1 Preface

Teile vom Jstacs-Paper?

2 Starter: Data handling

In Jstacs, data is organized at three levels:

- Alphabets for defining single symbols, and AlphabetContainers for defining aggregate alphabets,
- Sequences for defining sequences of symbols over a given alphabet,
- DataSets for defining sets of sequences.

Sequences are implemented as an array of numerical values. In case of discrete sequences over some symbolic alphabet, the symbols are mapped to contiguous discrete values starting at 0, which can be mapped back to the original symbols using the alphabet. This mapping is also used for the `toString()` method, e.g., for printing a sequence. The actual data type, i.e. byte, short, or integer, used to represented the symbols is chosen internally depending on the size of the alphabet. Alphabets, Sequences, and DataSets are immutable for reasons of security and data consistency. That means, an instance of those classes cannot be modified once it has been created.

2.1 Alphabets

Since Jstacs is tailoured at sequence analysis in bioinformatics, the most prominent alphabet is the DNAAlphabet, which is a singleton instance that can be accessed by:

```
DNAAlphabet dna = DNAAlphabet.SINGLETON;
```

For general discrete alphabets, i.e., any kind of categorical data, you can use a DiscreteAlphabet. Such an alphabet can be constructed in case-sensitive and insensitive variants (first argument) using a list of symbols. In this example, we create a case-sensitive alphabet with symbols ”W”, ”S”, ”w”, and ”x”:

```
DiscreteAlphabet discrete = new DiscreteAlphabet( false, "W", "S", "w", "
    x" );
```

If you rather want to define an alphabet over contiguous discrete numerical values, you can do so by calling a constructor that takes the maximum and minimum value of the desired range, and defines the alphabet as all integer values between minimum and maximum (inclusive). For example, to create a discrete alphabet over the values from 3 to 10, you can call

```
DiscreteAlphabet numerical = new DiscreteAlphabet( 3, 10 );
```

Continuous alphabets are defined over all reals (minus infinity to infinity) by default (see first line in the following example). However, if you want to define the continuous alphabet over a specific interval, you can specify the maximum and the minimum value of that interval. In the example, we define a continuous alphabet spanning all reals between 0 and 100:

```
ContinuousAlphabet continuousInf = new ContinuousAlphabet();
ContinuousAlphabet continuousPos = new ContinuousAlphabet( 0.0, 100.0 );
```

For the DNA alphabet, each symbols has a complementary counterpart. Since in some cases, a similar complementarity can also be defined for symbols other than DNA-nucleotides (e.g., for RNA sequences containing U instead of T), Jstacs allows to define generic complementable alphabets. These allow for example the generation of reverse complementary sequences out of an existing sequence. Here, we define a binary alphabet of symbols "A" and "B", where "A" is the complement of "B" and vice versa.

```
GenericComplementableDiscreteAlphabet complementable = new
    GenericComplementableDiscreteAlphabet( true, new String[]{"A","B"},
    new int[]{1,0} );
```

The first parameter again defines if this alphabet is case-insensitive (which is the case), the second parameter defines the symbols of the alphabet, and the third parameter specifies the index of the complementary symbol. For instance, the symbol at position 1 ("B") is set as the complement of the symbol at position 0 ("A") by setting the 0-th value of the integer array to 1.

After the definition of single alphabets, we switch to the creation of aggregate alphabets. Almost everywhere in Jstacs, we use aggregate alphabets to maintain generalizability. Since the aggregate alphabet containing only a DNAAlphabetis always the same, a singleton for such an AlphabetContaineris pre-defined:

```
AlphabetContainer dnaContainer = DNAAlphabetContainer.SINGLETON;
```

We can explicitly define an AlphabetContainer using a simple continuous alphabet by calling:

```
AlphabetContainer contContainer = new AlphabetContainer( continuousInf );
```

Aggregate alphabets become interesting if we need different symbols at different positions of a sequence, or even a mixture of discrete and continuous values. For example, we might want to represent sequences that consist of a DNA-nucleotide at the first position, some other discrete symbol at the second position, and a real number stemming from some measurement at the third position. Using the DNAAlphabet, the discrete Alphabet, and the continuous Alphabet defined above, we can define such an aggregate alphabet by calling

```
AlphabetContainer mixedContainer = new AlphabetContainer(dna, discrete,
    continuousPos);
```

To save memory, we can also re-use the same alphabet at different position of the aggregate alphabet. If we want to use a DNAAlphabet at positions 0, 1, and 3, and a continuous alphabet at positions 2, 4, and 5, we can use a constructor that takes the alphabets as the first argument and the assignment to the positions as the second argument:

```
AlphabetContainer complex = new AlphabetContainer( new Alphabet[]{dna,
    continuousInf}, new int[]{0,0,1,0,1,1} );
```

The alphabets are assigned to specific positions by their index in the array of the first argument.

2.2 Sequences

Single sequences can be created from an AlphabetContainer and a string. However, in most cases, we load the data from some file, which will be explained in the next sub-section. For creating a DNA sequence, we use a DNAAlphabet like the one defined above and a string over the DNA alphabet:

```
Sequence dnaSeq = Sequence.create( dnaContainer, "ACGTACGTACGT" );
```

In a similar manner, we define a continuous sequence. In this case, a single value is represented by more than one letter in the string. Hence, we need to define a delimiter between the values as a third argument, which is a space in the example.

```
Sequence contSeq = Sequence.create( contContainer, "0.5␣1.32642␣99.5␣20.4
   ␣5␣7.7" , "␣" );
```

We can also create sequences over the mixed alphabet defined above. In the example, the single values are delimited by a “;”.

```
Sequence mixedSeq = Sequence.create( mixedContainer, "C;x;5.67" , ";" );
```

For very large amounts of data or very long sequences, even the representation of symbols by byte values can be too memory-consuming. Hence, Jstacs also offers a representation of DNA sequences in a sparse encoding as bits of long values. You can create such a SparseSequence from a DNAAlphabet and a string:

```
Sequence sparse = new SparseSequence( dnaContainer, "ACGTACGTACGT" );
```

However, the reduced memory footprint comes at the expense of a slightly increased runtime for accessing symbols of a SparseSequence. Hence, it is not the default representation in Jstacs.

After we learned how to create sequences, we now want to work with them. First of all, you can obtain the length of a sequence from its `getLength()` method:

```
int length = dnaSeq.getLength();
```

Since on the abstract level of Sequence we do not distinguish between discrete and continuous sequences (and we also may have mixed sequences), there are two alternative methods to obtain one element of a sequence regardless of its content. With the first method, we can obtain the discrete value at a certain position (2 in the example):

```
int value = dnaSeq.discreteVal( 2 );
```

If the Sequence contains a continuous value at this position, it is discretized by default by returning the distance to the minimum value of the continuous alphabet at this position casted to an integer. If the Sequence contains a discrete value, that value is just returned in the encoding according to the AlphabetContainer. In a similar manner, we can obtain the continuous value at a position (5 in the example)

```
double value2 = contSeq.continuousVal( 5 );
```

where discrete values are just casted to `double`s.

We can obtain a sub-sequence of a Sequence using the method `getSubSequence(int,int)`, where the first parameter is the start position within the sequence, counting from 0, and the second parameter is the length of the extracted sub-sequence. So the following line of code would extract a sub-sequence of length 3 starting at position 2 of the original sequence or, stated differently, we skip the first two elements, extract the following three elements, and again skip everything after position 4.

```
Sequence contSub = contSeq.getSubSequence( 2, 3 );
```

Since Sequences in Jstacs are immutable, this method returns a new instance of Sequence, which is assigned to a variable `contSub` in the example. Hence, in cases where you need the same sub-sequences frequently in your code, for example in a ZOOPS-model or other models using sliding windows on a Sequence, we recommend to precompute these sub-sequences and store them in some auxiliary data structure in order to invest runtime in computations rather than garbage collection. Internally, sub-sequences only hold a reference on the original sequences and the start position and length within that sequence to keep the memory overhead of sub-sequences at a low level.

For Sequences defined over a ComplementableDiscreteAlphabet like the DNAAlphabet, we can also obtain the (reverse) complement of a sequence. For example, to create the reverse complementary sequence of a complete sequence, we call

```
Sequence revComp = dnaSeq.reverseComplement();
```

For the complement of a sub-sequence of length 6 starting at position 3 of the original sequence, we use

```
Sequence subComp = dnaSeq.complement( 3, 6 );
```

For some analyses, for instance permutation tests or for estimating false-positive rates of predictions, it is useful to create permuted variants of an original sequence. To this end, Jstacs provides a class PermutedSequence that creates a randomly permuted variant using the constructor

```
PermutedSequence permuted = new PermutedSequence( dnaSeq );
```

or a user-defined permutation by an alternative constructor. In the randomized variant, the positions of the original sequence are permuted independently of each other, which means that higher order properties of the sequence like di-nucleotide content are not preserved. If you want to create sequences with similar higher-order properties, have a look at the `emitSample()` method of HomogeneousModel.

Often, we want to add additional annotations to a sequence, for instance the occurrences of some binding motif, start and end positions of introns, or just the species a sequence is stemming from. To this end, Jstacs provides a number of SequenceAnnotations that can be added to a Sequence (or read from a FastA-file as we will see later). For instance, we can add the annotation for binding site of a motif called "new motif" of length 5 starting at position 3 of the forward strand of sequence `dnaSeq` using the `annotate` method of that sequence:

```
Sequence annotatedDnaSeq = dnaSeq.annotate( true, new MotifAnnotation( "
    new␣motif", 3, 5, Strand.FORWARD ) );
```

Again, this method creates a new Sequence object due to Sequences being immutable. After we added several SequenceAnnotations to a Sequence, we can obtain all those annotations by calling

```
SequenceAnnotation[] allAnnotations = annotatedDnaSeq.getAnnotation();
```

For retrieving annotations of a specific type, we can use the method `getSequenceAnnotationByType`

```
MotifAnnotation motif = (MotifAnnotation) annotatedDnaSeq.
    getSequenceAnnotationByType( "Motif", 0 );
```

to, for instance, obtain the first (index 0) annotation of type "Motif".

2.3 DataSets

In most cases, we want to load Sequences from some FastA or plain text file instead of creating Sequences manually from strings. In Jstacs, collections of Sequences are represented by DataSets. The class DataSet (and DNADataSet) provide constructors that work on a file or the path to a file, and parse the contents of the file to a DataSet, i.e. a collection of Sequences.

The most simple case is to create a DNADataSet from a FastA file. To do so, we call the constructor of DNADataSetwith the (absolute or relative) path to the FastA file:

```
DNADataSet dnaSample = new DNADataSet( "myfile.fa" );
```

For other file formats and types of Sequences, DataSet provides another constructor that works on the AlphabetContainerfor the data in the file, a StringExtractor that handles the extraction of the strings representing single sequences and skipping comment lines, and a delimiter between the elements of a sequence. Hence, the StringExtractor, a SparseStringExtractor in the example, requires the specification of the path to the file and the symbol that indicates comment line. For example, if we want to create a sample of continuous sequences stored in a tab-separated plain text file "myfile.tab", we use the AlphabetContainer with a continuous Alphabet from above, a StringExtractor with a hash as the comment symbol, and a tab as the delimiter:

```
DataSet contSample = new DataSet( contContainer, new
    SparseStringExtractor( "myfile.tab", '#' ), "\t" );
```

The SparseStringExtractor is tailoured to files containing many sequences, and reads the file line by line, where each line is converted to a Sequence and discarded before the next line is parsed.

Since SparseSequences are not one of the default representations of sequence in Jstacs (see above), these are not created by the constructors of DataSet or DNADataSet. However, the class SparseSequence provides a method `getSample` that takes the same arguments as the constructor of DataSet, for example

```
DataSet sparseSample = SparseSequence.getDataSet( dnaContainer, new
    SparseStringExtractor( "myfile.fa", '>' ) );
```

for reading DNA sequences from a FastA file, and returns a DataSet containing SparseSequences.

After we successfully created a DataSet, we want to access and use the Sequences within this DataSet. We retrieve a Sequence of a DataSet using the method `getElementAt(int)`. For instance, we get the fifth Sequence of `dnaSample` by calling

```
Sequence fifth = dnaSample.getElementAt( 5 );
```

We can also request the number of Sequences in a DataSet by the method `getNumberOfElements()` and use this information, for instance, to iterate over all Sequences. In the example, we just print the retrieved Sequences to standard out

```
for(int i=0;i<dnaSample.getNumberOfElements();i++){
 System.out.println(dnaSample.getElementAt( i ));
}
```

where the Sequences are printed in their original alphabet since their `toString()` method is overridden accordingly.

As an alternative to the iteration by explicit calls to these methods, DataSet also implements the `Iterable` interface, which facilitates the Java variant of foreach-loops as in the following example:

```
for(Sequence seq : contSample){
 System.out.println(seq.getLength());
}
```

Here, we just print the length of each Sequence in `contSample` to standard out.

We can also apply some of the sequence-level operations to all Sequences of a DataSet, and obtain a new DataSet containing the modified sequences. For example, we get a DataSet containing the sub-sequences of length 10 starting at position 3 of each sequence by calling

```
DataSet infix = dnaSample.getInfixDataSet( 3, 10 );
```

a DataSet of all suffixes starting at position 7 from

```
DataSet suffix = dnaSample.getSuffixDataSet( 7 );
```

or a DataSet containing all reverse complementary Sequences using

```
DataSet allRevComplements = dnaSample.getReverseComplementaryDataSet();
```

For cross-validation experiments, hold-out samplings, or similar procedures, it is useful to partition a sample randomly. DataSets in Jstacs support two types of partitionings. The first is to partition a DataSet into equally sized parts. What is “equally sized” can either be determined by the number of sequences or by the number of symbols of all sequences in a sample. Both measures are supported by Jstacs.

The second partitioning method creates partitions of a user-defined fraction of the original sample. For example, we partition the DataSet `dnaSample` into five equally sized parts according to the number of sequences in that DataSet by calling

```
DataSet[] fiveParts = dnaSample.partition( 5, PartitionMethod.
    PARTITION_BY_NUMBER_OF_ELEMENTS );
```

and we partition the same sample into parts containing 10, 20, and 70 percent of the symbols of the original DataSetby calling

```
DataSet[] randParts = dnaSample.partition( PartitionMethod.
    PARTITION_BY_NUMBER_OF_SYMBOLS, 0.1, 0.2, 0.7 );
```

In both cases, the Sequences in the DataSet are partitioned as atomic elements. That means, a Sequence is not cut into several parts to obtain exactly equally sized parts, but the size of a part may slightly (depending on the number of sequences and lengths of those sequences) differ from the specified percentages.

To create a new DataSet that contains all sub-sequences of a user-defined length of the original Sequences, we can use another constructor of DataSet. The sub-sequences are extracted in the same manner as we would do by shifting a sliding window over each sequence, extracting the sub-sequence under this window, and build a new DataSet of the extracted sub-sequences. For instance, we obtain a DataSet with all sub-sequences of length 8 using

```
DataSet sliding = new DataSet( dnaSample, 8 );
```

In the previous sub-section, we learned how to add SequenceAnnotations to a Sequence. Often, we want to use the annotation that is already present in an input file, for example the comment line of a FastA file. We can do so by specifying a SequenceAnnotationParser in the constructor of the DataSet. The simplest type of SequenceAnnotationParser is the SimpleSequenceAnnotationParser, which just extracts the complete comment line preceding a sequence.

```
DNADataSet dnaWithComments = new DNADataSet( "myfile.fa", '>', new
    SimpleSequenceAnnotationParser() );
```

Although the specification of the parser is quite simple, the extraction of the comment line as a string is a bit lengthy. We first obtain the Sequence from the DataSet, get the annotation of that sequence, obtain the first comment, called "result" in the hierarchy of Jstacs (you see in section 3, why), and convert the corresponding result object to a string.

```
String comment = dnaWithComments.getElementAt( 0 ).getAnnotation()[0].
    getResultAt( 0 ).getValue().toString();
```

If your comment line is defined in a "key-value" format with some generic delimiter between entries, you can Jstacs let parse the entries to distinct annotations. For instance, if the comment line has some format `key1=value1; key2=value2;...`, we can parse that comment line using the SplitSequenceAnnotationParser. This parser only requires the specification of the delimiter between key and value ("=" in the example) and the delimiter between different entries (";" in the example). Like before, we instantiate a SplitSequenceAnnotationParser as the last argument of the DNADataSet constructor:

```
DNADataSet dnaWithParsedComments = new DNADataSet( "myfile.fa", '>', new
    SplitSequenceAnnotationParser("=",";") );
```

We can now access all parsed annotations by the `getAnnotation()` method

```
SequenceAnnotation[] allAnnotations2 = dnaWithParsedComments.getElementAt
    ( 0 ).getAnnotation();
```

or, for instance, the `getSequenceAnnotationByType` introduced in the previous section, where the type corresponds to the key in the comment line, and the identifier of the retrieved SequenceAnnotation is identical to the value for that key in the comment line.

Jstacs only supports FastA and plain text files directly. However, you can access other formats or even data bases like Genbank using an adaptor to BioJava.

For example, we can use BioJava to load two sequences from Genbank.

```
GenbankRichSequenceDB db = new GenbankRichSequenceDB();
HashSet<String> idSet = new HashSet<String>( 2 );
idSet.add( "NC_001284.2" );
idSet.add( "NC_000932.1" );
RichSequenceDB subDB = db.getRichSequences( idSet );
RichSequenceIterator dbIterator = subDB.getRichSequenceIterator();
```

As a result, we obtain a `RichSequenceIterator`, which implements the `SequenceIterator` interface of BioJava. We can use a `SequenceIterator` in an adaptor method to obtain a Jstacs DataSet including converted annotations:

```
DataSet fromBioJava = BioJavaAdapter.sequenceIteratorToDataSet(
    dbIterator, null );
```

The second argument of the method allows for filtering for specific annotations using a BioJava `FeatureFilter`.

Vice versa, we can convert a Jstacs DataSet to a BioJava `SequenceIterator` by an analogous adaptor method:

```
SequenceIterator backFromJstacs = BioJavaAdapter.
    dataSetToSequenceIterator( fromBioJava, true );
```

By means of these two methods, we can use all BioJava facilities for loading and storing data from and to diverse file formats and loading data from data bases in our Jstacs applications.

3 Intermediate course: XMLParser, Parameters, and Results

In the early days of Jstacs, we stored models, classifiers, and other Jstacs objects using the standard serialization of Java. However, this mechanism made it impossible to load objects of earlier versions of a class and the files where not human-readable. Hence, we started to create a facility for storing objects to XML representations. In the current version of Jstacs, this is accomplished by an interface Storable for objects that can be converted to and from their XML representation, and a class XMLParser that can handle such Storables, primitives, and arrays thereof. In the first sub-section, we give examples how to use the XMLParser.

Another problem we wanted to handle has been the documentation of (external) parameters of models, classifiers, or other classes. Although documentation exists in the Javadocs, these are inaccessible from the code. Hence, we created classes for the documentation of parameters and sets of parameters, namely the subclasses of Parameter and ParameterSet. A Parameter at least provides the name of and a comment on the parameter that is described. In sub-classes, other values are also available like, for instance, the set or a range of allowed values. Such a description of parameters allows for manifold generic convenience applications. Current examples are the ParameterSetTagger, which facilitates the documentation of command line arguments on basis of a ParameterSet, or the GalaxyAdaptor, which allows for an easy integration of Jstacs applications into the Galaxy webserver. We give examples for the use and creation of Parameters and ParameterSets in the second sub-section.

Finally, the same problem also occurrs for the results of computations. With a generic documentation, these results can be displayed together with some annotation in a way that is appropriate for the current application. In Jstacs, we use Results and ResultSets for this purpose, and we show how to use these in the third sub-section.

3.1 XMLParser

In the following examples, let `buffer` be some `StringBuffer`. All kinds of primitives or Storables are appended to an existing `StringBuffer` surrounded by the specified XML tags by the static method `appendObjectWithTags` of XMLParser. For example, the following two lines append an integer with the value 5 using the tag `integer`, and a `String` with the tag `foo`:

```
int integer = 5;
XMLParser.appendObjectWithTags( buffer, integer, "integer" );
String bar = "hello␣world";
XMLParser.appendObjectWithTags( buffer, bar, "foo" );
```

If we assume that `buffer` was an empty `StringBuffer` before appeding these two elements, the resulting XML text will be

```
<integer>5</integer>
<foo>hallo welt</foo>
```

In exactly the same manner, we can append XML representations of arrays of primitives, for example a two-dimensional array of `double` s

```
double[][] da = new double[4][6];
XMLParser.appendObjectWithTags( buffer, da, "da" );
```

or complete Jstacs models that implement the Storable interface

```
HomogeneousMM hMM = new HomogeneousMM( new HomMMParameterSet( new
    AlphabetContainer( DNAAlphabet.SINGLETON ), 4, "hmm(0)", (byte) 0 ) )
    ;
XMLParser.appendObjectWithTags( buffer, hMM, "hMM" );
```

or even arrays of Storables:

```
Storable[] storAr = ArrayHandler.createArrayOf( hMM, 5 );
XMLParser.appendObjectWithTags( buffer, storAr, "storAr" );
```

The interface Storable only defines two things: first, an implemening class must provide a public method `toXML()` that returns the XML representation of this class as a `StringBuffer`, and second, it must provide a constructor that takes a single `StringBuffer` as its argument and re-creates an object out of this representation. The only exception from this rule are singleton, i.e., classes that implement the Singleton interface.

Of course, you can use the `appendObjectWithTags` method of the XMLParser inside the `toXML` method. By this means, it is possible to break down the conversion of complex models into smaller pieces if the building-blocks of a model are also Storables.

In analogy to storing objects, the XMLParser also provides facilities for loading primitives and Storables from their XML representation. These can also be used in the constructor according to the Storable interface. For example, we can load the value of the integer, we stored a few lines ago by calling

```
integer = (Integer) XMLParser.extractObjectForTags( buffer, "integer" );
```

where the second argument of `extractObjectForTags` is the tag surrounding the value and, of course, must be identical to the tag we specified when storing the value. Since `extractObjectForTags` is a generic method, we must explicitly cast the returned value to an `Integer`. As an alternative, we can also specify the class of the return type as a third argument like in the following example

```
da = XMLParser.extractObjectForTags( buffer, "da", double[][].class );
```

Here, we load the two-dimensional array of `doubles` that we stored a few lines ago. In perfect analogy, we can also load a single instance of a class implementing Storable

```
hMM = XMLParser.extractObjectForTags( buffer, "hMM", HomogeneousMM.class
    );
```

where in this case we again specify the class of the return type in the third argument, or arrays of Storable

```
storAr = (Storable[]) XMLParser.extractObjectForTags( buffer, "storAr" );
```

Of course, we can also specify the concrete sub-class of Storable for an array, if all instances are of the same class like in the following example:

```
HomogeneousMM[] hmAr = ArrayHandler.createArrayOf( hMM, 5 );
XMLParser.appendObjectWithTags( buffer, hmAr, "hmAr" );
hmAr = (HomogeneousMM[]) XMLParser.extractObjectForTags( buffer, "hmAr" )
    ;
```

3.2 Parameters & ParameterSets

Parameters in Jstacs are represented by different sub-classes of Parameter, which define different types of parameters. Parameters that take primitives or strings as values are defined by the class SimpleParameter, parameters that accept values from some `enum` type are defined by EnumParameter, parameters where the user can select from a number of predefined values are defined by SelectionParameter, parameters that represent a file argument are defined by FileParameter, and parameters that represent a range of values are represented by RangeParameter. In the following, we give some examples for the creation of parameter objects. Let us assume, we want to define a parameter for the length of the sequences accepted by some model. The maximum sequence length this model can handle is 100 and, of course, lengths cannot be negative. We create such a parameter object by the following lines of code:

```
SimpleParameter simplePar = new SimpleParameter( DataType.INT, "Sequence␣
    length", "The␣required␣length␣of␣a␣sequence", true, new
    NumberValidator<Integer>( 1, 100 ), 10 );
```

The first argument of the constructor defines the data type of the accepted values, which is an `int` in the example. The next two arguments are the name of and the comment for the parameter. The following boolean specifies if this parameter is required (`true`) or optional (`false`). The NumberValidator in the fifth argument allows for specifying the range of allowed values, which is 0 to 100 (inclusive) in the example. Finally, we define a default value for this parameter, which is 10 in the example. Similarly, we can define a SimpleParameter for some optional parameter that takes strings as values by the following line:

```
SimpleParameter simplePar2 = new SimpleParameter( DataType.STRING, "Name"
    , "The␣name␣of␣the␣game", false );
```

Again, the second and third arguments are the name and the comment, respectively.

We can define an EnumParameter, which accept values from some `enum` type as follows

```
EnumParameter enumpar = new EnumParameter( DataType.class, "Data␣types",
    true );
```

where the first argument defines the class of the `enum` type, the second is the name of that collection of values, and the third argument again specifies if this parameter is required.

A SelectionParameter accepts values from a pre-defined collection of values. For instance, if we want the user to select from two double values 5.0 and $5E6$, which are named "small" and "large", we can do so as follows:

```
SelectionParameter collPar = new SelectionParameter( DataType.DOUBLE, new
     String[]{"small", "large"}, new Double[]{5.0,5E6}, "Numbers", "A␣
    selection␣of␣numbers", true );
```

For the special case, where the user shall select the concrete implementation of an abstract class of interface, Jstacs provides a static convenience method `getSelectionParameter` in the class SubclassFinder. This method requires the specification of the super-class of the ParameterSetthat can be used to instantiate the implementations, the root package in which sub-classes or implementations shall be found, and, again, a name, a comment, and if this parameter is required. For example, we can find all classes that can be instantiated by a sub-class of SequenceScoringParameterSetthe package `de` and its sub-packages by calling

```
collPar = SubclassFinder.getSelectionParameter(
    SequenceScoringParameterSet.class, "de", "Sequence␣scores", "All␣
    Sequence␣scores␣in␣Jstacs␣that␣can␣be␣created␣from␣parameter␣sets",
    true );
```

The method returns a SelectionParameter from which a user can select the appropriate implementation. Classes that can be found in this manner must implement an additional interface called InstantiableFromParameterSet. The main purpose of this interface is that implementing classes must provide a constructor that takes a InstanceParameterSet as its only argument in analogy to the constructor of Storable working on a `StringBuffer`. InstanceParameterSets will be explained a few lines below.

As the name suggests, ParameterSets represent sets of such parameters. The most simple implementation of a ParameterSet is the SimpleParameterSet, which can be created just from a number of Parameters like in the following example:

```
SimpleParameterSet parSet = new SimpleParameterSet( simplePar,collPar );
```

Other ParameterSets are the ExpandableParameterSet and ArrayParameterSet, which can handle series of identical parameter types.

One special case of ParameterSets is the InstanceParameterSet, which has several sub-classes that can be used to instantiate new Jstacs objects like statistical models, scoring functions, or classifiers. If a new model, say an implementation of the TrainableStatisticalModel interface, shall be found via the SubclassFinder, or its parameters shall be set in a command line program using the ParameterSetTagger or in Galaxy, we need to create a new sub-class of InstanceParameterSet that represents all (external) parameters of this model. In this sub-class we must basically implement two methods: `getInstanceName` and `getInstanceComment` return the name of and a comment on the model class (i.e., in the example, the model we just implemented) that may be of help for a potential user. The constructor does the main work. By a call to the super-constructor, it initializes the list of parameters in this set and then adds the parameters of the model. For implementations of the TrainableStatisticalModel interface we may also extend SequenceScoringParameterSet, which already handles the AlphabetContainer and length of this model.

Not always do we have flat hierarchies of parameters. For instance, the choice of subsequent parameters may depend on the selection from some SelectionParameter. For this purpose, Jstacs provides a sub-class of Parameter that only serves as a container for a ParameterSet and is called ParameterSetContainer. Like other parameters, this container takes a name and a comment in its constructor, whereas the third argument is a ParameterSet:

```
ParameterSetContainer container = new ParameterSetContainer( "Set", "A␣
    set␣of␣parameters", parSet );
```

Since such a ParameterSetContainer can itself be part of another ParameterSet, we can build hierarchies or trees of Parameters and ParameterSets. ParameterSetContainers are also used internally to create SelectionParameters from an array of ParameterSets, e.g., for `getSelectionParameter` in the SubclassFinder.

3.3 Results & ResultSets

We distingiush two types of Results, namely NumericalResults and CategoricalResults. The first are results containing numerical values, which can be aggregated, for instance averaged, while the latter are results of categorial values like strings or booleans. For example, we can create a NumericalResult containing a single `double` value by the following line

```
NumericalResult res = new NumericalResult( "A␣double␣result", "This␣
    result␣contains␣some␣double␣value", 5.0 );
```

where, in analogy to Parameters, the first and the second argument are the name of and a comment on the result, respectively.

Similarly, we create a CategoricalResult by the following line

```
CategoricalResult catRes = new CategoricalResult( "A␣boolean␣result", "
    This␣result␣contains␣some␣boolean", true );
```

for a result that is a single `boolean` value.

As for ParameterSets, we can create sets of results using the class ResultSet

```
ResultSet resSet = new ResultSet( new Result[]{res,catRes} );
```

where we may also combine NumericalResults and CategoricalResults into a single set. Besides simple ResultSets, Jstacs comprises NumericalResultSets for combining only NumericalResults, which can be created in complete analogy to ResultSets.

Jstacs also provides a special class for averaging NumericalResult. This class is called MeanResultSet, and computes the average and standard error of the corresponding values of a number of NumericalResultSets. The corresponding NumericalResults in the NumericalResultSet are identified by their name as speficied upon creation.

We first create an empty MeanResultSet by calling its default constructor

```
MeanResultSet mrs = new MeanResultSet();
```

and susequently add a number of NumericalResultSets to this MeanResultSet.

```
Random r = new Random();
for(int i=0;i<10;i++){
 mrs.addResults( new NumericalResultSet( new NumericalResult( "Single", "
     A␣single␣result␣to␣be␣aggregated", r.nextDouble() ) ) );
}
```

In the example, these are just 10 uniformly distributed random numbers.

Finally, we call the method `getStatistics` of MeanResultSet to obtain the mean and standard error of the previously added values.

```
System.out.println( mrs.getStatistics() );
```

the result of this method is again returned as a NumericalResultSet. In the example, it is just printed to standard out.

4 First main course: Models

Statistical models that can be learned on a single input data set are represented by the interface TrainableStatisticalModel of Jstacs. In most cases, such models are learned by generative learning principles like maximum likelihood or maximum a-posteriori. For models that are learned from multiple data sets, commonly by discriminative learning principles, Jstacs provides another interface DifferentiableStatisticalModel, which will be presented in the next section.

In the following, we briefly describe all methods that are defined in the TrainableStatisticalModel interface. For convenience, an abstract implementation AbstractTrainSM of TrainableStatisticalModel exists, which provides standard implementations for many of these methods.

TrainableStatisticalModel extends the standard interface `Cloneable` and, hence, implementations must provide a `clone()` method, which returns a deep copy of all fields of an instance:

```
 *              if something went wrong while cloning
```

Since the implementation of the clone method is very model-specific, it must be implemented anew for each implementation of the TrainableStatisticalModel interface.

The parameters of statistical models are typically learned from some training data set. For this purpose, TrainableStatisticalModel specifies a method `train`

```
/**
```

that learns the parameters from the training data set `data`. By specification, successive calls to `train` must result in a model trained on the data set provided in the last call, as opposed to incremental learning on all data sets.

Besides this simple `train` method, TrainableStatisticalModel also declares another one

```
/**
```

that allows for the specification of weights for each input sequence. Since the previous method is a special case of this one where all weights are equal to 1, only the weighted variant must be implemented, if you decide to extend AbstractTrainSM. This method should be implemented such that the specification of `null` weights leads to the standard variant with all weights equal to 1. The actual training method may be totally problem and implementation specific. However, in most cases you might want to use one of the generative learning principles ML or MAP.

After a model has been trained it can be used to compute the likelihood of a sequence given the model and its (trained) parameters. The TrainableStatisticalModel interface specifies a number of methods for this purpose.

The first method just requires the specification of the Sequence object for which the likelihood is to be computed:

```
 *
 * @param seq
```

If the TrainableStatisticalModel has not been trained prior to calling this method, it is allowed to throw a `NotTrainedException`. The meaning of this method is slightly different for inhomogeneous, that is position-dependent, and homogeneous models. In case of an inhomogeneous model, for instance a position weight matrix, the specified Sequence must be of the same length as the model, i.e. the number of columns in the weight matrix. Otherwise an Exception should be thrown, since users may be tempted to misinterpret the return value as a probability of the complete, possibly longer or shorter, provided sequence. In case of homogeneous models, for instance homogeneous Markov models, this method must return the likelihood of the complete sequence. Since this is not always the desired result, to other methods are specified, which allow for computing the likelihood of sub-sequences. This method should also check if the provided Sequence is defined over the same AlphabetContainer as the model.

The first of these methods is

```
 * as for the method {@link #getLogScoreFor(Sequence)}.
 *
```

which computes the likelihood of the sub-sequence starting at position `startpos`. The resulting value of the likelihood must be the same as if the user had called `getProbFor(sequence.getSubSequence(startpos))`.

The second method reads

```
/**
```

and computes the likelihood of the sub-sequence starting at position `startpos` up to position `endpos` (inclusive). The resulting value of the likelihood must be the same as if the user had called `getProbFor(sequence.getSubSequence(startpos,endpos-startpos`1))+. Only the last method must be implemented if we decide for extending AbstractTrainSM, since the previous two can again be perceived as special cases.

In some cases, for instance for very long sequences, the computation of the likelihood may lead to numerical problems. Hence, the TrainableStatisticalModel interface in complete analogy defines methods for computing the log-likelihood. These methods are

Although the implementation of the log-variants is not required if you extend AbstractTrainSM, we strongly recommend to also implement `getLogProbFor(Sequence,int,int)` because otherwise it defaults to `Math.log(getProbFor(Sequence,int,int))` and, hence, inherits numerical problems that may occur for this method.

For convenience, TrainableStatisticalModel also provides a method for computing the log-likelihoods of all Sequences in a DataSet

which is already implemented in AbstractTrainSMby successive calls to `getLogProbFor(Sequence)`. This method also exists in a variant

where the user may specify an existing array for storing the computed log-likelihoods. This may be reasonable to save memory, for instance if we compute the log-likelihoods of a large number of sequences using different models.

If we want to use Bayesian principles for learning the parameters of a model, we need to specify a prior distribution on the parameter values. In some cases, for instance for using MAP estimation in an expectation maximization (EM) algorithm, it is not only necessary to estimate the parameters taking the prior into account, but also to know the value of the prior (or a term proportional to it). For this reason, the TrainableStatisticalModel interface defines the methods

and

which return this prior term and its logarithm, respectively. In the default implementation of AbstractTrainSM, the first method defaults to `Math.exp(getLogPriorTerm())`.

If the concept of a prior is not applicable for a certain model or other reasons prevent you from implementing these methods, `getLogPriorTerm()` should return `Double.NEGATIVE_INFINITY`.

Generative TrainableStatisticalModels can also be used to create new, artificial data according to the model assumptions. For this purpose, the TrainableStatisticalModel interface specifies a method wich returns a DataSet of `numberOfSequences` Sequences drawn from the model using its current parameter values. The second parameter `seqLength` allows for the specification of the lenghts of the drawn Sequences. For inhomogeneous model, which inherently define the length of possible sequence, this parameter should be `null` or an array of length 0. For homogeneous models, the lengths may either be specified for all drawn sequences by a single `int` value or by an array of length `numberOfSequences` specifying the length of each drawn sequence independently.

The implementation of this method is not always possible. In its default implementation in AbstractTrainSM, this method just throws an Exception and must be explicitly overridden to be functional.

TrainableStatisticalModel also defines some methods that are basically getters for typical properties of a TrainableStatisticalModel implementation. These are

which returns the current AlphabetContainer of the model,

which returns a (helpful) name of the current TrainableStatisticalModel instance,

which returns the length of the TrainableStatisticalModel (0 for homogeneous models), and

which returns the maximum number of preceeding positions that are considered for (conditional) probabilities. For instance, for a position weight matrix, this method should return 0, whereas for a homogeneous Markov model of order 2, it should return 2.

The method

must return `true` if the model has already been trained, i.e., the train method has been called at least once, and `false` otherwise.

The methods

and

can be used to return some properties of a TrainableStatisticalModel like the number of parameters, the depth of some tree structure, or whatever seems useful. In the latter case, these characteristics are limited to numerical values. If a model is used, e.g., in a cross validation (KFoldCrossValidation), these numerical properties are averaged over all cross validation iterations and displayed together with the performance measures.

The method

should return some `String` representation of the current model. Typically, this representation should include the current parameter values in some suitable format.

Finally, the method

can be used to replace the current AlphabetContainer by some (compatible) other AlphabetContainer. Compatible means that the new AlphabetContainer must define an identical alphabet, although it may be a different instance. This method may be helpful if in successive evaluations the consistency check between alphabets can be reduced to the comparison of references. The default implement in AbstractTrainSM should do the right thing in most cases.

Besides the possibility to implement new statistical models in Jstacs, many of them are already implemented and can readily be used. As a central facility for creating model instances of many standard models, Jstacs provides a TrainSMFactory.

Using the TrainSMFactory, we can create a new position weight matrix (PWM) model by calling

```
TrainableStatisticalModel pwm = TrainableStatisticalModelFactory.
    createPWM( alphabet, 10, 4.0 );
```

where we need to specify the (discrete) alphabet, the length of the matrix (10), i.e. the number of positions modeled, and an equivalent sample size (ESS) for MAP estimation (4.0). For the concept of an equivalent sample size and the BDeu prior used for most models in Jstacs, we refer the reader to (Heckerman). If the ESS is set to 0.0, the parameters are estimated by the ML instead of the MAP principle.

The factory method for an inhomogeneous Markov model of arbitrary order – the PWM model is just an inhomogeneous Markov model of order 0 – is

```
TrainableStatisticalModel imm = TrainableStatisticalModelFactory.
    createInhomogeneousMarkovModel( alphabet, 12, 4.0, (byte) 2 );
```

where the parameters are in complete analogy to the PWM model, expect the last one specifying the order of the inhomogeneous Markov model, which is 2 in the example.

We can also create permuted Markov models of order 1 and 2, where the positions of the sequences may be permuted before building an inhomogeneous Markov model. The permutation is chosen such that the mutual information between adjacent positions is maximized. We create a permuted Markov model of length 7 and order 1 by calling

```
TrainableStatisticalModel pmm = TrainableStatisticalModelFactory.
    createPermutedMarkovModel( alphabet, 7, 4.0, (byte) 1 );
```

Homogeneous Markov models are created by

```
TrainableStatisticalModel hmm = TrainableStatisticalModelFactory.
    createHomogeneousMarkovModel( alphabet, 400.0, (byte) 3 );
```

where the first parameter again specifies the alphabet, the second parameter

```
TrainableStatisticalModel zoops = TrainableStatisticalModelFactory.
    createZOOPS( pwm, hmm, new double[]{4,4}, true );

pwm.train( sam );

HomogeneousMM hmm2 = new HomogeneousMM( new HomMMParameterSet( alphabet,
    4.0, "hmm(0)", (byte) 0 ) );

BayesianNetworkTrainSM bnm = new BayesianNetworkTrainSM( new
    BayesianNetworkTrainSMParameterSet( alphabet, 8, 4.0, "Bayesian␣
    network", ModelType.BN, (byte) 1, LearningType.ML_OR_MAP ) );

HMMTrainingParameterSet trainingPars = new BaumWelchParameterSet( 5, new
    SmallDifferenceOfFunctionEvaluationsCondition( 1E-6 ),2 );
Emission[] emissions = new Emission[]{new DiscreteEmission( alphabet, 4.0
     ),new DiscreteEmission( alphabet, new double[]{2.0,1.0,1.0,2.0} )};
AbstractHMM myHMM = HMMFactory.createErgodicHMM( trainingPars, 1, 4.0,
    0.1, 100.0, emissions );

HigherOrderHMM hohmm = new HigherOrderHMM( trainingPars, new String[]{"A"
    ,"B"}, emissions,
  new TransitionElement( null, new int[]{0}, new double[]{4.0} ),
  new TransitionElement( new int[]{0}, new int[]{0,1}, new double
     []{2.0,2.0} ),
  new TransitionElement( new int[]{1}, new int[]{0}, new double[]{4.0} ))
     ;
```

```
System.out.println( hohmm.getGraphvizRepresentation( null ) );

MixtureTrainSM mixEm = new MixtureTrainSM( 8, new
    TrainableStatisticalModel[]{pwm,pwm}, 3, new double[]{4,0,4.0}, 1,
    new SmallDifferenceOfFunctionEvaluationsCondition( 1E-6 ),
    Parameterization.LAMBDA );

MixtureTrainSM mixGibbs = new MixtureTrainSM( 8, new
    TrainableStatisticalModel[]{pwm,pwm}, 3, new double[]{4,0,4.0}, 100,
    1000, new VarianceRatioBurnInTest( new
    VarianceRatioBurnInTestParameterSet( 3, 1.2 ) ) );

StrandTrainSM strandModel = new StrandTrainSM( imm, 3, 0.5, 1, new
    SmallDifferenceOfFunctionEvaluationsCondition( 1E-6 ),
    Parameterization.LAMBDA );

ZOOPSTrainSM zoops2 = new ZOOPSTrainSM( pwm, hmm, true, 4, 0.7, null, 1,
    new SmallDifferenceOfFunctionEvaluationsCondition( 1E-6 ),
    Parameterization.LAMBDA );
```

5 Second main course: ScoringFunctions

6 Third main course: Classifiers

7 Intermediate course: Optimization

8 Dessert: Alignments

9 Like some sweets: Utils and goodies