this having been said, it is always desirable to avoid excessively large files, so this is something that needs to be considered further.

The representation of feature values consisting of N-dimensional arrays is another issue that needs to be considered further. All of the options supported by ACE XML 2.0 can require a significant amount of space to represent large N-dimensional arrays, so the

addition of a binary representation option to the specification might be useful in cases where this could be a concern and where human readability is not a priority.

Another issue that would be appropriate to pursue would be the design of some external file format for specifying a vocabulary that could be defined with respect to units that are specified using the *coord_units* element. This would, for example, provide a way to automatically make a recording annotated with times stamps based on milliseconds compatible with another annotated based on samples (if the sampling rate is known, of course).

It would also be useful to take advantage of existing metadata standards in general. One sample approach might be to integrate Dublin Core RDF functionality into the ACE XML formats.[31]

Complex numbers also represent something that needs to be considered further. Under the current implementation, complex numbers can be simply represented as feature vectors of size 2. Unfortunately, this does not explicitly distinguish them from any other feature vector of size 2. Ideally, one would like to have some way of typing complex numbers in the same way that doubles or strings can be typed. Unfortunately, there is no complex primitive type in XML, nor is there a complex primitive in many of the most common programming languages, including Java. Furthermore, with respect to array representation, JSON does not allow the explicit use of complex numbers in arrays, beyond the current implicit practice of using general sub-arrays of size 2. This limitation of JSON is in itself a strong argument, in particular, against using an explicit complex number type. Furthermore, in most cases feature values are simply treated as features during machine learning, so it is often not relevant if a complex feature value is explicitly typed as such. Nevertheless, it could be important to more sophisticated learning techniques to have an explicit complex data type, so further though needs to be put into incorporating explicit complex typing into ACE XML Feature Value and Feature Description files, in both Cartesian and polar forms.

Another issue to address is multilingual support. XML is generally Unicode-based, so there is built-in support for many character sets, but testing of the ACE XML processing software to date has focused on English and French data, a scope that needs to be

[31] dublincore.org/schemas/rdfs/ and dublincore.org/documents/dc-rdf/index.shtml

expanded to ensure that the file formats are ready for wide international use. Further thought is needed in this area.

It would also be helpful to define specific standards for setting unique keys for the ACE XML ID fields that are used to merge data stored in the four different file formats. The onus is currently on the data encoder to ensure uniqueness, something that could potentially result in missed correspondences and conflicts. One possible solution might be to use something such as Music Brainz[32] IDs as primary keys, for example, but solutions like this tend to focus only on specific types of data, namely audio in this particular case. Extracting Music Brainz IDs for something like MIDI files is not tenable. Further thought is needed in this area as well.

There would also be advantages to writing versions of each of the ACE XML DTDs using an alternative XML schema that would specify the same formats using an alternative methodology. Such schemas would co-exist with the current DTDs, due to the advantages of DTDs expressed in **Section 7.5** with respect to accessibility to new ACE XML users. The main advantage offered by the use of an alternative more expressive schema format is the offloading of some of the file validation load from the ACE XML parsers to the general XML parsing libraries, since constraints on what could validly be contained in XML clauses could be more precisely defined.

Once the ACE XML 2.0 file formats are finalized, the next step will be to update the ACE parsers, processing utilities and data structures. The ACE API is ready as is for the porting of ACE XML 1.1 functionality to external software, but significant updates will be necessary to make it ACE XML 2.0 ready.

The implementation of reading, writing and processing ports for specific existing widely used MIR systems like Marsyas and CLAM and for general programming environments like Matlab and C++ will likely do much to encourage the wide adoption of the ACE XML formats by making them more easily accessible. Translation software will also be needed to translate information stored in existing formats, including ACE XML 1.1, into ACE XML 2.0.

Another area for future research is the development of a standardized way of storing lyrics for processing. Lyrics represent a potentially very rich source of information that is

[32] musicbrainz.org

currently underexploited in MIR, and it might be useful to develop a new ACE XML format specifically designed for storing lyrics and making it easy to extract various kinds of features from them.

It would also be helpful to develop a standardized file format for specifying queries that could be applied to data stored in ACE XML files. There are many general possible sources of inspiration, such as SQL, Z39.50[33] and FRBR.[34]

7.14 Sample file appendix

This section includes reproductions of complete ACE XML file samples for each of the ACE XML 1.1 and ACE XML 2.0 file formats. The *comments* clause of each of these files contains a description of how to use the file format in general. The first four of these sample files is set up to express information corresponding to the Weka ARFF file described in **Figure 7.5**, for the purpose of comparison between ACE XML 1.1 and Weka ARFF. The ACE XML files after these first four are intended to illustrate proposed changes to the ACE XML formats beyond the four basic stable ACE XML 1.1 formats.

```
<?xml version="1.0"?>
<!DOCTYPE feature_vector_file [
   <!ELEMENT feature_vector_file (comments, data_set+)>
   <!ELEMENT comments (#PCDATA)>
   <!ELEMENT data_set (data_set_id, section*, feature*)>
   <!ELEMENT data_set_id (#PCDATA)>
   <!ELEMENT section (feature+)>
   <!ATTLIST section start CDATA "" stop CDATA "">
   <!ELEMENT feature (name, v+)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT v (#PCDATA)>
]>

<feature_vector_file>

   <comments>This is an example of an XML file that stores extracted features
   that are to be used for training, testing or classification. The data_set_id
   element identifies the unique identifier of an instance that a particular set
   of features correspond to, in this case a file path. The section element
   delineates different sections of a given data set that are to be classified
   independently or semi-independently, such as analysis windows. The start and
   stop tags indicate when a section begins and ends, typically in terms of time
   stamps. The feature element delineates the different (potentially multi-
   dimensional) features extracted from an instance or section of an instance.
   The name element identifies the unique name of a particular feature. The v
   element indicates the value for a particular dimension of a
   feature.</comments>
```

[33] www.loc.gov/z3950/agency/
[34] www.ifla.org/VII/s13/wgfrbr/index.htm

```
<data_set>
   <data_set_id>C:\Recordings\Handel_4.wav</data_set_id>
   <section start="0" stop="99">
      <feature>
         <name>Spectral Centroid</name>
         <v>0.0</v>
      </feature>
   </section>
   <section start="100" stop="199">
      <feature>
         <name>Spectral Centroid</name>
         <v>440.0</v>
      </feature>
   </section>
   <section start="200" stop="283">
      <feature>
         <name>Spectral Centroid</name>
         <v>526.0</v>
      </feature>
   </section>
   <feature>
      <name>Duration</name>
      <v>250.0</v>
   </feature>
</data_set>

<data_set>
   <data_set_id>C:\Recordings\Handel_5.wav</data_set_id>
   <section start="0" stop="99">
      <feature>
         <name>Spectral Centroid</name>
         <v>0.0</v>
      </feature>
   </section>
   <section start="100" stop="199">
      <feature>
         <name>Spectral Centroid</name>
         <v>220.0</v>
      </feature>
   </section>
   <section start="200" stop="299">
      <feature>
         <name>Spectral Centroid</name>
         <v>115.0</v>
      </feature>
   </section>
   <section start="300" stop="342">
      <feature>
         <name>Spectral Centroid</name>
         <v>115.0</v>
      </feature>
   </section>
   <feature>
      <name>Duration</name>
      <v>372.5</v>
   </feature>
</data_set>

<data_set>
   <data_set_id>C:\Recordings\UnknownFile.wav</data_set_id>
   <section start="50" stop="99">
      <feature>
```

```
            <name>Spectral Centroid</name>
            <v>854.6</v>
         </feature>
      </section>
      <feature>
         <name>Duration</name>
         <v>960.3</v>
      </feature>
   </data_set>

</feature_vector_file>
```

Code Sample 7.1: A complete sample ACE XML 1.1 Feature Value file. This file starts with the Feature Value DTD, includes explanatory comments and specifies features for three windowed audio files. This file encodes the same feature values as those expressed in the Weka ARFF file shown in **Figure 7.5**. Some of the most essential differences between the two files are: the Feature Value file does not include class labels (which are instead found in Instance Label file shown in **Code Sample 7.3**), analysis windows are linked in the ACE XML file to the recording that they are extracted from and are each associated with time stamps, and the Duration feature is only specified in the ACE XML file for recordings as a whole, while the Spectral Centroid feature is only expressed in analysis windows.

```
<?xml version="1.0"?>
<!DOCTYPE feature_key_file [
   <!ELEMENT feature_key_file (comments, feature+)>
   <!ELEMENT comments (#PCDATA)>
   <!ELEMENT feature (name, description?, is_sequential, parallel_dimensions)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT description (#PCDATA)>
   <!ELEMENT is_sequential (#PCDATA)>
   <!ELEMENT parallel_dimensions (#PCDATA)>
]>

<feature_key_file>

   <comments>This is an example of an XML file that stores abstract data about
   features. The name element specifies the unique name of a feature. The
   optional description element can be used to include arbitrary descriptions or
   other metadata about features. The is_sequential element specifies whether or
   not the feature can be extracted from sub-sections of an instance. A value of
   true means that it can, and a value of false means that the feature can only
   be extracted from instances as a whole, not from their sub-sections. The
   parallel_dimensions element specifies the dimensionality of extracted vectors
   of the feature. This value will be 1 unless the feature is a multi-
   dimensional feature.</comments>

   <feature>
      <name>Spectral Centroid</name>
      <description>The spectral centre of mass of a signal window.</description>
      <is_sequential>true</is_sequential>
      <parallel_dimensions>1</parallel_dimensions>
   </feature>
```

```
   <feature>
      <name>Duration</name>
      <description>The duration in ms of a recording.</description>
      <is_sequential>false</is_sequential>
      <parallel_dimensions>1</parallel_dimensions>
   </feature>

</feature_key_file>
```

Code Sample 7.2: A complete sample ACE XML 1.1 Feature Description file. This file starts with the Feature Description DTD, includes explanatory comments and specifies the characteristics of two features, one of which is extracted from analysis windows of records, and one of which is extracted from recordings as a whole. This file encodes details about the features found in the Feature Value file shown in **Code Sample 7.1**.

```
<?xml version="1.0"?>
<!DOCTYPE classifications_file [
   <!ELEMENT classifications_file (comments, data_set+)>
   <!ELEMENT comments (#PCDATA)>
   <!ELEMENT data_set (data_set_id, misc_info*, role?, classification)>
   <!ELEMENT data_set_id (#PCDATA)>
   <!ELEMENT misc_info (#PCDATA)>
   <!ATTLIST misc_info info_type CDATA "">
   <!ELEMENT role (#PCDATA)>
   <!ELEMENT classification (section*, class*)>
   <!ELEMENT section (start, stop, class+)>
   <!ELEMENT class (#PCDATA)>
   <!ELEMENT start (#PCDATA)>
   <!ELEMENT stop (#PCDATA)>
]>

<classifications_file>

   <comments>This is an example of an XML file that stores class labels for
   instances and/or their sub-sections. The data_set element is used to
   delineate the start of each new overall instance, such as a musical
   recording. The data_set_id element is used to specify a unique identifier
   for each instance, such as a file path or a URI. The optional misc_info
   element can be used to provide as much metadata as desired to accompany each
   instance, and allows the specification of an info_type attribute to identify
   the type of each piece of metadata. The optional role element can be used to
   specify whether a file is to be used for training or testing in a particular
   evaluation run. The classification element is used to indicate the section of
   each data_set clause devoted to specifying actual class labels. The section
   element is used to delineate the sub-sections of instances, and the start and
   stop elements specify the potentially overlapping portions of the instance
   that each sub-section corresponds to. The class element is used to specify
   class labels for either instances as a whole or individual sub-sections of
   instances. More than one class label may be specified per instance or per
   sub-section.</comments>

   <data_set>
      <data_set_id>C:\Recordings\Handel_4.wav</data_set_id>
      <misc_info info_type="Composer">Handel</misc_info>
      <role>training</role>
      <classification>
         <section>
            <start>0</start>
            <stop>89</stop>
```

```
            <class>Silence</class>
         </section>
         <section>
            <start>90</start>
            <stop>194</stop>
            <class>Music</class>
         </section>
         <section>
            <start>195</start>
            <stop>299</stop>
            <class>Applause</class>
         </section>
      </classification>
   </data_set>

   <data_set>
      <data_set_id>C:\Recordings\Handel_5.wav</data_set_id>
      <role>training</role>
      <classification>
         <section>
            <start>0</start>
            <stop>88</stop>
            <class>Applause</class>
         </section>
         <section>
            <start>89</start>
               <stop>110</stop>
               <class>Speech</class>
         </section>
         <section>
            <start>111</start>
            <stop>157</stop>
            <class>Silence</class>
         </section>
         <section>
            <start>158</start>
            <stop>280</stop>
            <class>Music</class>
         </section>
         <section>
            <start>281</start>
            <stop>322</stop>
            <class>Applause</class>
         </section>
      </classification>
   </data_set>

   <data_set>
      <data_set_id>C:\Recordings\UnknownFile.wav</data_set_id>
      <misc_info info_type="Note">Not manually classified yet</misc_info>
      <classification></classification>
   </data_set>

</classifications_file>
```

Code Sample 7.3: A complete sample ACE XML 1.1 Instance Label file. This file starts with the Instance Label DTD, includes explanatory comments and specifies model class labels for sections of three recordings. No model classes are specified for the recordings as a whole, but this could certainly have been done within the Instance Label specification if desired. This file encodes the same class labels as those expressed in

the Weka ARFF file shown in **Figure 7.5**. Some of the most essential differences between this file and the ARFF file are: the Instance Label file does not include feature values (which are instead found in Feature Value file shown in **Code Sample 7.1**), class labels are explicitly associated with sub-sections that have precise time stamps rather than with overall recordings but there is no way to make this distinction in an ARFF file, identifying metadata is provided for each recording but there is no way to explicitly identify instances in ARFF files other than using comments, and the role of each of the instances for use as training or testing can be explicitly specified in the Instance Label file.

```
<?xml version="1.0"?>
<!DOCTYPE taxonomy_file [
   <!ELEMENT taxonomy_file (comments, parent_class+)>
   <!ELEMENT comments (#PCDATA)>
   <!ELEMENT parent_class (class_name, sub_class*)>
   <!ELEMENT class_name (#PCDATA)>
   <!ELEMENT sub_class (class_name, sub_class*)>
]>

<taxonomy_file>

   <comments>This is an example of an XML file that stores a hierarchical
   taxonomy of class labels. The parent_class element is used to define class
   labels at the highest level of the taxonomical tree. The optional sub_class
   element, which can be used recursively, is used to specify hierarchically
   subordinate classes of parent classes or of other sub-classes. If one wishes
   to use a flat taxonomy, then the sub-class element can simply not be used at
   all. The class_name element specifies the name of each class
   label.</comments>

   <parent_class>
      <class_name>Sound</class_name>

         <sub_class>
            <class_name>Speech</class_name>
         </sub_class>

         <sub_class>
            <class_name>Applause</class_name>
         </sub_class>

         <sub_class>
            <class_name>Music</class_name>
         </sub_class>

   </parent_class>

   <parent_class>
      <class_name>Silence</class_name>
   </parent_class>

</taxonomy_file>
```

Code Sample 7.4: A complete sample ACE XML 1.1 Class Ontology file. This file starts with the Class Ontology DTD, includes explanatory comments and specifies the root level classes (*Sound* and *Silence*) as well as three subordinate classes for the former (*Speech, Applause* and *Music*). This results in a total of four candidate leaf classes (*Silence, Speech, Applause* and *Music*). This file encodes details of the same class labels that are used in the Instance Label file shown in **Code Sample 7.3** as well as in the Weka ARFF file shown in **Figure 7.5**. However, the ARFF format provides no way to structure the candidate classes in any way, as Class Ontology files do.

```
<?xml version="1.0"?>
<!DOCTYPE ace_project_file [
   <!ELEMENT ace_project_file (comments, taxonomy_path,
                               feature_definitions_path, feature_vectors_path,
                               model_classifications_path, gui_preferences_path,
                               classifier_settings_path,
                               trained_classifiers_path, weka_arff_path)>
   <!ELEMENT comments (#PCDATA)>
   <!ELEMENT taxonomy_path (#PCDATA)>
   <!ELEMENT feature_definitions_path (path*)>
   <!ELEMENT feature_vectors_path (path*)>
   <!ELEMENT model_classifications_path (path*)>
   <!ELEMENT gui_preferences_path (#PCDATA)>
   <!ELEMENT classifier_settings_path (#PCDATA)>
   <!ELEMENT trained_classifiers_path (#PCDATA)>
   <!ELEMENT weka_arff_path (#PCDATA)>
   <!ELEMENT path (#PCDATA)>
]>

<ace_project_file>

   <comments>This is an example of an XML file that stores a list of ACE XML
   files that have been associated with each other for some form of processing
   together. The taxonomy_path element stores a reference (e.g. a file path or
   URI) to an ACE XML Class Ontology file. The feature_definitions_path element
   stores references to potentially multiple ACE XML Feature Description files.
   The feature_vectors_path element stores references to potentially multiple
   ACE XML Feature Value files. The model_classifications_path element stores
   references to potentially multiple ACE XML Instance Label files. The
   gui_preferences_path and classifier_settings_path elements store references
   to, respectively, ACE GUI preference and ACE machine learning settings in as
   of yet unspecified formats. The trained_classifiers_path element refers to a
   trained model stored as a Weka serialized object. Finally, the weka_arff_path
   element refers to a file in Weka ARFF format that may be combined with or
   used instead of ACE XML files.</comments>

   <taxonomy_path>/jMIR/TestFiles/ClassOntology.xml</taxonomy_path>
   <feature_definitions_path>
      <path>/jMIR/TestFiles/FeatureDescription.xml</path>
   </feature_definitions_path>
   <feature_vectors_path>
      <path>/jMIR/TestFiles/FeatureVector.xml</path>
   </feature_vectors_path>
   <model_classifications_path>
      <path>/jMIR/TestFiles/ModelInstanceLabels.xml</path>
      <path>/jMIR/TestFiles/PredictedInstanceLabels.xml</path>
   </model_classifications_path>
```

```
   <gui_preferences_path></gui_preferences_path>
   <classifier_settings_path></classifier_settings_path>
   <trained_classifiers_path></trained_classifiers_path>
   <weka_arff_path></weka_arff_path>

</ace_project_file>
```

Code Sample 7.5: A complete sample ACE XML 1.1 Project file. This file specifies a Feature Value file, a Class Ontology file and a Feature Description file that are to be used together. Two Instance Label files are also specified, one providing model class labels and the other for noting predicted class labels based on the output of a pattern recognition system. The other potential fields are left blank, as they are not applicable to this particular project.

```
<?xml version="1.0"?>
<!DOCTYPE ace_xml_project_file_2_0 [
   <!ELEMENT ace_xml_project_file_2_0 (comments?, feature_value_id,
                                       instance_label_id, class_ontology_id,
                                       feature_description_id, weka_arff_id?,
                                       trained_model_id?, uri?)>
   <!ELEMENT comments (#PCDATA)>
   <!ELEMENT feature_value_id (path*)>
   <!ELEMENT instance_label_id (path*)>
   <!ELEMENT class_ontology_id (#PCDATA)>
   <!ELEMENT feature_description_id (path*)>
   <!ELEMENT weka_arff_id (#PCDATA)>
   <!ELEMENT trained_model_id (#PCDATA)>
   <!ELEMENT uri (path*)>
   <!ATTLIST uri predicate CDATA #IMPLIED>
   <!ELEMENT path (#PCDATA)>
]>

<ace_xml_project_file_2_0>

   <comments>
         This is an example of an ACE XML 2.0 Project file. Files of this type
      are used to store a list of ACE XML files and other resources that have
      been associated with one another for some form of processing together.
         The feature_value_id element is used to store references (e.g., file
      paths or URIs) to zero to many ACE XML Feature Value files. The
      instance_label_id element is used to store references to zero to many ACE
      XML Instance Label files. The class_ontology_id element is used to a store
      reference to zero or one ACE XML Class Ontology file. The
      feature_description_id element is used to store references to zero to many
      ACE XML Feature Description files. The optional weka_arff_id element can
      be used to refer to a file in Weka ARFF format that may be combined with
      or used instead of ACE XML files. The optional trained_classifiers_path
      element can be used to refer to a trained classification model.
         Finally, zero to many uri clauses may be used to refer to external
      resources. The optional predicate attribute may be used with uri tags to
      indicate the kind of relationship between the subject containing the uri
      clause and the object that it refers to.
   </comments>

   <feature_value_id>
      <path>./2_0_FeatureValue.xml</path>
   </feature_value_id>
   <instance_label_id>
```

```
      <path>./2_0_InstanceLabel.xml</path>
   </instance_label_id>
   <class_ontology_id>./ClassOntology.xml</class_ontology_id>
   <feature_description_id>
      <path>./2_0_FeatureDescription.xml</path>
   </feature_description_id>
   <uri>
      <path>http://jmir.sourceforge.net/</path>
      <path>http://sourceforge.net/projects/jmir</path>
   </uri>

</ace_xml_project_file_2_0>
```

Code Sample 7.6: A complete sample ACE XML 2.0 Project file. This file expresses links to the ACE XML 2.0 sample files shown in **Code Samples 7.7** to **7.10** as well as to two jMIR SourceForge web sites.

```
<?xml version="1.0"?>
<!DOCTYPE ace_xml_feature_value_file_2_0 [
   <!ELEMENT ace_xml_feature_value_file_2_0 (comments?, related_resources?,
                                             instance+)>
   <!ELEMENT comments (#PCDATA)>
   <!ELEMENT related_resources (feature_value_file*, feature_description_file*,
                                instance_label_file*, class_ontology_file*,
                                project_file*, uri*)>
   <!ELEMENT feature_value_file (#PCDATA)>
   <!ELEMENT feature_description_file (#PCDATA)>
   <!ELEMENT instance_label_file (#PCDATA)>
   <!ELEMENT class_ontology_file (#PCDATA)>
   <!ELEMENT project_file (#PCDATA)>
   <!ELEMENT uri (#PCDATA)>
   <!ATTLIST uri predicate CDATA #IMPLIED>
   <!ELEMENT instance (instance_id, uri*, extractor*, coord_units?,
                       s*, precise_coord*, f*)>
   <!ELEMENT instance_id (#PCDATA)>
   <!ELEMENT extractor (#PCDATA)>
   <!ATTLIST extractor fname CDATA #REQUIRED>
   <!ELEMENT coord_units (#PCDATA)>
   <!ELEMENT s (uri*, f+)>
   <!ATTLIST s b CDATA #REQUIRED e CDATA #REQUIRED>
   <!ELEMENT precise_coord (uri*, f+)>
   <!ATTLIST precise_coord coord CDATA #REQUIRED>
   <!ELEMENT f (fid, uri*, (v+ | vd+ | vs+ | vj))>
   <!ATTLIST f type (int | double | float | complex | string) #IMPLIED>
   <!ELEMENT fid (#PCDATA)>
   <!ELEMENT v (#PCDATA)>
   <!ELEMENT vd (#PCDATA)>
   <!ATTLIST vd d0 CDATA #REQUIRED d1 CDATA #IMPLIED d2 CDATA #IMPLIED
                d3 CDATA #IMPLIED d4 CDATA #IMPLIED d5 CDATA #IMPLIED
                d6 CDATA #IMPLIED d7 CDATA #IMPLIED d8 CDATA #IMPLIED
                d9 CDATA #IMPLIED>
   <!ELEMENT vs (d+, v)>
   <!ELEMENT d (#PCDATA)>
   <!ELEMENT vj (#PCDATA)>
]>

<ace_xml_feature_value_file_2_0>

   <comments>
```

```
   This is an example of an ACE XML 2.0 Feature Value file. Files of this
   type are used to store feature values that have been extracted from
   instances. The optional comments and related_resources elements may be
   used to, respectively, informally note information about the file and
   provide links to related ACE XML files and other resources. In the latter
   case, it is generally preferable to use ACE XML Project files directly to
   associate files with one another instead.
      Each instance is stored in an instance clause and is uniquely
   identified using the instance_id element. This instance_id may be used to
   link each instance and its parts to appropriate class labels for the
   instance in an associated ACE XML Instance Label file. The feature
   extraction software used to extract each feature for an instance may be
   specified with the extractor element, where the contents specifies the
   software name and the fname attribute specifies the name of the feature.
      Features may be expressed for the instance as a whole (by entering them
   directly in the instance clause), for potentially overlapping sub-sections
   of the instance (by entering them in s clauses with b and e attributes
   noting the beginning and end of each section, typically but not
   necessarily in milliseconds) or for single points in the instance (by
   entering them in precise_coord clauses). In the cases of sub-sections
   and of points specified within instances, it is good practice to specify
   the units (e.g., seconds, ms, pixels, etc.) corresponding to the location
   markers. The optional coord_units element can be used to do this for each
   instance.
      The value(s) for each feature are entered in an f clause. Each feature
   is uniquely identified using the fid element, which may be used to
   associate the feature with a feature description in an associated ACE XML
   Feature Description file.
      Each feature may consist of a single value, 1-D vectors or arrays with
   an arbitrary number of dimensions. There are a number of alternative ways
   of specifying feature values. Any type of feature value from a single
   value to an N-dimensional array can be entered as a JSON array (i.e.,
   expressed using potentially nested square brackets) within a vj clause.
   Single value features or 1-D feature vectors may alternatively be
   represented using one or more v clauses. In the case of 1-D, to 10-D
   arrays, the additional alternative of specifying specific array
   coordinates with the d0 to d9 attributes is available using vd clauses,
   an option that permits sparse arrays.
      The final alternative for encoding arrays is offered by the vs element,
   which is more verbose but allows arrays with an arbitrary number of
   dimensions as well as sparse arrays. Each vs clause contains as many d
   elements as needed to specify array coordinates, followed by a single v
   element to specify the feature value. It is the responsibility of the ACE
   XML encoder to ensure that each vs clause has the same correct number of d
   elements and uses them in the same consistent order.
      It is possible to associate the values of particular features with
   particular data types using the type attribute, but in general it is
   preferable to omit this information in Feature Value files and specify it
   in ACE XML Feature Description files instead.
      The optional uri element may be used to associate individual instance,
   section, precise coordinate or feature clauses with external resources.
   The optional predicate attribute may be used with uri tags to indicate the
   kind of relationship between the subject containing the uri clause and the
   object that it refers to.
      The artificially generated data specified below includes three
   instances. The first shows how feature values can be expressed for
   overlapping sub-sections of an instance, unitary parts of an instance or
   for an instance as a whole. The second instance demonstrates how features
   of different dimensionalities can be expressed in different ways, namely
   single-value features, feature vectors, feature arrays and feature arrays
   of larger dimensionalities. The third and final instance demonstrates how
   features can be given specific data types.
</comments>
```

```
   <related_resources>

<feature_description_file>./2_0_FeatureDescription.xml</feature_description_file
>
      <instance_label_file>./2_0_InstanceLabel.xml</instance_label_file>
      <project_file>./2_0_Project.xml</project_file>
      <uri>http://jmir.sourceforge.net/</uri>
      <uri>http://sourceforge.net/projects/jmir</uri>
   </related_resources>

   <instance>
      <instance_id>Sub-Section Example</instance_id>
      <coord_units>ms</coord_units>

      <s b="0" e="99">
         <f>
            <fid>Feature Calculated Over Analysis Windows</fid>
            <v>0.5</v>
         </f>
      </s>
      <s b="80" e="120">
         <f>
            <fid>Feature Calculated Over Analysis Windows</fid>
            <v>0.3</v>
         </f>
      </s>
      <s b="100" e="150">
         <f>
            <fid>Feature Calculated Over Analysis Windows</fid>
            <v>3.0</v>
         </f>
      </s>

      <precise_coord coord="100">
         <f>
            <fid>Instantaneous Feature</fid>
            <v>180.0</v>
         </f>
      </precise_coord>
      <precise_coord coord="104">
         <f>
            <fid>Instantaneous Feature</fid>
            <v>350.0</v>
         </f>
      </precise_coord>

      <f>
         <fid>Feature Calculated For An Instance As A Whole</fid>
         <v>1000.8</v>
      </f>
   </instance>

   <instance>
      <instance_id>Feature Dimensionality Example</instance_id>
      <f>
         <fid>Single Value Feature</fid>
         <v>1</v>
      </f>
      <f>
         <fid>1-D Feature Vector</fid>
         <v>1</v>
         <v>2</v>
```

```
            <v>3</v>
         </f>
         <f>
            <fid>Second 1-D Feature Vector</fid>
            <vd d0="0">1</vd>
            <vd d0="1">2</vd>
         </f>
         <f>
            <fid>2-D Table Feature</fid>
            <vd d0="0" d1="0">1</vd>
            <vd d0="0" d1="1">2</vd>
            <vd d0="0" d1="2">3</vd>
            <vd d0="1" d1="0">11</vd>
            <vd d0="1" d1="1">22</vd>
            <vd d0="1" d1="2">33</vd>
         </f>
         <f>
            <fid>Second 2-D Table Feature</fid>
            <vs>
               <d>0</d><d>0</d>
               <v>1</v>
            </vs>
            <vs>
               <d>0</d><d>1</d>
               <v>2</v>
            </vs>
            <vs>
               <d>0</d><d>2</d>
               <v>3</v>
            </vs>
            <vs>
               <d>1</d><d>0</d>
               <v>11</v>
            </vs>
            <vs>
               <d>1</d><d>1</d>
               <v>22</v>
            </vs>
            <vs>
               <d>1</d><d>2</d>
               <v>33</v>
            </vs>
         </f>
         <f>
            <fid>3-D Array Feature</fid>
<vj>[[[[1],[2],[3],[4]],[[11],[22],[33],[44]],[[111],[222],[333],[444]]],[[[4],[
5],[6],[7]],[[44],[55],[66],[77]],[[444],[555],[666],[777]]]]</vj>
         </f>
      </instance>

      <instance>
         <instance_id>Typed Data Feature Example</instance_id>
         <f>
            <fid>Untyped Feature</fid>
            <v>1</v>
         </f>
         <f type="int">
            <fid>Integer Feature</fid>
            <v>1</v>
         </f>
         <f type="double">
            <fid>Double Feature</fid>
            <v>1.0</v>
```

```
      </f>
      <f type="string">
         <fid>String Feature</fid>
         <v>one</v>
      </f>
   </instance>

</ace_xml_feature_value_file_2_0>
```

Code Sample 7.7: A complete sample ACE XML 2.0 Feature Value file. An explanation of the file is provided in its *comments* clause. This file expresses artificially generated feature values extracted from the same instances referred to by the Instance Label file shown in **Code Sample 7.9**. The feature types correspond to those described by the Feature Description file shown in **Code Sample 7.8**.

```
<?xml version="1.0"?>
<!DOCTYPE ace_xml_feature_description_file_2_0 [
   <!ELEMENT ace_xml_feature_description_file_2_0 (comments?,
related_resources?,
                                                   global_parameter*, feature+)>
   <!ELEMENT comments (#PCDATA)>
   <!ELEMENT related_resources (feature_value_file*, feature_description_file*,
                                instance_label_file*, class_ontology_file*,
                                project_file*, uri*)>
   <!ELEMENT feature_value_file (#PCDATA)>
   <!ELEMENT feature_description_file (#PCDATA)>
   <!ELEMENT instance_label_file (#PCDATA)>
   <!ELEMENT class_ontology_file (#PCDATA)>
   <!ELEMENT project_file (#PCDATA)>
   <!ELEMENT uri (#PCDATA)>
   <!ATTLIST uri predicate CDATA #IMPLIED>
   <!ELEMENT feature (fid, description?, related_feature*, uri*, scope,
                      dimensionality?, data_type?, parameter*)>
   <!ELEMENT fid (#PCDATA)>
   <!ELEMENT description (#PCDATA)>
   <!ELEMENT related_feature (fid, relation_id?, uri*, explanation?)>
   <!ELEMENT relation_id (#PCDATA)>
   <!ELEMENT explanation (#PCDATA)>
   <!ELEMENT scope (#PCDATA)>
   <!ATTLIST scope overall (true|false) #REQUIRED
                   sub_section (true|false) #REQUIRED
                  precise_coord (true|false) #REQUIRED>
   <!ELEMENT dimensionality (uri*, size*)>
   <!ATTLIST dimensionality orthogonal_dimensions CDATA #REQUIRED>
   <!ELEMENT size (#PCDATA)>
   <!ELEMENT data_type (#PCDATA)>
   <!ATTLIST data_type type (int | double | float | complex | string) #REQUIRED>
   <!ELEMENT global_parameter (parameter_id, uri*, description?, value?)>
   <!ELEMENT parameter (parameter_id, uri*, description?, value?)>
   <!ELEMENT parameter_id (#PCDATA)>
   <!ELEMENT value (#PCDATA)>
]>

<ace_xml_feature_description_file_2_0>

   <comments>
         This is an example of an ACE XML 2.0 Feature Description file. Files of
      this type are used to store overall information about features in an
      abstract sense, not actual feature values.
```

The optional comments and related_resources elements may be used to, respectively, informally note information about the file and provide links to related ACE XML files and other resources. In the latter case, it is generally preferable to use ACE XML Project files directly to associate files with one another instead.

Each feature is stored in a feature clause and is uniquely identified using the fid element. This fid may be used to form associations with ACE XML Feature Value files that contain values extracted for these features in relation to particular instances. The description element may be used to provide qualitative information about each feature.

Features may each be appropriate to extract for an instance as a whole, for sub-sections of an instance and/or for precise points in an instance (e.g., a moment in time). This information can be specified using the scope element.

Although this information may be omitted if it varies, the orthogonal_dimensions attribute of the dimensionality element can be used to specify the number of different dimensions of the coordinate system for a feature (e.g., one for a feature vector, two for a table structure, etc.) and the size element can be used to indicate the size of each of these dimensions (e.g., number or rows and number of columns in a table structure).

The option is provided to associate the values of particular features with particular data types using the type attribute of the data_type element. This information may be specified in ACE XML Feature Value files as well, but Feature Description files take precedence in the case of contradictions.

Specific parameters may be associated with individual features, such as the roll-off point for the Spectral Roll-off feature, for example. A parameter clause should be used for each such parameter of a feature, the parameter_id element is used to identify the parameter uniquely, the description element can be used to describe the parameter in general, and the value element can be used to express numerical parameter values.

Global feature parameters may also be specified that apply to all features in the Feature Description file using the global_parameter element. This is useful for specifying overall pre-processing of audio files to be applied before all features are extracted, for example, such as downsampling or normalization. The global_parameter element works with the same mechanics as the parameter element.

An optional related_feature clause may be used to specify other features that are related to any given feature. This could be used, for example, to note that one feature is an alternative implementation of another. The fid element in the related feature clause should be used to specify the name of a feature specified in the fid clause of another feature. The relation_id element can be used to specify a specific externally defined type of relationship, and the explanation element can be used to provide a qualitative description of the relationship.

The optional uri element may be used to associate individual features, feature dimensionalities, feature parameters and feature relations with external resources. The optional predicate attribute may be used with uri tags to indicate the kind of relationship between the subject containing the uri clause and the object that it refers to.

The artificially generated data specified below includes 14 features. These features demonstrate a variety of ways that features can be configured. The features explained here are the same used in the artificial feature values expressed in the 2_0_FeatureValue.xml ACE XML Feature Value file.

```
</comments>

<related_resources>
   <feature_value_file>./2_0_FeatureValue.xml</feature_value_file>
   <project_file>./2_0_Project.xml</project_file>
   <uri>http://jmir.sourceforge.net/</uri>
   <uri>http://sourceforge.net/projects/jmir</uri>
```

```
</related_resources>

<feature>
   <fid>Feature Calculated Over Analysis Windows</fid>
   <description>An artificial feature.</description>
   <scope overall="false" sub_section="true" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="1">
      <size>1</size>
   </dimensionality>
   <data_type type="double"></data_type>
</feature>

<feature>
   <fid>Instantaneous Feature</fid>
   <description>An artificial feature.</description>
   <scope overall="false" sub_section="false" precise_coord="true"></scope>
   <dimensionality orthogonal_dimensions="1">
      <size>1</size>
   </dimensionality>
   <data_type type="double"></data_type>
</feature>

<feature>
   <fid>Feature Calculated For An Instance As A Whole.</fid>
   <description>An artificial feature.</description>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="1">
      <size>1</size>
   </dimensionality>
   <data_type type="double"></data_type>
</feature>

<feature>
   <fid>Single Value Feature</fid>
   <description>An artificial feature.</description>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="1">
      <size>1</size>
   </dimensionality>
   <data_type type="int"></data_type>
</feature>

<feature>
   <fid>1-D Feature Vector</fid>
   <description>An artificial feature.</description>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="1">
      <size>3</size>
   </dimensionality>
   <data_type type="int"></data_type>
</feature>

<feature>
   <fid>Second 1-D Feature Vector</fid>
   <description>An artificial feature.</description>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="1">
      <size>2</size>
   </dimensionality>
   <data_type type="int"></data_type>
</feature>

<feature>
```

```
   <fid>2-D Table Feature</fid>
   <description>An artificial feature.</description>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="2">
      <size>2</size>
      <size>3</size>
   </dimensionality>
   <data_type type="int"></data_type>
</feature>

<feature>
   <fid>Second 2-D Table Feature</fid>
   <description>An artificial feature.</description>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="2">
      <size>2</size>
      <size>3</size>
   </dimensionality>
   <data_type type="int"></data_type>
</feature>

<feature>
   <fid>3-D Array Feature</fid>
   <description>An artificial feature.</description>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="3">
      <size>2</size>
      <size>3</size>
      <size>4</size>
   </dimensionality>
   <data_type type="int"></data_type>
</feature>

<feature>
   <fid>Untyped Feature</fid>
   <description>An artificial feature.</description>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="1">
      <size>1</size>
   </dimensionality>
</feature>

<feature>
   <fid>Integer Feature</fid>
   <description>An artificial feature.</description>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="1">
      <size>1</size>
   </dimensionality>
   <data_type type="int"></data_type>
</feature>

<feature>
   <fid>Double Feature</fid>
   <description>An artificial feature.</description>
   <scope overall="true" sub_section="false" precise_coord="false"></scope>
   <dimensionality orthogonal_dimensions="1">
      <size>1</size>
   </dimensionality>
   <data_type type="double"></data_type>
</feature>

<feature>
```

```
      <fid>String Feature</fid>
      <description>An artificial feature.</description>
      <scope overall="true" sub_section="false" precise_coord="false"></scope>
      <dimensionality orthogonal_dimensions="1">
         <size>1</size>
      </dimensionality>
      <data_type type="string"></data_type>
   </feature>

   <feature>
      <fid>Feature With Parameters</fid>
      <description>An artificial feature.</description>
      <scope overall="true" sub_section="true" precise_coord="false"></scope>
      <parameter>
         <parameter_id>Some Parameter</parameter_id>
         <description>Some parameter for this feature.</description>
         <value>27</value>
      </parameter>
   </feature>

</ace_xml_feature_description_file_2_0>
```

Code Sample 7.8: A complete sample ACE XML 2.0 Feature Description file. An explanation of the file is provided in its *comments* clause. This file expresses abstract information about features. The first twelve of these features correspond to the feature values expressed in the Feature Value file shown in **Code Sample 7.7**.

```
<?xml version="1.0"?>
<!DOCTYPE ace_xml_instance_label_file_2_0 [
   <!ELEMENT ace_xml_instance_label_file_2_0 (comments?, related_resources?,
                                              instance+)>
   <!ELEMENT comments (#PCDATA)>
   <!ELEMENT related_resources (feature_value_file*, feature_description_file*,
                                instance_label_file*, class_ontology_file*,
                               project_file*, uri*)>
   <!ELEMENT feature_value_file (#PCDATA)>
   <!ELEMENT feature_description_file (#PCDATA)>
   <!ELEMENT instance_label_file (#PCDATA)>
   <!ELEMENT class_ontology_file (#PCDATA)>
   <!ELEMENT project_file (#PCDATA)>
   <!ELEMENT uri (#PCDATA)>
   <!ATTLIST uri predicate CDATA #IMPLIED>
   <!ELEMENT instance (instance_id, misc_info*, related_instance*, uri*,
                       coord_units?, section*, precise_coord*, class*)>
   <!ATTLIST instance role (training | testing | predicted) #IMPLIED>
   <!ELEMENT instance_id (#PCDATA)>
   <!ELEMENT related_instance (instance_id, relation_id?, uri*, explanation?)>
   <!ELEMENT relation_id (#PCDATA)>
   <!ELEMENT explanation (#PCDATA)>
   <!ELEMENT misc_info (info_id, uri*, info)>
   <!ELEMENT info_id (#PCDATA)>
   <!ELEMENT info (#PCDATA)>
   <!ELEMENT coord_units (#PCDATA)>
   <!ELEMENT section (uri*, class+)>
   <!ATTLIST section begin CDATA #REQUIRED end CDATA #REQUIRED>
   <!ELEMENT precise_coord (uri*, class+)>
   <!ATTLIST precise_coord coord CDATA #REQUIRED>
   <!ELEMENT class (class_id, uri*)>
   <!ATTLIST class weight CDATA "1">
   <!ATTLIST class source_comment CDATA #IMPLIED>
```

```
   <!ELEMENT class_id (#PCDATA)>
]>

<ace_xml_instance_label_file_2_0>

   <comments>
        This is an example of an ACE XML 2.0 Instance Label file. Files of this
      type are used to annotate class labels of instances. This can be used to
      specify ground-truth data used for training and evaluation, or it could
      specify predicted class labels output by a classification system, for
      example.
         The optional comments and related_resources elements may be used to,
      respectively, informally note information about the file and provide links
      to related ACE XML files and other resources. In the latter case, it is
      generally preferable to use ACE XML Project files directly to associate
      files with one another instead.
         Each instance is stored in an instance clause and is uniquely
      identified using the instance_id element. This instance_id may be used to
      link each instance and its parts to appropriate feature values for the
      instance in an associated ACE XML Feature Value file.
         The optional new related_instance element may be used within an
      instance clause to specify a relationship of any kind between the instance
      and any other instance, referred to using its instance_id. For example, it
      might be noted that one musical recording is a cover of another musical
      recording. The relation_id element can be used to specify a specific
      externally defined type of relationship, and the explanation element can
      be used to provide a qualitative description of the relationship.
         Similarly, the misc_info element may be used to specify any kind of
      metadata about an instance. The metadata field name is specified using
      the info_id element, and the metadata itself is put in an info clause.
         Each instance may be given a functional purpose using the role
      attribute, which can be used to specify whether the instance is to be used
      for training ground truth, for testing and evaluation or is labelled with
      class labels predicted by an external system.
         Class labels may be expressed for the instance as a whole (by entering
      them directly in the instance clause), for potentially overlapping sub-
      Sections of the instance (by entering them in section clauses with begin
      and end attributes noting the beginning and end of each section, typically
      but not necessarily in milliseconds) or for single points in the instance
      (by entering them in precise_coord clauses). In the cases of sub-sections
      and of ponts specified within instances, it is good practice to specify
      the units (e.g., seconds, ms, pixels, etc.) corresponding to the location
      markers. The optional coord_units element can be used to do this for each
      instance.
         Each class label is specified using the class_id element in a class
      clause. This class_id can be used to link to information about classes
      stored in an ACE XML Class Ontology file. The source_comment attribute
      can be used to specify how the labels were arrived at for a given
      instance (e.g. labeled by hand, classified automatically, etc.).
         Weighted class support may also be specified using the weight
      attribute. Weights specified here are assumed to be proportional, and
      should be normalized during parsing. Weights of 1 are assigned by default
      if no weight is explicitly specified.
         The optional uri element may be used to associate individual instance,
      section, precise coordinate or feature clauses with external resources.
      The optional predicate attribute may be used with uri tags to indicate the
      kind of relationship between the subject containing the uri clause and the
      object that it refers to.
         The artificial instance labels specified below correspond to the same
      instances for which artificial features are provided in the
      2_0_FeatureValue.xml ACE XML Feature Value file. Also, the artificial
      class labels used below match the candidate labels specified in the
      2_0_ClassOntology.xml ACE XML Class Ontology file. misc_info clauses are
```

```
   used below to explain each of the instances, and related_instance clauses
   are used to demonstrate the relationship between two of the instances.
</comments>

<related_resources>
   <feature_value_file>./2_0_FeatureValue.xml</feature_value_file>
   <class_ontology_file>./2_0_ClassOntology.xml</class_ontology_file>
   <project_file>./2_0_Project.xml</project_file>
   <uri>http://jmir.sourceforge.net/</uri>
   <uri>http://sourceforge.net/projects/jmir</uri>
</related_resources>

<instance>
   <instance_id>Sub-Section Example</instance_id>

   <misc_info>
      <info_id>Explanatory note</info_id>
      <info>
         The data below labels this instance as belonging to two different
         classes. The portion from 0 to 50 ms belongs only to Class Label 1
         and the portion from 50 ms to 150 ms belongs only to Class Label 2.
         Although this information could be fully expressed in only two of
         the section clauses below, additional redundant clauses are added
         below to demonstrate how precise coordinate and overall class
         annotations can be used. It also demonstrates how multi-class
         membership can be used, and how weighted class membership can be
         used. For example, the overall class labels for the instance include
         both of the classes because overall both classes are present, and
         Class Label 2 is given twice the weight because it covers twice the
         time interval of Class Label 1 in the instance overall. Again, this
         is for illustration only, and typically only the annotation shown
         for the Sub-Section Example Efficient instance would be used.
      </info>
   </misc_info>

   <related_instance>
      <instance_id>Sub-Section Example Efficient</instance_id>
      <relation_id>Copy</relation_id>
   </related_instance>

   <coord_units>ms</coord_units>

   <section begin="0" end="50">
      <class>
         <class_id>Class Label 1</class_id>
      </class>
   </section>
   <section begin="30" end="70">
      <class>
         <class_id>Class Label 1</class_id>
      </class>
      <class>
         <class_id>Class Label 2</class_id>
      </class>
   </section>
   <section begin="40" end="80">
      <class weight="1">
         <class_id>Class Label 1</class_id>
      </class>
      <class weight="3">
         <class_id>Class Label 2</class_id>
      </class>
   </section>
```

```
      <section begin="51" end="150">
         <class>
            <class_id>Class Label 2</class_id>
         </class>
      </section>

      <precise_coord coord="75">
         <class>
            <class_id>Class Label 2</class_id>
         </class>
      </precise_coord>

      <class weight="1">
         <class_id>Class Label 1</class_id>
      </class>
      <class weight="2">
         <class_id>Class Label 2</class_id>
      </class>
   </instance>

   <instance>
      <instance_id>Sub-Section Example Efficient</instance_id>

      <misc_info>
         <info_id>Explanatory note</info_id>
         <info>
            A more efficient representation of the same information shown in the
            Sub-Section Example.
         </info>
      </misc_info>

      <related_instance>
         <instance_id>Sub-Section Example</instance_id>
         <relation_id>Copy</relation_id>
      </related_instance>

      <coord_units>ms</coord_units>

      <section begin="0" end="50">
         <class>
            <class_id>Class Label 1</class_id>
         </class>
      </section>
      <section begin="51" end="150">
         <class>
            <class_id>Class Label 2</class_id>
         </class>
      </section>
   </instance>

   <instance>
      <instance_id>Feature Dimensionality Example</instance_id>

      <misc_info>
         <info_id>Explanatory note</info_id>
         <info>
             Unlike the Sub-Section Example instance, this instance is not
             broken into sections. It still has multiple class labels, however,
             but in this case they overlap, instead of having a discrete
             sequential relationship as in the Sub-Section Example. In this
             case, no weights are specified, so it is assumed that the instance
             belongs to all three classes equally. Of course, weighted class
             memberships could have been specified if appropriate.
```

```
         </info>
      </misc_info>

      <class>
         <class_id>Shared Class Label 1</class_id>
      </class>
      <class>
         <class_id>Shared Class Label 2</class_id>
      </class>
      <class>
         <class_id>Shared Class Label 3</class_id>
      </class>
   </instance>

   <instance>
      <instance_id>Typed Data Feature Example</instance_id>

      <misc_info>
         <info_id>Explanatory note</info_id>
         <info>
             This instance belongs to only one class.
         </info>
      </misc_info>

      <class>
         <class_id>Class Label 1</class_id>
      </class>
   </instance>

</ace_xml_instance_label_file_2_0>
```

Code Sample 7.9: A complete sample ACE XML 2.0 Instance Label file. An explanation of the file is provided in its *comments* clause. This file specifies class labels for particular instances. The instances shown here correspond to the instances for which feature values are expressed in the Feature Value file shown in **Code Sample 7.7**. Similarly, the candidate classes correspond to those shown in the ACE XML Class Ontology file outlined in **Code Sample 7.10**.

```
<?xml version="1.0"?>
<!DOCTYPE ace_xml_class_ontology_file_2_0 [
   <!ELEMENT ace_xml_class_ontology_file_2_0 (comments?, related_resources?,
                                              class+)>
   <!ATTLIST ace_xml_class_ontology_file_2_0 weights_relative (true|false)
                                                              #REQUIRED>
   <!ELEMENT comments (#PCDATA)>
   <!ELEMENT related_resources (feature_value_file*, feature_description_file*,
                                instance_label_file*, class_ontology_file*,
                                project_file*, uri*)>
   <!ELEMENT feature_value_file (#PCDATA)>
   <!ELEMENT feature_description_file (#PCDATA)>
   <!ELEMENT instance_label_file (#PCDATA)>
   <!ELEMENT class_ontology_file (#PCDATA)>
   <!ELEMENT project_file (#PCDATA)>
   <!ELEMENT uri (#PCDATA)>
   <!ATTLIST uri predicate CDATA #IMPLIED>
   <!ELEMENT class (class_id, misc_info*, uri*, related_class*, sub_class*)>
   <!ELEMENT class_id (#PCDATA)>
   <!ELEMENT misc_info (info_id, uri*, info)>
```

```
   <!ELEMENT info_id (#PCDATA)>
   <!ELEMENT info (#PCDATA)>
   <!ELEMENT related_class (class_id, relation_id?, uri*, explanation?)>
   <!ATTLIST related_class weight CDATA "1">
   <!ELEMENT relation_id (#PCDATA)>
   <!ELEMENT explanation (#PCDATA)>
   <!ELEMENT sub_class (class_id, relation_id?, uri*, explanation?)>
   <!ATTLIST sub_class weight CDATA "1">
]>

<ace_xml_class_ontology_file_2_0 weights_relative="true">

   <comments>
         This is an example of an ACE XML 2.0 Class Ontology file. Files of this
      type are used to store candidate class labels as well as information about
      the abstract relationships between classes.
         The optional comments and related_resources elements may be used to,
      respectively, informally note information about the file and provide links
      to related ACE XML files and other resources. In the latter case, it is
      generally preferable to use ACE XML Project files directly to associate
      files with one another instead.
         Each class is declared in a class element clause. Each is uniquely
      identified with the class_id element, which may be used to form
      connections with labelled instances stored in ACE XML Instance Label
      files.
         Metadata of any kind about each class can be stored using the misc_info
      element. The metadata field name is specified using the info_id element,
      and the metadata itself is put in an info clause.
         Two types of relationships may be specified between classes. The first
      type consists of general ontological relationships that can link any class
      with any other class unidirectionally. This is done using the
      related_class element. Bidirectional links can be formed by declaring the
      opposite class in each of the two class' related_class clauses.
         The second type of relationship is hierarchical. The sub_class element
      can be used to declare subordinate classes of any given class. Although
      subordinate classes can certainly have other classes subordinate to them
      in turn, there is no need to indicate parent classes in any way, as this
      information is implied.
         It is also possible to weight any related_class or sub_class connection
      using the weight attribute of both of these elements. Weights can be
      either absolute or relative (which means the weights for each class'
      connections are normalized on parsing), as defined by the global
      weights_relative attribute, which must be either true or false. If weight
      attributes are omitted then weights of 1 are assumed.
         Aside from the class_id of classes that are linked to, both the
      related_class and sub_class elements can also include qualitative
      explanations of each connection and a relation_id that corresponds to some
      externally defined keyword.
         The optional uri element may be used to associate individual instance,
      section, precise coordinate or feature clauses with external resources.
      The optional predicate attribute may be used with uri tags to indicate the
      kind of relationship between the subject containing the uri clause and the
      object that it refers to.
         The artificial class labels specified below correspond to the same
      labels that are used in the 2_0_InstanceLabel XML ACE XML Instance Label
      file. misc_info clauses are used below to illustrate how class
      relationships can be defined.
   </comments>

   <related_resources>
      <instance_label_file>./2_0_InstanceLabel.xml</instance_label_file>
      <project_file>./2_0_Project.xml</project_file>
      <uri>http://jmir.sourceforge.net/</uri>
```

```
   <uri>http://sourceforge.net/projects/jmir</uri>
</related_resources>

<class>
   <class_id>Exclusive Classes</class_id>

   <misc_info>
      <info_id>Explanation of taxonomical role</info_id>
      <info>
             This is a parent class of artificial classes that cannot apply
          to the same instance at the same time.
             A general ontological (non-hierarchical) connection is made to
          the Overlapping Classes class as well for the purpose of
          demonstration.
      </info>
   </misc_info>

   <related_class>
      <class_id>Overlapping Classes</class_id>
      <relation_id>Opposing parent nodes</relation_id>
   </related_class>

   <sub_class>
      <class_id>Class Label 1</class_id>
   </sub_class>
   <sub_class>
      <class_id>Class Label 2</class_id>
   </sub_class>
</class>

<class>
   <class_id>Overlapping Classes</class_id>

   <misc_info>
      <info_id>Explanation of taxonomical role</info_id>
      <info>
             This is a parent class of artificial classes that can apply to
          the same instance at the same time.
             A general ontological (non-hierarchical) connection is made to
          the Exclusive Classes class as well for the purpose of
          demonstration.
      </info>
   </misc_info>

   <related_class>
      <class_id>Exclusive Classes</class_id>
      <relation_id>Opposing parent nodes</relation_id>
   </related_class>

  <sub_class>
      <class_id>Shared Class Label 1</class_id>
   </sub_class>
   <sub_class>
      <class_id>Shared Class Label 2</class_id>
   </sub_class>
   <sub_class>
      <class_id>Shared Class Label 3</class_id>
   </sub_class>
</class>

<class>
   <class_id>Class Label 1</class_id>
```

```
      <misc_info>
         <info_id>Explanation of taxonomical role</info_id>
         <info>
                This class is a hierarchical descendant of the Exclusive Classes
            class. Note that hierarchical relationships are only specified in
            the parent class' statement, and no corresponding statement is
            necessary here.
         </info>
      </misc_info>
   </class>

   <class>
      <class_id>Class Label 2</class_id>

      <misc_info>
         <info_id>Explanation of ontological connections</info_id>
         <info>
                This class demonstrates how weighted ontological connections can
            be made to any other class, regardless of hierarchical sub-class
            structures. Note how the connection to Shared Class Label 1 is
            bidirectional, but the connection to Shared Class Label 2 is
            unidirectional. The connection to Shared Class Label 1 is also
            given a weight 3 times that of the connection to Shared Class
            Label 2, and this is a relative weighting because of the global
            ace_xml_class_ontology_file_2_0 weights_relative attribute
            setting above.
         </info>
      </misc_info>

      <related_class weight="3">
         <class_id>Shared Class Label 1</class_id>
         <relation_id>Ontological Connection</relation_id>
      </related_class>

      <related_class weight="1">
         <class_id>Shared Class Label 2</class_id>
         <relation_id>Ontological Connection</relation_id>
      </related_class>
   </class>

   <class>
      <class_id>Shared Class Label 1</class_id>

      <related_class>
         <class_id>Class Label 2</class_id>
         <relation_id>Ontological Connection</relation_id>
      </related_class>
   </class>

   <class>
      <class_id>Shared Class Label 2</class_id>
   </class>

   <class>
      <class_id>Shared Class Label 3</class_id>
   </class>

</ace_xml_class_ontology_file_2_0>
```

Code Sample 7.10: A complete sample ACE XML 2.0 Class Ontology file. An explanation of the file is provided in its *comments* clause. This file specifies candidate

class labels and relationships between them. The classes labels used here correspond to those used to label instances in the Instance Label file shown in **Code Sample 7.9**.

7.15 Chapter bibliography

Amatrain, X., P. Arumi, and M. Ramirez. 2002 CLAM: Yet another library for audio and music processing? *Proceedings of the ACM Conference on Object Oriented Programming, Systems, and Applications.* 22–3.

Burred J. J., C. E. Cella, G. Peeters, A. Röbel, and D. Schwarz. 2008. Using the SDIF Sound Description Interchange Format for audio features. *Proceedings of the International Conference on Music Information Retrieval.* 427–32.

Cannam, C., C. Landone, M. Sandler, and J. P. Bello. 2006. The Sonic Visualiser: A visualization platform for semantic descriptors from musical signals. *Proceedings of the International Conference on Music Information Retrieval.* 324–7.

Downie, J. S., K. West, A. Ehmann, and E. Vincent. 2005. The 2005 Music Information Retrieval Evaluation eXchange (MIREX 2005): Preliminary overview. *Proceedings of the International Conference on Music Information Retrieval.* 320–3.

Kitahara, T. 2008. A unified and extensible framework for developing music information processing systems. *Unpublished manuscript.*

McKay, C. 2004. Automatic genre classification of MIDI recordings. *M.A. Thesis*. McGill University, Canada.

McKay, C., R. Fiebrink, D. McEnnis, B. Li, and I. Fujinaga. 2005. ACE: A framework for optimizing music classification. *Proceedings of the International Conference on Music Information Retrieval.* 42–9.

Mierswa, I., M. Wurst, R. Klinkenberg, M. Scholzn, and T. Euler. 2006. YALE: Rapid prototyping for complex data mining tasks. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 935–40.

Raimond, Y. 2009. A distributed music information system. *Doctoral Dissertation*. Queen Mary, University of London, U.K.

Raimond, Y., S. Adbdallah, M. Sandler, and F. Giasson. 2007. The Music Ontology. *Proceedings of the International Conference on Music Information Retrieval.* 417–22.

Raimond, Y., and M. Sandler. 2008. A web of musical information. *Proceedings of the International Conference on Music Information Retrieval.* 263–28.

Schwarz, D., and M. Wright. 2000. Extensions and applications of the SDIF Sound Description Interchange Format. *Proceedings of the International Computer Music Conference*. 481–4.

Tzanetakis, G., and P. Cook. 2000. Marsyas: A framework for audio analysis. *Organized Sound* 4 (3): 169–75.

Tzanetakis, G., L. G. Martins, L. F. Teixeira, C. Castillo, R. Jones, and M. Lagrange. 2008. Interoperability and the Marsyas 0.2 runtime. *Proceedings of the International Computer Music Conference.*

Whitehead, P., E. Friedman-Hill, and E. Vander Veer. 2002. *Java and XML: Your visual blueprint for creating Java-enhanced Web programs.* Mississauga, Canada: Wiley Publishing Inc.

Witten, I. H., and E. Frank. 2005. *Data mining: Practical machine learning tools and techniques*. New York: Morgan Kaufman.

Wright, M., A. Chaudhary, A. Freed, S. Khoury, and D. Wessel. 1999. Audio applications of the Sound Description Interchange Format standard. *Proceedings of the Audio Engineering Society Convention.* 276–9.

CrestMuse Project. Retrieved December 30, 2008, from http://www.crestmuse.jp/index-e.html.

Dublin Core Metadata Initiative. Retrieved February 16, 2009, from http://dublincore.org.

JSON. Retrieved January 15, 2009, from http://json.org/.

Meandre. Retrieved January 7, 2009, from http://seasr.org/meandre/.

Music Ontology Specification. Retrieved December 30, 2008, from http://www.musicontology.com.

SDIF Sound Description Interchange Format. Retrieved December 30, 2008, from http://sdif.sourceforge.net.

Weka >> ARFF. Retrieved December 30, 2008, from http://weka.wiki.sourceforge.net/ARFF.