Jstacs Cookbook

Jan Grau and Jens Keilwagen

May 15, 2012

Contents

1 Preface

Sequence analysis is one of the major subjects of bioinformatics. Several existing libraries combine the representation of biological sequences with exact and approximate pattern matching as well as alignment algorithms. We present Jstacs, an open source Java library, which focuses on the statistical analysis of biological sequences instead. Jstacs comprises an efficient representation of sequence data and provides implementations of many statistical models with generative and discriminative approaches for parameter learning. Using Jstacs, classifiers can be assessed and compared on test datasets or by cross-validation experiments evaluating several performance measures. Due to its strictly object-oriented design Jstacs is easy to use and readily extensible.

Jstacs is a joint project of the groups Bioinformatics and Pattern Recognition and Bioinformatics at the Institute of Computer Science of Martin Luther University Halle-Wittenberg and the Research Group Data Inspection at the Leibniz Institute of Plant Genetics and Crop Plant Research.

Jstacs is listed in the machine learning open-source software (mloss) repository.

The complete API documentation can be found at `http://www.jstacs.de/api-2.0/`.

A reference card of Jstacs classes and methods can be found at the end of this cookbook or as a separate download.

1.1 General structure of Jstacs

A coarse view on the structure of Jstacs is presented in Figure 1. Being a library for statistical analysis and classification of sequence data, Jstacs is organized around the abstract class AbstractClassifier, the interface StatisticalModel and its two sub-interfaces TrainableStatisticalModel, and DifferentiableStatisticalModel.

StatisticalModels represent statistical models in general, which can compute the log-likelihood of a given input sequence and define prior densities on their parameters. The abstract implementation AbstractTrainableStatisticalModel of TrainableStatisticalModelis the base class of many generatively learned models such as Bayesian networks, hidden Markov models, or mixture models, and can be learned from a single input data set. In constrast, DifferentiableStatisticalModels provide all facilities for numerical optimization of parameters, which is especially necessary for discriminative parameter learning. The abstract base class of all DifferentiableStatisticalModel implementations is AbstractDifferentiableStatisticalModel. Currently, DifferentiableStatisticalModels include Bayesian networks, Markov models, a ZOOPS model, and mixture models.

AbstractClassifier defines the general properties of a classifier. Its sub-class AbstractScoreBasedClassifier adds additional methods for the classification of sequences based on a

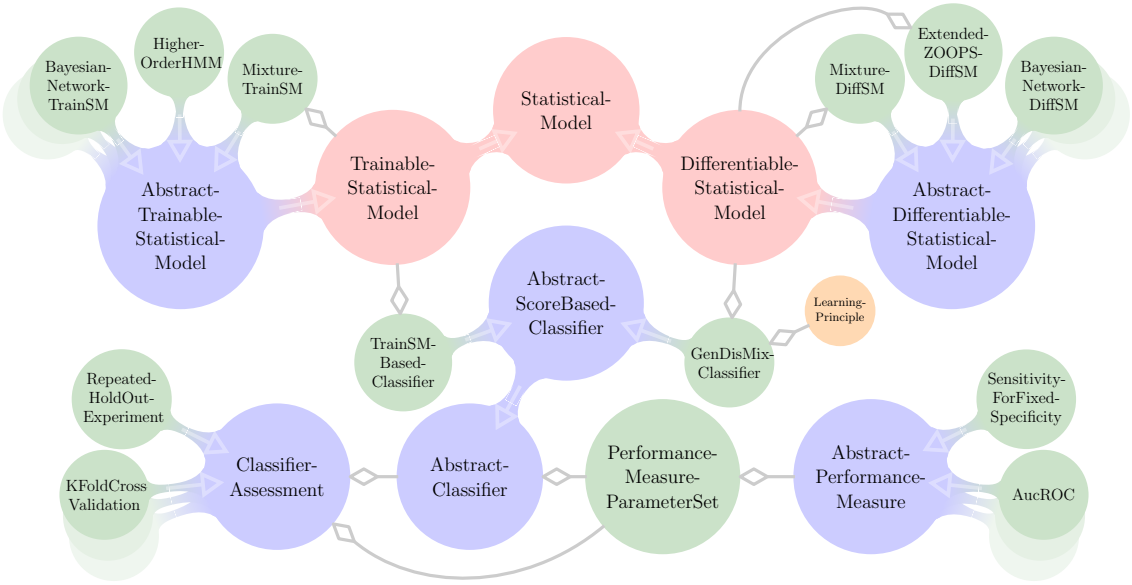

Figure 1: Part of the class structure of Jstacs. Interfaces are depicted in red, abstract classes in blue, concrete classes in green, and enums in orange. Continuous transitions represent inheritance, whereas arrows with diamond heads represent usage.

sequence and class-specific score. Two concrete sub-classes of AbstractScoreBasedClassifier are the TrainSMBasedClassifier, which works on TrainableStatisticalModels, and the GenDisMixClassifier, which works on DifferentiableStatisticalModel.

AbstractClassifiers can be assessed either on dedicated training and test data sets or in ClassifierAssessments like KFoldCrossValidation or RepeatedHoldOutExperiment. The performance measures used in such an assessment are collected in PerformanceMeasureParameterSets containing sub-classes of the abstract class AbstractPerformanceMeasure.

A more detailed view on all of these classes will be given in the remainder of this cookbook.

1.2 About this cookbook

This document is not a cookbook in the sense of a collection of recipes. The intention of this cookbook is rather to learn how to cook with the ingredients and tools provided by Jstacs.

Nonetheless, we present a collection of recipes in the last section, where you find the code of executable code examples that can also be downloaded as a zip file.

We are aware that a library of this size seems daunting on first sight – and on second sight as well. However, we hope that despite the inevitable complexity and size of such a library, this cookbook may help to get a picture of the structure, design principles, and capabilities of Jstacs.

This cookbook is structured as follows: in the section "Quick start: Jstacs in a nutshell", we give a very brief introduction to Jstacs enabling to run first programs without reading the complete cookbook.

In the section "Starter: Data handling", we explain how data are represented in Jstacs, and how you can read data from files.

In section "Intermediate course: XMLParser, Parameters, and Results", we present some facilities, an XML parser and the representation of parameters and results, that are used frequently within Jstacs and are necessary for the following parts.

In section "First main course: SequenceScores", we present sequence scores, statistical models, and their sub-interfaces and sub-classes. We explain the methods defined in these interfaces and classes, and we show how you can create and use their existing implementations.

In section "Second main course: Classifiers", we explain classifiers and assessment of classifiers using different performance measures.

In section "Intermediate course: Optimization", we present the facilities for numerical optimization.

In section "Dessert: Alignments, Utils, and goodies", we list utility classes and methods, that we think might be of help for your own implementations.

Finally, in section "Recipes", we give a number of executable code examples that may serve as a starting point of your own classes and applications.

2 Quick start: Jstacs in a nutshell

This section is for the unpatient who like to directly start using Jstacs without reading the complete cookbook. If you do not belong to this group, you can skip this section.

Here, we provide code snippets for simple tasks including reading a data set or creating models and classifiers that might be frequently used. In addition, some of the basic code examples in section Recipes may also serve as a basis for a quick start into Jstacs.

For reading a FastA file, we call the constructor of the DNADataSet with the (absolute or relative) path to the FastA file. Subsequently, we can determine the alphabets used.

```
DNADataSet ds = new DNADataSet( args[0] );
AlphabetContainer con = ds.getAlphabetContainer();
```

Detailed information about data sets, sequences, and alphabets can be found in section 3.

2.1 Statistical models and classifiers using generative learning principles

In Jstacs, statistical models that use generative learning principles to infer their parameters implement the interface TrainableStatisticalModel. For convenience, Jstacs provides the TrainableStatisticalModelFactory, which allows creating various simple models in an easy manner. Creating for instance a position weight matrix (PWM) model is just one line of code.

```
TrainableStatisticalModel pwm =
    TrainableStatisticalModelFactory.createPWM( con,
    ds.getElementLength(), 4 );
```

Similarily, other models including inhomogeneous Markov models, permuted Markov models, Bayesian networks, homogeneous Markov models, ZOOPS models, and hidden Markov models can be created using the TrainableStatisticalModelFactory and the HMMFactory, respectively.

Given some model `pwm`, we can directly infer the model parameters based on some data set `ds` using the `train` method.

```
pwm.train( ds );
```

After the model has been trained, it can be used for scoring sequences using the `getLogProbFor` methods. More information about the interface TrainableStatisticalModel can be found in section 5.2.1.

We can build a classifier based on a set of TrainableStatisticalModels, e.g. two PWM models.

```
AbstractClassifier cl = new TrainSMBasedClassifier( pwm, pwm );
```

2.2 Further statistical models and classifiers

Sometimes, we like to learn statistical models by other learning principles that require numerical optimization. For this purpose, Jstacs provides the interface DifferentiableStatisticalModel and the factory DifferentiableStatisticalModelFactory in close analogy to TrainableStatisticalModel and TrainableStatisticalModelFactory (cf. 5.2.2). Creating a classifier using two PWM models and the maximum supervised posterior learning principle can be accomplished by calling

```
DifferentiableStatisticalModel pwm =
    DifferentiableStatisticalModelFactory.createPWM(con, 10, 4);

GenDisMixClassifierParameterSet pars = new
    GenDisMixClassifierParameterSet(con,10,(byte)10,1E-9,1E-10,1,
    false,KindOfParameter.PLUGIN,true,1);

AbstractClassifier cl = new MSPClassifier( pars, pwm, pwm );
```

2.3 Using classifiers

As we have seen in the previous subsections, we can build classifiers based on statistical models. The main functionality is predicting the class label of a sequence and assessing the performance of a classifier. For these tasks, Jstacs provides the methods `classify` and `evaluate`, respectively.

For classifying a sequence, we call

```
byte res = cl.classify( seq );
```

on a trained classifier. The method returns numerical class labels in the same order as data is provided for training starting from index 0.

For evaluating the performance of a classifier, we need to compute some performance measures. For convenience, Jstacs provides the possibility of getting a bunch of standard measures including point measures and areas under curves (cf. 6.3). Based on such measures, we can determine the performance of the classifier.

```
NumericalPerformanceMeasureParameterSet params =
    PerformanceMeasureParameterSet.createFilledParameters();
System.out.println( cl.evaluate(params, true, data) );
```

Here, `true` indicates that an `Exception` should be thrown if a measure could not be computed, and `data` is an array of data sets, where the index within this array encodes the class.

For assessing the performance of a classifier using some repeated procedure of training and testing, Jstacs provides the class ClassifierAssessment (cf. 6.4).

3 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 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.

3.1 Alphabets

Since Jstacs is tailored 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 minimum and maximum 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 DNAAlphabet is always the same, a singleton for such an AlphabetContainer is 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.

3.2 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 subsection. 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".

3.3 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 DNADataSet with the (absolute or relative) path to the FastA file:

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

For other file formats and types of Sequences, DataSet provides another constructor that works on the AlphabetContainer for 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 contDataSet = new DataSet( contContainer, new
    SparseStringExtractor( "myfile.tab", '#' ), "\t" );
```

The SparseStringExtractor is tailored 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 sparseDataSet = 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 = dnaDataSet.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<dnaDataSet.getNumberOfElements();i++){
 System.out.println(dnaDataSet.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 : contDataSet){
 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 = dnaDataSet.getInfixDataSet( 3, 10 );
```

a DataSet of all suffixes starting at position 7 from

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

or a DataSet containing all reverse complementary Sequences using

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

For cross-validation experiments, hold-out samplings, or similar procedures (cf. package assessment), it is useful to partition a sample randomly. DataSets in Jstacs support two types of partitionings. The first is to partition a DataSet into k almost 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 = dnaDataSet.partition(
    PartitionMethod.PARTITION_BY_NUMBER_OF_ELEMENTS, 5 );
```

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

```
DataSet[] randParts = dnaDataSet.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( dnaDataSet, 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 4, 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 databases like Genbank using an adaptor to BioJava.

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

```
GenbankRichSequenceDB db = new GenbankRichSequenceDB();

SequenceIterator dbIterator = new SimpleSequenceIterator(
  db.getRichSequence( "NC_001284" ),
  db.getRichSequence( "NC_000932" )
);
```

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.

4 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, Singletons, Strings, Classes, 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.

4.1 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 appending these two elements, the resulting XML text will be

```
<integer><className>java.lang.Integer</className>5</integer>
<foo><className>java.lang.String</className>hello world</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(
    DNAAlphabetContainer.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 implementing 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 singletons, 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 `double`s 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" );
```

4.2 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 ParameterSet that 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 SequenceScoringParameterSet the 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 subclasses that can be used to instantiate new Jstacs objects like statistical models 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 subclass 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.

4.3 Results & ResultSets

In Jstacs, several types of Results are implemented. The two basic Result types are NumericalResults and CategoricalResults. The first are results containing numerical values, which can be aggregated, for instance averaged, while the latter are results of categorical 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 specified upon creation.

We first create an empty MeanResultSet by calling its default constructor

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

and subsequently 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.

5 First main course: SequenceScores

SequenceScores define scoring functions over sequences that can assign a score to each input sequence. Such scores can then be used to classify sequences by chosing the class represented by that SequenceScore yielding the maximum score.

Sub-iterfaces of SequenceScore are DifferentiableSequenceScore, which is tailored to numerical optimization, and StatisticalModel, which represents statistical models defining a proper likelihood. StatisticalModel, in turn, has two sub-interfaces TrainableStatisticalModel and DifferentiableStatisticalModel, where the latter joins the properties of StatisticalModels and DifferentiableSequenceScores. All of these will be explained in the sub-sections of this section.

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

```
public SequenceScore clone() throws CloneNotSupportedException;
```

Since the implementation of the clone method is very score-specific, it must be implemented anew for each implementation of the SequenceScore interface. Besides that SequenceScore also extends Storable, which demands the implementation of the `toXML()` method and a constructor working on a `StringBuffer` containing an XML-representation (see section 4.1).

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

```
public AlphabetContainer getAlphabetContainer();
```

which returns the current AlphabetContainer of the model, i.e., the AlphabetContainer of Sequences this SequenceScore can score,

```
public String getInstanceName();
```

which returns a - distinguishing - name of the current SequenceScore instance, and

```
public int getLength();
```

which returns the length of the SequenceScore, i.e. the possible length of input sequences. This length is defined to be zero, if the score can handle Sequences of variable length.

The methods

```
public ResultSet getCharacteristics() throws Exception;
```

and

```
public NumericalResultSet getNumericalCharacteristics() throws
    Exception;
```

can be used to return some properties of a SequenceScore 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 scoring of input Sequences is performed by three methods `getLogScore`. The name of this method reflects that the returned score should be on a lagorithmic scale. For instance, for a stastical model with a proper likelihood over the sequences this score would correspond to the log-likelihood. The first method

```
public double getLogScoreFor(Sequence seq);
```

returns the score for the given Sequence. For reasons of efficiency, it is not intended to check if the length or the AlphabetContainer of the Sequence match those of the SequenceScore. Hence, no exception is declared for this method. So if this method is used (instead of the more stringent methods of StatisticalModel, see 5.2) this should be checked externally.

The second method is

```
public double getLogScoreFor(Sequence seq, int start);
```

which returns the score for the sub-sequence starting at position `start` end extending to the end of the Sequence or the end of the SequenceScore, whichever comes first.

The third method is

```
public double getLogScoreFor(Sequence seq, int start, int end)
    throws Exception;
```

and computes the score for the sub-sequence between `start` and `end`, where the value at position `end` is the last value considered for the computation of the score.

Two additional methods

```
public double[] getLogScoreFor(DataSet data) throws Exception;
```

and

```
public void getLogScoreFor(DataSet data, double[] res) throws
    Exception;
```

allow for the computation of such scores for all Sequences in a DataSet. The first method returns these scores, whereas the second stores the scores to the provided `double` array. This may be reasonable to save memory, for instance if we compute the log-likelihoods of a large number of sequences using different models.

The result of this method (w.r.t. one Sequence) should be identical to independent calls of `getLogScore`. Hence, these method are pre-implemented exactly in this way in the abstract implementations of SequenceScore.

Finally, the return value of the method

```
public boolean isInitialized();
```

indicates if a SequenceScore has been initialized, i.e., all parameters have been set to some (initial or learned) value and the SequenceScore is ready to score Sequences.

5.1 DifferentiableSequenceScores

The interface DifferentiableSequenceScore extends SequenceScore and adds all methods that are necessary for a numerical optimization of the parameters of a SequenceScore. This especially includes the computation of the gradient w.r.t. the parameters for a given input Sequence.

Before we can start a numerical optimization, we first need to specify initial parameters. DifferentiableSequenceScore declares two methods for that purpos. The first is

```
public void initializeFunction(int index, boolean freeParams,
  DataSet[] data, double[][] weights) throws Exception;
```

which is tailored to initializing the parameters from a given data set. The parameters have the following meaning: `index` is the index of the class this DifferentiableSequenceScore represents. For some parameterizations, we can either optimize (and initialize) only the free parameters of the model, i.e., excluding parameters with values that can be computed from the values of a subset of the remaining parameters, or we can optimize all parameters. For example such parameters could be those of a DNA dice, where the probability of one nucleotide is fixed if we know the probabilities of all other nucleotides. If `freeParams` is `true`, only the free parameters are used. The array `data` holds the input data, where `data[index]` holds the data for the current class, and `weights` are the weights on each Sequence in `data`.

The second method for initialization is

```
public void initializeFunctionRandomly(boolean freeParams) throws
    Exception;
```

which initializes all parameters randomly, where the exact method for drawing parameter values is implementation-specific.

Like for the method `getLogScore` of SequenceScore, there are three methods for computing the gradient w.r.t. the parameters. The first is

```
public double getLogScoreAndPartialDerivation(Sequence seq,
    IntList indices, DoubleList partialDer);
```

where `seq` is the Sequence for which the gradient is to be computed, and `indices` and `partialDer` are lists that shall be filled with the indexes and values of the partial derivation, respectively. Assume that we implement a DifferentiableSequenceScore f with N

parameters and $d_i = \frac{\partial \log f}{\partial \lambda_i}$ is the partial derivation of the log-score w.r.t. the i-th parameter of f. Then we would add the index i to `indices` and (before subsequent additions to `indices` or `partialDer`) add d_i to `partialDer`. Partial derivations, i.e., components of the gradient, that are zero may be omitted.

The other two methods

```
public double getLogScoreAndPartialDerivation(Sequence seq, int
    start, IntList indices, DoubleList partialDer);
```

and

```
public double getLogScoreAndPartialDerivation( Sequence seq, int
    start, int end, IntList indices, DoubleList partialDer )
    throws Exception;
```

additionally allow for the specification of a start (and end) position within the Sequence `seq` in the same manner as specified for `getLogScore` of SequenceScore.

The method

```
public int getNumberOfParameters();
```

returns the number of parameters (N in the example above) of parameters of this DifferentiableSequenceScore. It is used, for example, to create temporary arrays for the summation of gradients during numerical optimization.

Since numerical optimization for non-convex problems may get stuck in local optima, we often need to restart numerical optimization multiple times from different initial values of the parameters. As the number of starts that is required to yield the global optimum with high probability is highly score-specific, DifferentiableSequenceScore defines a method

```
public int getNumberOfRecommendedStarts();
```

that returns the number of starts that is recommended by the developer of this DifferentiableSequenceScore. If multiple DifferentiableSequenceScore are used in a numerical optimization (for instance for the different classed), the maximum of these values is used in the optimization. If your parameterization is known to be convex, this method should return 1.

The currently set parameters (either by initialization or explicitly, see below) can be obtained by

```
public double[] getCurrentParameterValues() throws Exception;
```

and new parameter values may be set by

```
public void setParameters(double[] params, int start);
```

where `start` is the index of the value for the first parameter of this DifferentiableSequenceScore within `params`.

Finally, the method

```
public double getInitialClassParam(double classProb);
```

returns the class parameter for a given class probability. The default implementation in the abstract class AbstractDifferentiableSequenceScore returns `Math.log(classProb)`. This abstract class also provides standard implementations for many of the other methods.

5.2 StatisticalModels

StatisticalModels extend the specification of SequenceScores by the computation of proper (normalized) likelihoods for input sequences. To this end, it defines three methods for computing the log-likelihood of a Sequence given a StatisticalModel and its current parameter values.

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

```
public double getLogProbFor(Sequence sequence) throws Exception;
```

If the StatisticalModel has not been trained prior to calling this method, it is allowed to throw an `Exception`. 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 models that can handle Sequences of varying lengths, 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 second method is

```
public double getLogProbFor(Sequence sequence, int startpos)
    throws Exception;
```

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

The third method reads

```
public double getLogProbFor(Sequence sequence, int startpos, int
    endpos) throws Exception;
```

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))`.

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 StatisticalModel interface defines the method

```
public double getLogPriorTerm() throws Exception;
```

which returns the logarithm of this prior term.

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

StatisticalModels can also be used to create new, artificial data according to the model assumptions. For this purpose, the StatisticalModel interface specifies a method

```
public DataSet emitDataSet(int numberOfSequences, int...
    seqLength)
  throws NotTrainedException, Exception;
```

which 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 variable-length 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 the abstract implementations of StatisticalModel, this method just throws an Exception and must be explicitly overridden to be functional.

The last method defined in StatisticalModel is

```
public byte getMaximalMarkovOrder() throws
    UnsupportedOperationException;
```

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.

5.2.1 TrainableStatisticalModels

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 (ML) or maximum a-posteriori (MAP). 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 sub-section.

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

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

```
public void train(DataSet data) throws Exception;
```

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 method

```
public void train(DataSet data, double[] weights) throws
    Exception;
```

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 AbstractTrainableStatisticalModel. 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 as defined in StatisticalModel.

The method

```
public String toString();
```

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

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 TrainableStatisticalModel instances of many standard models, Jstacs provides a TrainableStatisticalModelFactory.

Using the TrainableStatisticalModelFactory, 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 is the equivalent sample size, and the third parameter is the order of the Markov model.

We can create a ZOOPS model with a PWM as motif model and the homogeneous Markov model created above as flanking model using

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

where the third parameter contains the hyper-parameters for the two components of the ZOOPS model, and the last boolean allows two switch training of the flanking model on or off. In the example, we choose to train the flanking model and the motif model.

After we created a TrainableStatisticalModel, we can train it on input data. For instance, we train the PWM created above on a training DataSet `ds` by calling:

```
pwm.train( ds );
```

Instead of using the TrainableStatisticalModelFactory, we can also create TrainableStatisticalModels directly using their constructors. For example, we create a homogeneous Markov model of order 0 by calling

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

or we create a Bayesian network of (full) order 1, i.e., a Bayesian tree, of length 8 learned by MAP with an equivalent sample size of 4.0 with

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

Due to the complexity of hidden Markov models, these have their own factory class called HMMFactory. To create a new hidden Markov model, we first need to define the parameters of the training procedure, which is Baum-Welch training in the example

```
HMMTrainingParameterSet trainingPars = new
    BaumWelchParameterSet( 5, new
    SmallDifferenceOfFunctionEvaluationsCondition( 1E-6 ),2 );
```

The parameters provided to that ParameterSet are the number of restarts of the training, the termination condition for each training, and the number of threads used for the (multi-threaded) optimization.

Then we create the emissions of the HMM. Here, we choose two discrete emissions over the DNAAlphabet:

```
Emission[] emissions = new Emission[]{new DiscreteEmission(
    alphabet, 4.0 ),new DiscreteEmission( alphabet, new
    double[]{2.0,1.0,1.0,2.0} )};
```

For the first emission, we use a BDeu prior with equivalent sample size 4, whereas for the second emission, we explictly specify the hyper-parameters of the BDe prior.

Finally, we use the HMMFactory to create a new ergodic, i.e., fully connected, HMM for this training procedure and these emissions:

```
AbstractHMM myHMM = HMMFactory.createErgodicHMM( trainingPars,
    1, 4.0, 0.1, 100.0, emissions );
```

For non-standard architectures of the HMM, we can also use a constructor of HigherOrderHMM, although we only use order 1 in the example,

```
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} ));
```

The arguments are: the training parameters, the names of the (two) states, the emissions as specified above, and TransitionElements that specify the transitions allowed in the HMM. In the example, we may start in the 0-th state (A) represented by the first TransitionElement, from that state we may either loop in state 0 or proceeed to state 1, and from state 1 we may only go back to state 0. For all transitions, we specify hyper-parameters for the corresponding Dirichlet prior.

Once an HMM is created, we can use a method `getGraphvizRepresentation` to obtain the content of a .dot file with a graphical representation (nodes and edges) of the HMM:

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

Another important type of TrainableStatisticalModels within Jstacs are mixture models. Such mixture models can be mixtures of any sub-class of TrainableStatisticalModeland can be learned either by expectation-maximization (EM) or by Gibbs sampling.

We create a mixture model of two PWMs learned by expectation maximization using the constructor

```
MixtureTrainSM mixEm = new MixtureTrainSM( 10, new
    TrainableStatisticalModel[]{pwm,pwm}, 3, new
    double[]{4.0,4.0}, 1, new
    SmallDifferenceOfFunctionEvaluationsCondition( 1E-6 ),
    Parameterization.LAMBDA );
```

where the first argument is the length of the model, the second are the TrainableStatisticalModels used as mixture components, the third is the number of restarts of the EM, the fourth are the hyper-parameters for the prior on the component probabilities, i.e., mixture weights, the fifth is the termination condition, and the last is the parameterization.

For mixture models we can choose between the parameterization in terms of probabilities (called `THETA`) and the so called "natural parameterization" as log-probabilities (called `LAMBDA`).

If we want to use Gibbs sampling instead of EM, we use the constructor

```
MixtureTrainSM mixGibbs = new MixtureTrainSM( 10, new
    TrainableStatisticalModel[]{pwm,pwm}, 3, new
    double[]{4.0,4.0}, 100, 1000, new VarianceRatioBurnInTest(
    new VarianceRatioBurnInTestParameterSet( 3, 1.2 ) ) );
```

where the first 4 arguments remain unchanged, the fifth is the number of intial steps in the stationary phase, the sixth is the total number of stationary samplings, and the seventh is a burn-in test for determining the end of the burn-in phase and the start of the stationary phase.

A special case of a mixture model is the StrandTrainSM, which is a mixture over the two strands of DNA. For both strands, the same component TrainableStatisticalModel is used but it is evaluated once for the original Sequence and once for its reverse complement. We construct a StrandTrainSM of the inhomogeneous Markov model created by the TrainableStatisticalModelFactory by calling

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

where the first argument is the component TrainableStatisticalModel, the second is the number of starts, the third is the probability of the forward strand (which is fixed in the example but can also be learned), the fourth is the hyper-parameter for drawing initial strands (1 results in a uniform distribution on the simplex), the fifth is the termination condition, and the last is the parameterization (as above).

5.2.2 DifferentiableStatisticalModels

DifferentiableStatisticalModels are StatisticalModels that also inherit the properties of DifferentiableSequenceScores. That means, DifferentiableStatisticalModels define a proper likelihood over the input sequences and at the same time can compute partial derivations w.r.t. their parameters.

This also means that they can compute a - not necessarily normalized - log-score using the `getLogScore` methods and normalized log-likelihoods using the `getLogProb` methods. In many cases it is more efficient to use the `getLogScore` methods and, in addition, to compute the normalization constant to obtain proper probabilities seperately. Hence, DifferentiableStatisticalModel defines a method

```
public double getLogNormalizationConstant();
```

that returns the logarithm of that normalization constant, which only depends on the current parameter values but not on the input sequence. For computing the gradients with respect to the parameters, we also need the partial derivations of the log-normalization constant with respect to the parameters. To this end DifferentiableStatisticalModel provides a method

```
public double getLogPartialNormalizationConstant(int
    parameterIndex)
```

that takes the (internal) index of the parameter for which the partial derivation shall be computed as the only argument.

A similar schema is used for the prior on the parameters. In addition to the method `getLogPriorTerm` of StatisticalModel, the method

```
public void addGradientOfLogPriorTerm(double[] grad, int start)
```

adds the values of the gradient of the log-prior with respect to all parameters to the given array `grad` starting at position `start`.

The method

```
public boolean isNormalized();
```

returns `true` if the DifferentiableStatisticalModel is already normalized, that means if `getLogScore` already returns proper probabilities. If that is the case, `getLogNormalizationConstant` must always return 0, and `getLogPartialNormalizationConstant` must return `Double.NEGATIVE_INFINITY` for all parameters.

Finally,

```
public double getESS();
```

returns the equivalent sample size used for the prior of this DifferentiableStatisticalModel.

The DifferentiableStatisticalModel implementations of standard models already exist in Jstacs. For example, we can create a Bayesian network of length 10 with the network structure learned by means of the discriminative "explaining away residual" with

```
BayesianNetworkDiffSM bnDsm = new BayesianNetworkDiffSM( new
    BayesianNetworkDiffSMParameterSet( alphabet, 10, 4.0, true,
    new BTExplainingAwayResidual( new double[]{4.0,4.0} ) ) );
```

where the parameters of the ParameterSet are the AlphabetContainer, the length and the equivalent sample size of the DifferentiableStatisticalModel, a switch wether (generative) plug-in parameters shall be used and an object for the structure measure.

An inhomogeneous Markov model of order 1, length 8, and with equivalent sample size 4.0 is created by

```
MarkovModelDiffSM mmDsm = new MarkovModelDiffSM( alphabet, 8,
    4.0, true, new InhomogeneousMarkov( 1 ) );
```

DifferentiableStatisticalModels are mainly used for discriminative parameter learning in Jstacs. To discriminatively learn the parameters of two inhomogeneous Markov models as created above, we need the following three lines.

```
GenDisMixClassifierParameterSet params = new
    GenDisMixClassifierParameterSet( alphabet, 8,
    Optimizer.QUASI_NEWTON_BFGS, 1E-6, 1E-6, 1, false,
    KindOfParameter.PLUGIN, true, 4 );
GenDisMixClassifier cl = new GenDisMixClassifier( params, new
    CompositeLogPrior(), LearningPrinciple.MSP, mmDsm, mmDsm );
cl.train( data );
```

The discriminative learning principle used in the example is maximum supervised posterior (MSP). For details about the GenDisMixClassifier used in the example, see section 6.2

A homogeneous Markov models of order 3 with equivalent sample size 4.0 is created by the constructor

```
HomogeneousMMDiffSM hmmDsm = new HomogeneousMMDiffSM( alphabet,
    3, 4.0, 100 );
```

The last argument of this constructor is the expected length of the input sequences. This length is used to compute consistent hyper-parameters from the equivalent sample size for this class, since the (transition) parameters of a homogeneous Markov model are used for more than one position of an input sequence.

Similar to hidden Markov models that are TrainableStatisticalModels, we can create a DifferentiableStatisticalModel hidden Markov model with the following lines:

```
DifferentiableEmission[] emissions = new
    DifferentiableEmission[]{new DiscreteEmission( alphabet, 4.0
    ),new DiscreteEmission( alphabet, new
    double[]{2.0,1.0,1.0,2.0} )};
NumericalHMMTrainingParameterSet trainingParameterSet = new
    NumericalHMMTrainingParameterSet( 3, new
    SmallDifferenceOfFunctionEvaluationsCondition( 1E-6 ), 2,
    Optimizer.QUASI_NEWTON_BFGS, 1E-6, 1 );
DifferentiableHigherOrderHMM hmm = new
    DifferentiableHigherOrderHMM( trainingParameterSet, new
    String[]{"A","B"} , new int[]{0,1}, new
    boolean[]{true,true},emissions, true,4.0,
  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} ));
```

The emissions and transitions of the hidden Markov model are the same as in the TrainableStatisticalModel case. However, we use training parameters for numerical optimization in this example.

A mixture DifferentiableStatisticalModel of two inhomogeneous Markov models can be created by

```
MixtureDiffSM mixDsm = new MixtureDiffSM( 3, true, mmDsm,mmDsm );
```

where the first argument is the recommended number of restarts, and the second argument indicates if plug-in parameters shall be used.

Similar to TrainableStatisticalModels, we can also define a mixture over the DNA strands with

```
StrandDiffSM strandDsm = new StrandDiffSM( mmDsm, 0.5, 1, true,
    InitMethod.INIT_BOTH_STRANDS );
```

where the first argument is the component DifferentiableStatisticalModel, the second is the a-priori probability of the forward strand, the third is the recommended number of restarts, the fourth, again, indicates if plug-in parameters are used, and the fifth indicates on which strands the component DifferentiableStatisticalModel is initialized.

We can use this StrandDiffSM together with a homogeneous Markov model as the flanking model to specify a ZOOPS DifferentiableStatisticalModel using the constructor

```
ExtendedZOOPSDiffSM zoops = new ExtendedZOOPSDiffSM(
    ExtendedZOOPSDiffSM.CONTAINS_SOMETIMES_A_MOTIF, 500, 4,
    false, hmmDsm, strandDsm, null, true );
```

Here, the first argument specifies that we want to use ZOOPS (as opposed to OOPS, where each sequence contains a motif occurrence), the second is the lenght of the input sequences, the third is the number of restarts, the fourth switches plug-in parameters, the fifth and sixth are the flanking and motif model, respectively, the seventh specifies a position distribution for the motifs (where `null` defaults to a uniform distribution), and the last indicates if the flanking model is initialized (again).

Sometimes it is useful to model different parts of a sequence by different DifferentiableStatisticalModels, for instance if we have some structural knowledge about the input sequences. To this end, Jstacs provides a class IndependentProductDiffSM, for which we can specify the models of different parts independently. We create such a IndependentProductDiffSM by calling

```
IndependentProductDiffSM ipsf = new IndependentProductDiffSM(
    4.0, true, bnDsm,mmDsm );
```

where the first parameter is the equivalent sample size, the second switches plug-in parameters, and the remaining are the DifferentiableStatisticalModels to be combined. Besides

this constructor, other constructors allow for specifying the lengths of the parts of a sequence to be modeled by the different DifferentiableStatisticalModels or the re-use of the same DifferentiableStatisticalModel for different, possibly remote, parts of an input sequence.

Since DifferentiableStatisticalModels provide methods for computing log-likelihoods, for computing normalization constants, and for determining the gradient with respect to their parameters, they can also be learned generatively by numerical optimization. For this purpose, a wrapper class exists that creates a TrainableStatisticalModel out of any DifferentiableStatisticalModel. We create such a wrapper object with

```
DifferentiableStatisticalModelWrapperTrainSM trainSm = new
    DifferentiableStatisticalModelWrapperTrainSM( mmDsm, 4,
    Optimizer.QUASI_NEWTON_BFGS, new
    SmallDifferenceOfFunctionEvaluationsCondition( 1E-6 ), 1E-6,
    1 );
trainSm.train( data[0] );
```

where we need to specify the DifferentiableStatisticalModel that shall be learned and the parameters of the numerical optimization (see section 7 for details). By this means, we can use any DifferentiableStatisticalModel in the world of TrainableStatisticalModels as well, for instance to build a TrainSMBasedClassifier(see next section).

6 Second main course: Classifiers

Classifiers allow to classify, i.e., label, previously uncharacterized data. In Jstacs, we provide the abstract class AbstractClassifier that declares three important methods besides several others.

The first method trains a classifier, i.e., it somehow adjusts to the train data:

```
public void train( DataSet... s ) throws Exception {
```

The second method classifies a given Sequence:

```
public abstract byte classify( Sequence seq ) throws Exception;
```

If we like to classify for instance the first sequence of a data set, we might use

```
System.out.println( cl.classify( data[0].getElementAt(0) ) );
```

In addition to this method, another method `classify(DataSet)` exists that performs a classification for all Sequences in a DataSet.

The third method allows for assessing the performance. Typically this is done on test data

```
public final ResultSet evaluate( PerformanceMeasureParameterSet
    params, boolean exceptionIfNotComputeable, DataSet... s )
    throws Exception {
```

where `params` is a ParameterSet of performance measures (cf. subsection 6.3), `exceptionIfNotComputeable` indicates if an exception should be thrown if a performance measure could not be computed, and `s` is an array of data sets, where dimension `i` contains data of class `i`.

The abstract sub-class AbstractScoreBasedClassifier of AbstractClassifier adds an additional method for computing a joint score for an input Sequence and a given class:

```
public double getScore( Sequence seq, int i ) throws Exception {
```

Similar to the `classify` method. For two-class problems, the method

```
public double[] getScores( DataSet s ) throws Exception {
```

allows for computing the score-differences given foreground and background class for all Sequences in the DataSet `s`. Such scores are typically the sum of the a-priori class log-score or log-probability and the score returned by `getLogScore` of SequenceScore or `getLogProb` of StatisticalModel.

Sometimes data is not split into test and train data for several diverse reasons, as for instance limited amount of data. In such cases, it is recommended to utilize some repeated procedure to split the data, train on one part and classify on the other part. In Jstacs, we

provide the abstract class ClassifierAssessment that allows to implement such procedures. In subsection 6.4, we describe how to use ClassifierAssessment and its extension.

But at first, we will focus on classifiers. Any classifier in Jstacs is an extension of the AbstractClassifier. In this section, we present on two concrete implementations, namely TrainSMBasedClassifier (cf. subsection 6.1) and GenDisMixClassifier (cf. subsection 6.2).

6.1 TrainSMBasedClassifier

The class TrainSMBasedClassifier implements a classifier on TrainableStatisticalModels, i.e., for each class the classifier holds a TrainableStatisticalModel.

If we like to build a binary classifier using PWMs for each class, we first create a PWM that is a TrainableStatisticalModel.

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

Then we can use this instance to create the classifier using

```
AbstractClassifier cl = new TrainSMBasedClassifier( pwm, pwm );
```

Thereby, we do not need to clone the PWM instance, as this is done internally for safety reasons. If we like to build a classifier that allows to distinguish between N classes, we use the same constructor but provide N TrainableStatisticalModels.

If we train a TrainSMBasedClassifier, the train method of the internally used TrainableStatisticalModels is called. For classifying a sequence, the TrainSMBasedClassifiercalls `getLogProbFor` of the internally used TrainableStatisticalModels and incorporates some class weight.

6.2 GenDisMixClassifier

The class GenDisMixClassifier implements a classifier using the unified generative-discriminative learning principle to train the internally used DifferentiableStatisticalModels. In analogy to the TrainSMBasedClassifier, the GenDisMixClassifier holds for each class a DifferentiableStatisticalModel.

If we like to build a GenDisMixClassifier, we have to provide the parameters for this classifier:

```
GenDisMixClassifierParameterSet ps = new
    GenDisMixClassifierParameterSet( alphabet, 10, (byte) 10,
    1E-6, 1E-9, 1, false, KindOfParameter.PLUGIN, true, 2 );
```

This line of code generate a ParameterSet for a GenDisMixClassifier. It states the used AlphabetContainer, the sequence length, an indicator for the numerical algorithm that is used during training, an epsilon for stopping the numerical optimization, a line epsilon for stopping the line search within the numerical optimization, a start distance for the line search, a switch that indicates whether the free or all parameter should be used, an enum that indicates the kind of class parameter initialization, a switch that indicates whether normalization should be used during optimization, and the number of threads used during numerical optimization.

If we like to build a binary classifier using PWMs for each class, we create a PWM that is a DifferentiableStatisticalModel.

```
DifferentiableStatisticalModel pwm2 = new BayesianNetworkDiffSM(
    alphabet, 10, 4.0, true, new InhomogeneousMarkov(0) );
```

Now, we are able to build a GenDisMixClassifier that uses the maximum likelihood learning principle.

```
cl = new GenDisMixClassifier(ps,
    DoesNothingLogPrior.defaultInstance, LearningPrinciple.ML,
    pwm2, pwm2 );
```

In close analogy, we can build a GenDisMixClassifier that uses the maximum conditional likelihood learning principle, if we use `LearningPrinciple.MCL`.

However, if we like to use a Bayesian learning principle we have to specify a prior that represents our prior knowledge. One of the most popular priors is the product Dirichlet prior. We can create an instance of this prior using

```
LogPrior prior = new CompositeLogPrior();
```

This class utilizes methods of DifferentiableStatisticalModel (cf. `getLogPriorTerm()` and `addGradientOfLogPriorTerm(double[], int)`) to provide the correct prior.

Given a prior, we can build a GenDisMixClassifier using for instance the maximum supervised learning principle:

```
cl = new GenDisMixClassifier(ps, prior, LearningPrinciple.MSP,
    pwm2, pwm2 );
```

Again in close analogy, we can build a GenDisMixClassifier that uses the maximum a-posteriori learning principle, if we use `LearningPrinciple.MAP`.

Alternative, we can build a GenDisMixClassifier that utilize the unified generative-discriminative learning principle. If we like to do so, we have to provide a weighting that sums to 1 and represents the weights for the conditional likelihood, the likelihood and the prior.

```
cl = new GenDisMixClassifier(ps, prior, new
    double[]{0.4,0.1,0.5}, pwm2, pwm2 );
```

6.3 Performance measures

If we like to assess the performance of any classifier, we have to use the method `evaluate` (see beginning of this section). The first argument of this method is a PerformanceMeasureParameterSet that hold the performance measures to be computed. The most simple way to create an instance is

```
PerformanceMeasureParameterSet measures =
    PerformanceMeasureParameterSet.createFilledParameters( false,
    0.999, 0.95, 0.95, 1 );
```

which yields an instance with all standard performance measures of Jstacs and specified parameters. The first argument states that all performance measures should be included. If we would change the argument to `true`, only numerical performance measures would be included an the returned instance would be a NumericalPerformanceMeasureParameterSet. The other four arguments are parameters for some performance measures.

Another way of creating a PerformanceMeasureParameterSet is to directly use performance measures extending the class AbstractPerformanceMeasure. For instance if we like to use the area under the curve (auc) for ROC and PR curve, we create

```
AbstractPerformanceMeasure[] m = {new AucROC(), new AucPR()};
```

Based on this array, we can create a PerformanceMeasureParameterSet that only contains these performance measures.

```
measures = new PerformanceMeasureParameterSet( m );
```

6.4 Assessment

If we like to assess the performance of any classifier based on an array of data sets that is not split into test and train data, we have to use some repeated procedure. In Jstacs, we provide the ClassifierAssessment that is the abstract super class of any such an procedure. We have already implemented the most widely used procedures (cf. KFoldCrossValidation and RepeatedHoldOutExperiment).

Before performing a ClassifierAssessment, we have to define a set of numerical performance measures. The performance measure have to be numerical to allow for an averaging. The most simple way to create such a set is

```
NumericalPerformanceMeasureParameterSet numMeasures =
    PerformanceMeasureParameterSet.createFilledParameters();
```

However, you can choose other measures as described in the previous subsection.

In this subsection, we exemplarily present how to perform a k-fold cross validation in Jstacs. First, we have to create an instance of KFoldCrossValidation. There several constructor to do so. Here, we use the constructor that used AbstractClassifiers.

```
ClassifierAssessment assessment = new KFoldCrossValidation( cl );
```

Second, we have to specify the parameters of the KFoldCrossValidation.

```
KFoldCrossValidationAssessParameterSet params = new
    KFoldCrossValidationAssessParameterSet(
    PartitionMethod.PARTITION_BY_NUMBER_OF_ELEMENTS,
    cl.getLength(), true, 10 );
```

These parameter are the partition method, i.e., the way how to count entries during a partitioning, the sequence length for the test data, a switch indicating whether an exception should be thrown if a performance measure could not be computed (cf. `evaluate` in AbstractClassifier), and the number of repeats k.

Now, we are able to perform a ClassifierAssessment just by calling the method `assess`.

```
System.out.println( assessment.assess( numMeasures, params, data
    ) );
```

We print the result (cf. ListResult) of this assessment to standard out. If we like to perform other ClassifierAssessments, as for instance, a RepeatedHoldOutExperiment, we have to use a specific ParameterSet (cf. KFoldCrossValidation and KFoldCrossValidationAssessParameterSet).

7 Intermediate course: Optimization

For many tasks in Jstacs, especially for numerical parameter learning, we need numerical optimization techniques.

We start the description of numerical optimization in Jstacs with the definition of a function that depends on parameters to be optimized. The most simple way to define such a function is to extend the abstract class NumericalDifferentiableFunction. In this abstract class, the gradient of the function is approximated by evaluating the function in an epsilon neightborhood around the current parameter values. Hence, a sub-class of NumericalDifferentiableFunction must only implement two methods. The first returns the number of parameters, and the second returns the function value for given parameter values.

Let us assume that we want to optimize, i.e., minimize, a function $f(x_1, x_2) = x_1^2 + x_2^2$. This functions depends on two parameters. Hence, we implement

```
public int getDimensionOfScope() {
 return 2;
}
```

And we implement the method returning the function value as

```
public double evaluateFunction( double[] x ) throws
    DimensionException, EvaluationException {
 return x[0]*x[0] + x[1]*x[1];
}
```

Now we are set to start a numerical optimization.

However, often we can derive the gradient analytically, which often yield a more efficient numerical optimization. Hence, the abstract class DifferentiableFunction allows to implement the computation of the gradient explicitly. To this end, we extend DifferentiableFunction and implement an additional method that computes the gradient:

```
public double[] evaluateGradientOfFunction( double[] x ) throws
    DimensionException, EvaluationException {
 return new double[]{2.0*x[0], 2.0*x[1]};
}
```

Now we can start a numerical optimization. To this end, we need to specify a TerminationCondition that determines when to stop the iterations of the optimization. For example, such a TerminationCondition may stop the optimization, if the difference of successive function evaluations does not exceed a given threshold:

```
AbstractTerminationCondition tc = new
    SmallDifferenceOfFunctionEvaluationsCondition( 1E-6 );
```

In this example, this threshold is set to 10^{-6}. Another option would be to stop after 100 iterations or if the gradient becomes small:

```
AbstractTerminationCondition tc2 = new IterationCondition(100);
AbstractTerminationCondition tc3 = new SmallGradientConditon(
    1E-6 );
```

And we may also combine several TerminationCondition in a CombinedCondition:

```
TerminationCondition combined = new CombinedCondition( 2, tc,
    tc3 );
```

The first parameter (2) specifies that the CombinedCondition only allows to continue to the next iteration if both of the supplied TerminationConditions do so.

For the numerical optimization, we create initial parameters and start the optimization:

```
double[] parameters = new double[df.getDimensionOfScope()];
Optimizer.optimize( Optimizer.QUASI_NEWTON_BFGS, df, parameters,
    combined, 1E-6, new ConstantStartDistance( 1E-4 ), System.out
    );
```

The arguments of this static method have the following meaning: The first argument defines the technique for the optimization. In the example, we set this to the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno. As an alternative, Jstacs offers other quasi-Newton method including limited-memory variants, different conjugate gradients approaches, and steepest descent. The second argument is the DifferentiableFunction, the third are the initial parameter values, the fourth is the TerminationCondition, and the fifth is the threshold on the difference of function values during the line search. Jstacs uses Brent's method, which is a combination of quadratic interpolation and golden ratio, for the line search. The sixth argument is the initial step size during the line search, and the last argument is an `OutputStream` to which output of the optimization is written. Here, we specify `System.out`. If this argument is `null`, output is suppressed.

After the optimization has finished, the supplied parameter array (`parameters` in the example) contains the optimal parameter values.

8 Dessert: Alignments, Utils, and goodies

In this section, we present a motley composition of interesting classes of Jstacs.

8.1 Alignments

In this subsection, we present how to compute Alignments using Jstacs.

If we like to compute an alignment, we first have to define the costs for match, mismatch, and gaps. In Jstacs, we provide the interface Costs that declares all necessary method used during the alignment. In this example, we restrict to simple costs that are 0 for match, 1 for match, 0.5 for a gap.

```
Costs costs = new SimpleCosts( 0, 1, 0.5 );
```

Second, we have to provide an instance of Alignment. This instance contains all information needed for an alignment and stores for instance matrices used for dynamic programming. When creating an instance, we have to specify which kind of alignment we like to have. Jstacs supports local, global and semi-global alignments (cf. Alignment.AlignmentType).

```
Alignment align = new Alignment( AlignmentType.GLOBAL, costs );
```

In second constructor it is also possible to specify the number of off-diagonals to be used in the alignment leading to a potential speedup.

Finally, we can compute the optimal alignment between two Sequences and write the result to the standard output.

```
System.out.println( align.getAlignment( seq1, seq2 ) );
```

The alignment instance can be reused for aligning further sequences.

In Jstacs, we also provide the possibility of computing optimal alignments with affine gap costs. For this reason, we implement the class AffineCosts that is used to specify the cost for a gap opening. The costs for gap elongation are given by the gap costs of the internally used instance of Costs.

```
costs = new AffineCosts( 1, costs );
align = new Alignment( AlignmentType.GLOBAL, costs );
System.out.println( align.getAlignment( seq1, seq2 ) );
```

8.2 REnvironment: Connection to R

In this subsection, we show how to access R (cf. http://www.r-project.org/) from Jstacs. R is a project for statistical computing that allows for performing complex computations and creating nice plots.

In some cases, it is reasonable to use R from within Jstacs. To do so, we have to create a connection to R. We utilize the package `Rserve` (cf. http://www.rforge.net/Rserve/) of R that allows to communicate between Java and R. Having a running instance of `Rserve`, we can create a connection via

```
REnvironment re = new REnvironment();
```

However, in some cases we have to specify the login name, a password, and the port for the communication which is possible via alternative constructors.

Now, we are able to do diverse things in R. Here, we only present three methods, but REnvironment provides more functionality. First, we copy an array of `double`s from Java to R

```
re.createVector( "values", values );
```

and second, we modify it

```
re.voidEval( "values=rnorm(length(values));" );
```

Finally, the REnvironment allows to create plots as PDF, TeX, or `BufferedImage`.

```
re.plotToPDF( "plot(values,t=\"l\");", "values.pdf", true );
```

8.3 ArrayHandler: Handling arrays

In this subsection, we present a way to easily handle arrays in Java, i.e., to cast, clone, and create arrays with elements of generic type. To this end, we implement the class ArrayHandler in Jstacs.

Let's assume we have a two dimensional array of either primitives of some Java class and we like to create a deep clone as it is necessary for member fields in clone methods.

```
double[][] twodim = new double[5][5];
```

Traditionally, we would have to implement `for`-loops to do so. However, the ArrayHandler implements this functionality in a generic manner providing one method for this purpose.

```
double[][] clone = ArrayHandler.clone( twodim );
```

A second use case, is the creation of arrays, where each and every entry is a clone of some instance.

```
TrainableStatisticalModel pwm =
    TrainableStatisticalModelFactory.createPWM(
    DNAAlphabetContainer.SINGLETON, 10, 4.0 );
TrainableStatisticalModel[] models = ArrayHandler.createArrayOf(
    pwm, 10 );
```

The third use case is to cast an array. Even if all elements of the array are from the same class, the component type of the array might be different (some super class). A simple cast will fail in those cases. However, the ArrayHandler provides two methods for casting arrays. Here, we present the more important method, which allows to specify the array component type and performs the cast operation.

```
Object[] m = new Object[]{
     TrainableStatisticalModelFactory.createPWM(
        DNAAlphabetContainer.SINGLETON, 10, 4.0 ),
     TrainableStatisticalModelFactory.createHomogeneousMarkovModel(
        DNAAlphabetContainer.SINGLETON, 40.0, (byte)0 )
};
TrainableStatisticalModel[] sms = ArrayHandler.cast(
    TrainableStatisticalModel.class, models );
```

8.4 ToolBox

The class ToolBox contains several static methods for recurring tasks. For example, you can compute the maximum of an array of `double`s

```
double max = ToolBox.max( values );
```

or the sum of the values

```
double sum = ToolBox.sum( values );
```

or you can obtain the index of the first maximum value in the provided array

```
int maxIndex = ToolBox.getMaxIndex( values );
```

8.5 Normalisation

Another frequently needed functionality is the handling of log-values. Assume that we have an array `values` containing a number of log-probabilities l_i. What we want to compute is the logarithm of the sum of the original probabilities, i.e., $\log(\sum_i \exp(l_i))$. The naive computation of this sum often results in numerical problems, especially, if the original probabilities are very different. A more exact solution is provided by the static method `getLogSum` of the class Normalisation, which can be accessed by calling

```
double logSum = Normalisation.getLogSum( values );
```

Of course, this method does not only work for probabilities, but for general log-values.

Sometimes, we also want to normalize the given probabilities. That means, given the log-probabilities l_i, we want to obtain normalized probabilities $p_i = \frac{\exp(l_i)}{\sum_j \exp(l_j)}$. This normalization is performed by calling

```
Normalisation.logSumNormalisation( values );
```

and after the call, the array `values` contains the normalized probabilities (not log-probabilities!).

Finally, we might want to do the same for probabilities, i.e. given probabilities q_i in an array `values`, we want to compute $p_i = \frac{q_i}{\sum_j q_j}$ using

```
Normalisation.sumNormalisation( values );
```

A typical application for the last two methods are (log) joint probabilities that we want use to compute conditional probabilities by dividing by a marginal probability.

8.6 Goodies

The class SafeOutputStream is a simple way to switch between writing outputs of a program to standard out, to a file, or to completely suppress output. This class is basically a wrapper for other output streams that can handle `null` values. You can create a SafeOutputStream writing to standard out with

```
OutputStream stream = SafeOutputStream.getSafeOutputStream(
    System.out );
```

If you provided `null` to the factory method instead, output would be suppressed, while no modifications in code using this SafeOutputStream would be necessary.

Finally, the class SubclassFinder can be used to search for all subclasses of a given class in a specified package and its sub-packages. For example, if we want to find all concrete sub-classes of TrainableStatisticalModel, i.e., classes that are not abstract and can be instantiated, in all sub-packages of `de.jstacs`, we call

```
LinkedList<Class<? extends TrainableStatisticalModel>> list =
    SubclassFinder.findInstantiableSubclasses(
    TrainableStatisticalModel.class, "de.jstacs" );
```

and obtain a linked list containing all such classes. Other methods in SubclassFinder allow for searching for general sub-types including interfaces and abstract classes, or for filtering the results by further required interfaces.

9 Recipes

In this section, we show some more complex code examples. All these code examples can be downloaded as a zip file and may serve as a starting points for your own applications.

9.1 Creation of user-specfic alphabet

In this example, we create a new ComplementableDiscreteAlphabet using the generic implementation. We then use this Alphabet to create a Sequence and compute its reverse complement.

```
String[] symbols = {"A", "C", "G", "T", "-"};
//new alphabet
DiscreteAlphabet abc = new DiscreteAlphabet( true, symbols );

//new alphabet that allows to build the reverse complement of a sequence
int[] revComp = new int[symbols.length];
revComp[0] = 3; //symbols[0]^rc = symbols[3]
revComp[1] = 2; //symbols[1]^rc = symbols[2]
revComp[2] = 1; //symbols[2]^rc = symbols[1]
revComp[3] = 0; //symbols[3]^rc = symbols[0]
revComp[4] = 4; //symbols[4]^rc = symbols[4]
GenericComplementableDiscreteAlphabet abc2 = new
    GenericComplementableDiscreteAlphabet( true, symbols, revComp );

Sequence seq = Sequence.create( new AlphabetContainer( abc2 ), "ACGT-" );
Sequence rc = seq.reverseComplement();
System.out.println( seq );
System.out.println( rc );
```

9.2 Learning a position weight matrix from data

In this example, we show how to load sequence data into Jstacs and how to learn a position weight matrix (inhomogeneous Markov model of order 0) on these data.

```
//read data from FastA file
DNADataSet ds = new DNADataSet( args[0] );
AlphabetContainer con = ds.getAlphabetContainer();
//create position weight matrix model
TrainableStatisticalModel pwm =
    TrainableStatisticalModelFactory.createPWM( con,
    ds.getElementLength(), 4 );
//train it on the input data
pwm.train( ds );
//print the trained model
System.out.println(pwm);
```

9.3 Learning a homogeneous Markov model from data

In this example, we show how to load sequence data into Jstacs and how to learn a homogeneous Markov model of order 1 on these data.

```
//read data from FastA file
DNADataSet ds = new DNADataSet( args[0] );
//create homogeneous Markov model of order 1
TrainableStatisticalModel hmm =
    TrainableStatisticalModelFactory.createHomogeneousMarkovModel(
    ds.getAlphabetContainer(), 4, (byte)1 );
//train it on the input data
hmm.train( ds );
//print the trained model
System.out.println(hmm);
```

9.4 Generating data from a homogeneous Markov model

In this example, we show how to learn a homogeneous Markov model of order 2 from data (similar to the previous example), and use the learned model to generate new data following the same distribution as the original data.

```
//read data from FastA file
DNADataSet ds = new DNADataSet( args[0] );
//create homogeneous Markov model of order 2
TrainableStatisticalModel hmm =
    TrainableStatisticalModelFactory.createHomogeneousMarkovModel(
    ds.getAlphabetContainer(), 4, (byte)2 );
//train it on the input data
hmm.train( ds );

//generate 100 sequences of length 20
DataSet generated = hmm.emitDataSet( 100, 20 );
//print these data
System.out.println(generated);
//and save them to a plain text file
generated.save( new File(args[1]) );
```

9.5 Learning a mixture model from data

In this example, we show how to load sequence data into Jstacs and how to learn a mixture model of two position weight matrices on these data using the expectation maximization algorithm.

```
//read data from FastA file
DNADataSet ds = new DNADataSet( args[0] );
//create position weight matrix model
```

```
TrainableStatisticalModel pwm =
    TrainableStatisticalModelFactory.createPWM(
    ds.getAlphabetContainer(), ds.getElementLength(), 4 );
//create mixture model of two position weight matrices
TrainableStatisticalModel mixture =
    TrainableStatisticalModelFactory.createMixtureModel( new
    double[]{4,4}, new TrainableStatisticalModel[]{pwm,pwm} );
//train it on the input data using EM
mixture.train( ds );
//print the trained model
System.out.println(mixture);
```

9.6 Analysing data with different models

In this example, we show how to use the TrainableStatisticalModelFactory to create inhomogeneous and homogeneous Markov models, and Bayesian trees, and how to learn these models on a common data set.

```
//read data from FastA file
DNADataSet ds = new DNADataSet( args[0] );

//get alphabet, length from data
AlphabetContainer alphabet = ds.getAlphabetContainer();
int length = ds.getElementLength();
//set ESS used for all models
double ess = 4;

TrainableStatisticalModel[] models = new TrainableStatisticalModel[4];
//create position weight matrix
models[0] = TrainableStatisticalModelFactory.createPWM( alphabet,
    length, 4 );
//create inhomogeneous Markov model of order 1 (WAM)
models[1] =
    TrainableStatisticalModelFactory.createInhomogeneousMarkovModel(
    alphabet, length, ess, (byte)1 );
//create Bayesian tree
models[2] = TrainableStatisticalModelFactory.createBayesianNetworkModel(
    alphabet, length, ess, (byte)1 );
//create homogeneous Markov model of order 2
models[3] =
    TrainableStatisticalModelFactory.createHomogeneousMarkovModel(
    alphabet, ess, (byte)2 );

//train and print all models
for(int i=0;i<models.length;i++){
 models[i].train( ds );
 System.out.println(models[i]);
}
```

9.7 De-novo motif discovery with a sunflower hidden Markov model)

In this example, we show how to use the HMMFactory to create a sunflower hidden Markov model (HMM) with two motifs of different lengths. We show how to train the sunflower HMM on input data, which are typically long sequences containing an over-represented motif. After training the HMM, we show how to compute and output the Viterbi paths for all sequences, which give an indication of the position of motif occurrences.

```
//load data
DataSet data = new DNADataSet(args[0]);
//define parameters of Baum-Welch training using all available processor
    cores
BaumWelchParameterSet pars = new BaumWelchParameterSet(10, new
    SmallDifferenceOfFunctionEvaluationsCondition(1E-6),
    AbstractMultiThreadedOptimizableFunction.getNumberOfAvailableProcessors());
//create sunflower HMM with motifs of length 8 and 12
AbstractHMM hmm = HMMFactory.createSunflowerHMM(pars,
    data.getAlphabetContainer(), 0, data.getElementLength(), true, 8,12);
//train the HMM using Baum-Welch
hmm.train(data);
//print the trained HMM
System.out.println(hmm);
//print Viterbi paths of all sequences
for(int i=0;i<data.getNumberOfElements();i++){
 Pair<IntList,Double> p = hmm.getViterbiPathFor(data.getElementAt(i));
 System.out.println(p.getSecondElement()+"\t"+p.getFirstElement());
}
```

9.8 Learning a classifier using the generative maximum a-posteriori principle

In this example, we show how to train a classifier based on a position weight matrix model and a homogeneous Markov model on training data, and how to use the trained classifier to classify sequences.

```
//read data from FastA files
DataSet[] data = new DataSet[2];
data[0] = new DNADataSet( args[0] );
data[1] = new DNADataSet( args[1] );

//create position weight matrix model
TrainableStatisticalModel pwm =
    TrainableStatisticalModelFactory.createPWM(
    data[0].getAlphabetContainer(), data[0].getElementLength(), 4 );
//create homogeneous Markov model of order 1
TrainableStatisticalModel hmm =
    TrainableStatisticalModelFactory.createHomogeneousMarkovModel(
    data[1].getAlphabetContainer(), 4, (byte)1 );
//build a classifier using these models
TrainSMBasedClassifier cl = new TrainSMBasedClassifier( pwm, hmm );
```

```
//train it on the training data
cl.train( data );

//print the trained classifier
System.out.println(cl);
//classify one of the sequences
Sequence seq = data[0].getElementAt( 0 );
byte res = cl.classify( seq );
//print sequence and classification result
System.out.println(seq+" -> "+res);

//evaluate
NumericalPerformanceMeasureParameterSet params =
    PerformanceMeasureParameterSet.createFilledParameters();
System.out.println( cl.evaluate(params, true, data) );
```

9.9 Learning a classifier using the discriminative maximum supervised posterior principle

In this example, we show how to use the DifferentiableStatisticalModelFactory to create a position weight matrix and how to learn a classifier based on two position weight matrices using the discriminative maximum supervised posterior principle.

```
//read data from FastA files
DataSet[] data = new DataSet[2];
data[0] = new DNADataSet( args[0] );
data[1] = new DNADataSet( args[1] );
AlphabetContainer con = data[0].getAlphabetContainer();

//define differentiable PWM model
DifferentiableStatisticalModel pwm =
    DifferentiableStatisticalModelFactory.createPWM(con, 10, 4);
//parameters for numerical optimization
GenDisMixClassifierParameterSet pars = new
    GenDisMixClassifierParameterSet(con,10,(byte)10,1E-9,1E-10,1,
    false,KindOfParameter.PLUGIN,true,1);
//define and train classifier
AbstractClassifier cl = new MSPClassifier( pars, pwm, pwm );
cl.train( data );

System.out.println(cl);
```

9.10 Creating Data sets

In this example, we show different ways of creating a DataSet in Jstacs from plain text and FastA files and using the adaptor to BioJava.

```
String home = args[0]+File.separator;

//load DNA sequences in FastA-format
DataSet data = new DNADataSet( home+"myfile.fa" );

//create a DNA-alphabet
AlphabetContainer container = DNAAlphabetContainer.SINGLETON;

//create a DataSet using the alphabet from above in FastA-format
data = new DataSet( container, new SparseStringExtractor(
    home+"myfile.fa", StringExtractor.FASTA ));

//create a DataSet using the alphabet from above
data = new DataSet( container, new SparseStringExtractor(
    home+"myfile.txt" ));

//defining the ids, we want to obtain from NCBI Genbank:
GenbankSequenceDB db = new GenbankSequenceDB();

//at the moment the following fails due to a problem in BioJava
    hopefully fixed in the next legacy release
//this may fail if BioJava fails to load the sequence, e.g. if you are
    not connected to the internet
/*SimpleSequenceIterator it = new SimpleSequenceIterator(
  db.getSequence( "NC_001284.2" ),
  db.getSequence( "NC_000932.1" )
  );
 */

RichSequenceIterator it = IOTools.readGenbankDNA( new BufferedReader(
    new FileReader( home+"example.gb" ) ), null );

//conversion to Jstacs DataSet
data = BioJavaAdapter.sequenceIteratorToDataSet( it, null );
System.out.println(data);
```

9.11 Using TrainSMBasedClassifier

In this example, we show how to create a TrainSMBasedClassifier using to position weight matrices, train this classifier, classify previously unlabeled data, store the classifier to its XML representation, and load it back into Jstacs.

```
String home = args[0];

//create a DataSet for each class from the input data, using the DNA
    alphabet
DataSet[] data = new DataSet[2];
data[0] = new DNADataSet( args[1] );

//the length of our input sequences
int length = data[0].getElementLength();
```

```
data[1] = new DataSet( new DNADataSet( args[2] ), length );

//create a new PWM
BayesianNetworkTrainSM pwm = new BayesianNetworkTrainSM( new
    BayesianNetworkTrainSMParameterSet(
  //the alphabet and the length of the model:
  data[0].getAlphabetContainer(), length,
  //the equivalent sample size to compute hyper-parameters
  4,
  //some identifier for the model
  "my␣PWM",
  //we want a PWM, which is an inhomogeneous Markov model (IMM) of order
     0
  ModelType.IMM, (byte) 0,
  //we want to estimate the MAP-parameters
  LearningType.ML_OR_MAP ) );

//create a new classifier
TrainSMBasedClassifier classifier = new TrainSMBasedClassifier( pwm, pwm
    );
//train the classifier
classifier.train( data );
//sequences that will be classified
DataSet toClassify = new DNADataSet( args[3] );

//use the trained classifier to classify new sequences
byte[] result = classifier.classify( toClassify );
System.out.println( Arrays.toString( result ) );

//create the XML-representation of the classifier
StringBuffer buf = new StringBuffer();
XMLParser.appendObjectWithTags( buf, classifier, "classifier" );

//write it to disk
FileManager.writeFile( new File(home+"myClassifier.xml"), buf );

//read XML-representation from disk
StringBuffer buf2 = FileManager.readFile( new
    File(home+"myClassifier.xml") );

//create new classifier from read StringBuffer containing XML-code
AbstractClassifier trainedClassifier = (AbstractClassifier)
    XMLParser.extractObjectForTags(buf2, "classifier");
```

9.12 Using GenDisMixClassifier

In this example, we show how to create GenDisMixClassifiers using two position weight matrices. We show how GenDisMixClassifiers can be created for all basic learning principles (ML, MAP, MCL, MSP), and how these classifiers can be trained and assessed.

```
//read FastA-files
DataSet[] data = {
```

```
        new DNADataSet( args[0] ),
        new DNADataSet( args[1] )
};
AlphabetContainer container = data[0].getAlphabetContainer();
int length = data[0].getElementLength();

//equivalent sample size =^= ESS
double essFg = 4, essBg = 4;
//create DifferentiableSequenceScore, here PWM
DifferentiableStatisticalModel pwmFg = new BayesianNetworkDiffSM(
    container, length, essFg, true, new InhomogeneousMarkov(0) );
DifferentiableStatisticalModel pwmBg = new BayesianNetworkDiffSM(
    container, length, essBg, true, new InhomogeneousMarkov(0) );

//create parameters of the classifier
GenDisMixClassifierParameterSet cps = new
    GenDisMixClassifierParameterSet(
  container,//the used alphabets
  length,//sequence length that can be modeled/classified
  Optimizer.QUASI_NEWTON_BFGS, 1E-1, 1E-1, 1,//optimization parameter
  false,//use free parameters or all
  KindOfParameter.PLUGIN,//how to start the numerical optimization
  true,//use a normalized objective function
  AbstractMultiThreadedOptimizableFunction.getNumberOfAvailableProcessors()//number
      of compute threads
);

//create classifiers
LearningPrinciple[] lp = LearningPrinciple.values();
GenDisMixClassifier[] cl = new GenDisMixClassifier[lp.length+1];
//elementary learning principles
int i = 0;
for( ; i < cl.length-1; i++ ){
 System.out.println( "classifier␣" + i + "␣uses␣" + lp[i] );
 cl[i] = new GenDisMixClassifier( cps, new CompositeLogPrior(), lp[i],
     pwmFg, pwmBg );
}

//use some weighted version of log conditional likelihood, log
    likelihood, and log prior
double[] beta = {0.3,0.3,0.4};
System.out.println( "classifier␣" + i + "␣uses␣the␣weights␣" +
    Arrays.toString( beta ) );
cl[i] = new GenDisMixClassifier( cps, new CompositeLogPrior(), beta,
    pwmFg, pwmBg );

//do what ever you like

//e.g., train
for( i = 0; i < cl.length; i++ ){
 cl[i].train( data );
}

//e.g., evaluate (normally done on a test data set)
```

```
PerformanceMeasureParameterSet mp =
    PerformanceMeasureParameterSet.createFilledParameters();
for( i = 0; i < cl.length; i++ ){
 System.out.println( cl[i].evaluate( mp, true, data ) );
}
```

9.13 Accessing R from Jstacs

Here, we show a number of examples how R can be used from within Jstacs using RServe.

```
REnvironment e = null;
try {
 //create a connection to R with YOUR server name, login and password
 e = new REnvironment();//might be adjusted

 System.out.println( "java:␣" + System.getProperty( "java.version" ) );
 System.out.println();
 System.out.println( e.getVersionInformation() );

 // compute something in R
 REXP erg = e.eval( "sin(10)" );
 System.out.println( erg.asDouble() );

 //create a histrgram in R in 3 steps
 //1) create the data
 e.voidEval( "a␣=␣100;" );
 e.voidEval( "n␣=␣rnorm(a)" );
 //2) create the plot command
 String plotCmd = "hist(n,breaks=a/5)";
 //3a) plot as pdf
 e.plotToPDF( plotCmd, args[0]+"/test.pdf", true );
 //or
 //3b) create an image and show it
 BufferedImage i = e.plot( plotCmd, 640, 480 );
 REnvironment.showImage( "histogramm", i, JFrame.EXIT_ON_CLOSE );

} catch ( Exception ex ) {
 ex.printStackTrace();
} finally {
 if( e != null ) {
  try {
   //close REnvironment correctly
   e.close();
  } catch ( Exception e1 ) {
   System.err.println( "could␣not␣close␣REnvironment." );
   e1.printStackTrace();
  }
 }
}
```

9.14 Getting ROC and PR curve from a classifier

In this example, we show how a classifier (loaded from disk) can be assessed on test data, and how we can plot ROC and PR curves of this classifier and test data set.

```
public static void main(String[] args) throws Exception {
 //read XML-representation from disk
 StringBuffer buf2 = FileManager.readFile( new
     File(args[0]+"myClassifier.xml") );

 //create new classifier from read StringBuffer containing XML-code
 AbstractClassifier trainedClassifier = (AbstractClassifier)
     XMLParser.extractObjectForTags(buf2, "classifier");

 //create a DataSet for each class from the input data, using the DNA
     alphabet
 DataSet[] test = new DataSet[2];
 test[0] = new DNADataSet( args[1] );

 //the length of our input sequences
 int length = test[0].getElementLength();

 test[1] = new DataSet( new DNADataSet( args[2] ), length );


 AbstractPerformanceMeasure[] m = { new PRCurve(), new ROCCurve() };
 PerformanceMeasureParameterSet mp = new PerformanceMeasureParameterSet(
     m );
 ResultSet rs = trainedClassifier.evaluate( mp, true, test );

 REnvironment r = null;
 try {
  r = new REnvironment();//might be adjusted
  for( int i = 0; i < rs.getNumberOfResults(); i++ )  {
   Result res = rs.getResultAt(i);
   if( res instanceof DoubleTableResult ) {
    DoubleTableResult dtr = (DoubleTableResult) res;
    ImageResult ir = DoubleTableResult.plot( r, dtr );
    REnvironment.showImage( dtr.getName(), ir.getValue() );
   } else {
    System.out.println( res );
   }
  }
 } catch( Exception e ) {
  e.printStackTrace();
 } finally {
  if( r != null ) {
   r.close();
  }
```

9.15 Performing crossvalidation

In this example, we show how we can compare classifiers built on different types of models and using different learning principles in a cross validation. Specifically, we create a position weight matrix, use that matrix to create a mixture model, and we create an inhomogeneous Markov model of order 3. We do so in the world of TrainableStatisticalModels and in the world of DifferentiableStatisticalModels. We then use the mixture model as foreground model and the inhomogeneous Markov model as the background model when building classifiers. The classifiers are learned by the generative MAP principle and the discriminative MSP principle, respectively. We then assess these classifiers in a 10-fold cross validation.

```
//create a DataSet for each class from the input data, using the DNA
    alphabet
DataSet[] data = new DataSet[2];
data[0] = new DNADataSet( args[0] );

//the length of our input sequences
int length = data[0].getElementLength();

data[1] = new DataSet( new DNADataSet( args[1] ), length );

AlphabetContainer container = data[0].getAlphabetContainer();

//create a new PWM
BayesianNetworkTrainSM pwm = new BayesianNetworkTrainSM( new
    BayesianNetworkTrainSMParameterSet(
  //the alphabet and the length of the model:
  container, length,
  //the equivalent sample size to compute hyper-parameters
  4,
  //some identifier for the model
  "my␣PWM",
  //we want a PWM, which is an inhomogeneous Markov model (IMM) of order
     0
  ModelType.IMM, (byte) 0,
  //we want to estimate the MAP-parameters
  LearningType.ML_OR_MAP ) );

//create a new mixture model using 2 PWMs
MixtureTrainSM mixPwms = new MixtureTrainSM(
  //the length of the mixture model
  length,
  //the two components, which are PWMs
  new TrainableStatisticalModel[]{pwm,pwm},
  //the number of starts of the EM
  10,
  //the equivalent sample sizes
  new double[]{pwm.getESS(),pwm.getESS()},
  //the hyper-parameters to draw the initial sequence-specific component
     weights (hidden variables)
  1,
```

```
  //stopping criterion
  new SmallDifferenceOfFunctionEvaluationsCondition(1E-6),
  //parameterization of the model, LAMBDA complies with the
  //parameterization by log-probabilities
  Parameterization.LAMBDA);

//create a new inhomogeneous Markov model of order 3
BayesianNetworkTrainSM mm = new BayesianNetworkTrainSM(
  new BayesianNetworkTrainSMParameterSet( container, length, 256, "my␣
      iMM(3)", ModelType.IMM, (byte) 3, LearningType.ML_OR_MAP ) );

//create a new PWM scoring function
BayesianNetworkDiffSM dPwm = new BayesianNetworkDiffSM(
  //the alphabet and the length of the scoring function
  container, length,
  //the equivalent sample size for the plug-in parameters
  4,
  //we use plug-in parameters
  true,
  //a PWM is an inhomogeneous Markov model of order 0
  new InhomogeneousMarkov(0));

//create a new mixture scoring function
MixtureDiffSM dMixPwms = new MixtureDiffSM(
  //the number of starts
  2,
  //we use plug-in parameters
  true,
  //the two components, which are PWMs
  dPwm,dPwm);

//create a new scoring function that is an inhomogeneous Markov model of
    order 3
BayesianNetworkDiffSM dMm = new BayesianNetworkDiffSM(container, length,
    4, true, new InhomogeneousMarkov(3));

//create the classifiers
int threads =
    AbstractMultiThreadedOptimizableFunction.getNumberOfAvailableProcessors();
AbstractScoreBasedClassifier[] classifiers = new
    AbstractScoreBasedClassifier[]{
          //model based with mixture model and Markov model
          new TrainSMBasedClassifier( mixPwms, mm ),
          //conditional likelihood based classifier
          new MSPClassifier( new
              GenDisMixClassifierParameterSet(container, length,
            //method for optimizing the conditional likelihood and
            //other parameters of the numerical optimization
            Optimizer.QUASI_NEWTON_BFGS, 1E-2, 1E-2, 1, true,
                KindOfParameter.PLUGIN, false, threads),
            //mixture scoring function and Markov model scoring function
            dMixPwms,dMm )
};
```

```
//create an new k-fold cross validation using above classifiers
KFoldCrossValidation cv = new KFoldCrossValidation( classifiers );

//we use a specificity of 0.999 to compute the sensitivity and a
    sensitivity of 0.95 to compute FPR and PPV
NumericalPerformanceMeasureParameterSet mp =
    PerformanceMeasureParameterSet.createFilledParameters();
//we do a 10-fold cross validation and partition the data by means of
    the number of symbols
KFoldCrossValidationAssessParameterSet cvpars = new
    KFoldCrossValidationAssessParameterSet(PartitionMethod.PARTITION_BY_NUMBER_OF_SYMB
    length, true, 10);

//compute the result of the cross validation and print them to System.out
System.out.println( cv.assess( mp, cvpars, data ) );
```

9.16 Implementing a TrainableStatisticalModel

In this example, we show how to implement a new TrainableStatisticalModel. Here, we implement a simple homogeneous Markov models of order 0 to focus on the technical side of the implementation. A homogeneous Markov model of order 0 has parameters θ_a where a is a symbol of the alphabet Σ and $\sum_{a\in\Sigma}\theta_a = 1$. For an input sequence $\mathbf{x} = x_1, \ldots, x_L$ it models the likelihood

$$P(\mathbf{x}|\boldsymbol{\theta}) = \prod_{l=1}^{L} \theta_{x_l}.$$

In the implementation, we use log-parameters $\log \theta_a$.

```
public class HomogeneousMarkovModel extends
    AbstractTrainableStatisticalModel {

 private double[] logProbs;//array for the parameters, i.e. the
    probabilities for each symbol

 public HomogeneousMarkovModel( AlphabetContainer alphabets ) throws
    Exception {
  super( alphabets, 0 ); //we have a homogeneous
    TrainableStatisticalModel, hence the length is set to 0
  //a homogeneous TrainableStatisticalModel can only handle simple
    alphabets
  if(! (alphabets.isSimple() && alphabets.isDiscrete()) ){
   throw new Exception("Only␣simple␣and␣discrete␣alphabets␣allowed");
  }
  //initialize parameter array
  this.logProbs = new double[(int) alphabets.getAlphabetLengthAt( 0 )];
  Arrays.fill( logProbs, -Math.log(logProbs.length) );
 }

 public HomogeneousMarkovModel( StringBuffer stringBuff ) throws
    NonParsableException {
```

```
        super( stringBuff );
    }

protected void fromXML( StringBuffer xml ) throws NonParsableException {
 //extract our XML-code
 xml = XMLParser.extractForTag( xml, "homogeneousMarkovModel" );
 //extract all the variables using XMLParser
 alphabets = (AlphabetContainer) XMLParser.extractObjectForTags( xml,
     "alphabets" );
 length = XMLParser.extractObjectForTags( xml, "length", int.class );
 logProbs = XMLParser.extractObjectForTags( xml, "logProbs",
     double[].class );
}

public StringBuffer toXML() {
 StringBuffer buf = new StringBuffer();
 //pack all the variables using XMLParser
 XMLParser.appendObjectWithTags( buf, alphabets, "alphabets" );
 XMLParser.appendObjectWithTags( buf, length, "length" );
 XMLParser.appendObjectWithTags( buf, logProbs, "logProbs" );
 //add our own tag
 XMLParser.addTags( buf, "homogeneousMarkovModel" );
 return buf;
}

public String getInstanceName() {
            return "Homogeneous Markov model of order 0";
        }

public double getLogPriorTerm() throws Exception {
            //we use ML-estimation, hence no prior term
            return 0;
        }

public NumericalResultSet getNumericalCharacteristics() throws Exception {
 //we do not have much to tell here
 return new NumericalResultSet(new NumericalResult("Number of
     parameters","The number of parameters this model
     uses",logProbs.length));
}

public double getLogProbFor( Sequence sequence, int startpos, int endpos
    ) throws NotTrainedException, Exception {
 double seqLogProb = 0.0;
 //compute the log-probability of the sequence between startpos and
     endpos (inclusive)
 //as sum of the single symbol log-probabilities
 for(int i=startpos;i<=endpos;i++){
  //directly access the array by the numerical representation of the
      symbols
  seqLogProb += logProbs[sequence.discreteVal( i )];
 }
 return seqLogProb;
}
```

```
 public boolean isInitialized() {
        return true;
    }

 public void train( DataSet data, double[] weights ) throws Exception {
  //reset the parameter array
  Arrays.fill( logProbs, 0.0 );
  //default sequence weight
  double w = 1;
  //for each sequence in the data set
  for(int i=0;i<data.getNumberOfElements();i++){
   //retrieve sequence
   Sequence seq = data.getElementAt( i );
   //if we do have any weights, use them
   if(weights != null){
    w = weights[i];
   }
   //for each position in the sequence
   for(int j=0;j<seq.getLength();j++){
    //count symbols, weighted by weights
    logProbs[ seq.discreteVal( j ) ] += w;
   }
  }
  //compute normalization
  double norm = 0.0;
  for(int i=0;i<logProbs.length;i++){ norm += logProbs[i]; }
  //normalize probs to obtain proper probabilities
  for(int i=0;i<logProbs.length;i++){ logProbs[i] = Math.log(
      logProbs[i]/norm ); }
 }
}
```

9.17 Implementing a DifferentiableStatisticalModel

In this example, we show how to implement a new DifferentiableStatisticalModel. Here, we implement a simple position weight matrix, i.e., an inhomogeneous Markov model of order 0. Since we want to use this position weight matrix in numerical optimization, we parameterize it in the so called "natural parameterization", where the probability of symbol a at position l is $P(X_l = a|\boldsymbol{\lambda}) = \frac{\exp(\lambda_{l,a})}{\sum_{\tilde{a}} \exp(\lambda_{l,\tilde{a}})}$. Since we use a product-Dirichlet prior on the parameters, we transformed this prior to the parameterization we use.

Here, the method `getLogScore` returns a log-score that can be normalized to a proper log-likelihood by subtracting a log-normalization constant. The log-score for an input sequence $\mathbf{x} = x_1, \ldots, x_L$ essentially is

$$S(\mathbf{x}|\boldsymbol{\lambda}) = \sum_{l=1}^{L} \lambda_{l,x_l}.$$

The normalization constant is a partition function, i.e., the sum of the scores over all possible input sequences:

$$\begin{aligned} Z(\boldsymbol{\lambda}) &= \sum_{\mathbf{x} \in \Sigma^L} \exp(S(\mathbf{x}|\boldsymbol{\lambda})) \\ &= \sum_{\mathbf{x} \in \Sigma^L} \prod_{l=1}^{L} \exp(\lambda_{l,x_l}) \\ &= \prod_{l=1}^{L} \sum_{a \in \Sigma} \exp(\lambda_{l,a}) \end{aligned}$$

Thus, the likelihood is defined as

$$P(\mathbf{x}|\lambda) = \frac{\exp(S(\mathbf{x}|\boldsymbol{\lambda}))}{Z(\boldsymbol{\lambda})}$$

and

$$\log P(\mathbf{x}|\lambda) = S(\mathbf{x}|\boldsymbol{\lambda})) - \log Z(\boldsymbol{\lambda}).$$

```
public class PositionWeightMatrixDiffSM extends
    AbstractDifferentiableStatisticalModel {

 private double[][] parameters;// array for the parameters of the PWM in
     natural parameterization
 private double ess;// the equivalent sample size
 private boolean isInitialized;// if the parameters of this PWM are
     initialized
 private Double norm;// normalization constant, must be reset for new
     parameter values

 public PositionWeightMatrixDiffSM( AlphabetContainer alphabets, int
     length, double ess ) throws IllegalArgumentException {
  super( alphabets, length );
  //we allow only discrete alphabets with the same symbols at all positions
  if(!alphabets.isSimple() || !alphabets.isDiscrete()){
   throw new IllegalArgumentException( "This␣PWM␣can␣handle␣only␣discrete␣
       alphabets␣with␣the␣same␣alphabet␣at␣each␣position." );
  }
  //create parameter-array
  this.parameters = new double[length][(int)alphabets.getAlphabetLengthAt(
      0 )];
  //set fields
  this.ess = ess;
  this.isInitialized = false;
  this.norm = null;
 }
```

```
/**
 * @param xml
 * @throws NonParsableException
 */
public PositionWeightMatrixDiffSM( StringBuffer xml ) throws
    NonParsableException {
 //super-constructor in the end calls fromXML(StringBuffer)
 //and checks that alphabet and length are set
 super( xml );
}

@Override
public int getSizeOfEventSpaceForRandomVariablesOfParameter( int index ) {
 //the event space are the symbols of the alphabet
 return parameters[0].length;
}

@Override
public double getLogNormalizationConstant() {
 //only depends on current parameters
 //-> compute only once
 if(this.norm == null){
  norm = 0.0;
  //sum over all sequences of product over all positions
  //can be re-ordered for a PWM to the product over all positions
  //of the sum over the symbols. In log-space the outer
  //product becomes a sum, the inner sum must be computed
  //by getLogSum(double[])
  for(int i=0;i<parameters.length;i++){
   norm += Normalisation.getLogSum( parameters[i] );
  }
 }
 return norm;
}

@Override
public double getLogPartialNormalizationConstant( int parameterIndex )
    throws Exception {
 //norm computed?
 if(norm == null){
  getLogNormalizationConstant();
 }
 //row and column of the parameter
 //in the PWM
 int symbol = parameterIndex%(int)alphabets.getAlphabetLengthAt( 0 );
 int position = parameterIndex/(int)alphabets.getAlphabetLengthAt( 0 );
 //partial derivation only at current position, rest is factor
 return norm - Normalisation.getLogSum( parameters[position] ) +
     parameters[position][symbol];
}

@Override
public double getLogPriorTerm() {
 double logPrior = 0;
```

```
 for(int i=0;i<parameters.length;i++){
  for(int j=0;j<parameters[i].length;j++){
   //prior without gamma-normalization (only depends on hyper-parameters),
   //uniform hyper-parameters (BDeu), tranformed prior density,
   //without normalization constant (getLogNormalizationConstant()*ess
       subtracted later)
   logPrior += ess/alphabets.getAlphabetLengthAt( 0 ) * parameters[i][j];
  }
 }
 return logPrior;
}

@Override
public void addGradientOfLogPriorTerm( double[] grad, int start ) throws
    Exception {
 for(int i=0;i<parameters.length;i++){
  for(int j=0;j<parameters[i].length;j++,start++){
   //partial derivations of the logPriorTerm above
   grad[start] = ess/alphabets.getAlphabetLengthAt( 0 );
  }
 }
}

@Override
public double getESS() {
 return ess;
}

@Override
public void initializeFunction( int index, boolean freeParams, DataSet[]
    data, double[][] weights ) throws Exception {
 if(!data[index].getAlphabetContainer().checkConsistency( alphabets ) ||
   data[index].getElementLength() != length){
  throw new IllegalArgumentException( "Alphabet␣or␣length␣to␣not␣match."
      );
 }
 //initially set pseudo-counts
 for(int i=0;i<parameters.length;i++){
  Arrays.fill( parameters[i], ess/alphabets.getAlphabetLengthAt( 0 ) );
 }
 //counts in data
 for(int i=0;i<data[index].getNumberOfElements();i++){
  Sequence seq = data[index].getElementAt( i );
  for(int j=0;j<seq.getLength();j++){
   parameters[j][ seq.discreteVal( j ) ] += weights[index][i];
  }
 }
 for(int i=0;i<parameters.length;i++){
  //normalize -> MAP estimation
  Normalisation.sumNormalisation( parameters[i] );
  //parameters are log-probabilities from MAP estimation
  for(int j=0;j<parameters[i].length;j++){
   parameters[i][j] = Math.log( parameters[i][j] );
  }
```

```
 }
 norm = null;
 isInitialized = true;
}

@Override
public void initializeFunctionRandomly( boolean freeParams ) throws
    Exception {
 int al = (int)alphabets.getAlphabetLengthAt( 0 );
 //draw parameters from prior density -> Dirichlet
 DirichletMRGParams pars = new DirichletMRGParams( ess/al, al );
 for(int i=0;i<parameters.length;i++){
  parameters[i] = DirichletMRG.DEFAULT_INSTANCE.generate( al, pars );
  //parameters are log-probabilities
  for(int j=0;j<parameters[i].length;j++){
   parameters[i][j] = Math.log( parameters[i][j] );
  }
 }
 norm = null;
 isInitialized = true;
}


@Override
public double getLogScoreFor( Sequence seq, int start ) {
 double score = 0.0;
 //log-score is sum of parameter values used
 //normalization to likelihood can be achieved
 //by subtracting getLogNormalizationConstant
 for(int i=0;i<parameters.length;i++){
  score += parameters[i][ seq.discreteVal( i+start ) ];
 }
 return score;
}

@Override
public double getLogScoreAndPartialDerivation( Sequence seq, int start,
    IntList indices, DoubleList partialDer ) {
 double score = 0.0;
 int off = 0;
 for(int i=0;i<parameters.length;i++){
  int v = seq.discreteVal( i+start );
  score += parameters[i][ v ];
  //add index of parameter used to indices
  indices.add( off + v );
  //derivations are just one
  partialDer.add( 1 );
  off += parameters[i].length;
 }
 return score;
}

@Override
public int getNumberOfParameters() {
```

```
 int num = 0;
 for(int i=0;i<parameters.length;i++){
  num += parameters[i].length;
 }
 return num;
}

@Override
public double[] getCurrentParameterValues() throws Exception {
 double[] pars = new double[getNumberOfParameters()];
 for(int i=0,k=0;i<parameters.length;i++){
  for(int j=0;j<parameters[i].length;j++,k++){
   pars[k] = parameters[i][j];
  }
 }
 return pars;
}

@Override
public void setParameters( double[] params, int start ) {
 for(int i=0;i<parameters.length;i++){
  for(int j=0;j<parameters[i].length;j++,start++){
   parameters[i][j] = params[start];
  }
 }
 norm = null;
}

@Override
public String getInstanceName() {
 return "Position␣weight␣matrix";
}


@Override
public boolean isInitialized() {
 return isInitialized;
}

@Override
public StringBuffer toXML() {
 StringBuffer xml = new StringBuffer();
 //store all fields with XML parser
 //including alphabet and length of the super-class
 XMLParser.appendObjectWithTags( xml, alphabets, "alphabets" );
 XMLParser.appendObjectWithTags( xml, length, "length" );
 XMLParser.appendObjectWithTags( xml, parameters, "parameters" );
 XMLParser.appendObjectWithTags( xml, isInitialized, "isInitialized" );
 XMLParser.appendObjectWithTags( xml, ess, "ess" );
 XMLParser.addTags( xml, "PWM" );
 return xml;
}

@Override
```

```
protected void fromXML( StringBuffer xml ) throws NonParsableException {
 xml = XMLParser.extractForTag( xml, "PWM" );
 //extract all fields
 alphabets = (AlphabetContainer)XMLParser.extractObjectForTags( xml,
     "alphabets" );
 length = XMLParser.extractObjectForTags( xml, "length", int.class );
 parameters = (double[][])XMLParser.extractObjectForTags( xml,
     "parameters" );
 isInitialized = XMLParser.extractObjectForTags( xml, "isInitialized",
     boolean.class );
 ess = XMLParser.extractObjectForTags( xml, "ess", double.class );
}
```

Jstacs reference card

Data handling

Alphabet: A set of symbols
new DiscreteAlphabet(caseInsensitive,alphabet): Create an arbitrary discrete alphabet
new ContinuousAlphabet(min,max): Create a continuous alphabet between min and max
DNAAlphabet.SINGLETON: Singleton instance of a DNA-alphabet

AlphabetContainer: A set of Alphabets and their assigment to positions
new AlphabetContainer(alphabets): Create an aggregate alphabet out of Alphabets
DNAAlphabetContainer.SINGLETON: Singleton instance of aggregate DNA-alphabet

Sequence seq: Representing a biological sequence
Sequence.create(alphabets,string): Create a sequence from a string
seq.getLength(): Obtain the length of a sequence
seq.discreteVal(pos): Obtain the discrete value at a position (counting from 0) of a sequence
seq.continuousVal(pos): Obtain the continuous value at a position (counting from 0) of a sequence

DataSet data: A set of sequences using the same AlphabetContainer
new DataSet(annotation,sequences): Create a data set from sequences
new DNADataSet(filename): Create a data set of DNA sequences from a FastA file
data.getNumberOfElements(): Obtain the number of sequences in a data set
data.getElementAt(index): Obtain the sequence at index from a data set
data.getInfixDataSet(start,length): Get a data set containing all infixes of a given length starting at a given position of all sequences in the current data set

Statistical models

StatisticalModel statMod: Interface for all statistical models
TrainableStatisticalModel trainSM: Interface for statistical models that can be trained from a single data set
DifferentiableStatisticalModel diffSM: Interface for statistical models that can be trained using gradient-based methods

TrainableStatisticalModelFactory: Factory for standard implementations of TrainableStatisticalModels
TrainableStatisticalModelFactory.createPWM(alphabets,length,ess): Create a PWM model of a given length
TrainableStatisticalModelFactory.createInhomogeneousMarkovModel(alphabets,length,ess,order): Create an inhomogeneous Markov model of a given length and order
TrainableStatisticalModelFactory.createHomogeneousMarkovModel(alphabets,ess,order): Create a homogeneous Markov model of a given order
TrainableStatisticalModelFactory.createMixtureModel(hyperpars,models): Create a mixture model from TrainableStatisticalModels

DifferentiableStatisticalModelFactory: Factory for standard implementations of DifferentiableStatisticalModels
DifferentiableStatisticalModelFactory.createPWM(alphabets,length,ess): Create a PWM model of a given length
DifferentiableStatisticalModelFactory.createInhomogeneousMarkovModel(alphabets,length,ess,order): Create an inhomogeneous Markov model of a given length and order
DifferentiableStatisticalModelFactory.createHomogeneousMarkovModel(alphabets,ess,order,priorLength): Create a homogeneous Markov model of a given order
DifferentiableStatisticalModelFactory.createMixtureModel(models): Create a mixture model from DifferentiableStatisticalModels

HMMFactory: Factory for standard implementations of hidden Markov models

statMod.emitDataSet(number,length): Generate a given number of sequences with specified length from this StatisticalModel using the current parameter values
statMod.getLogProbFor(sequence): Obtain the log probability (likelihood) of a sequence for this StatisticalModel
trainSM.train(data): Train a TrainableStatisticalModel from a data set
diffSM.initializeFunctionRandomly(): Initialize the parameters of this DifferentiableStatisticalModel randomly
diffSM.getLogScoreFor(sequence): Obtain a log score (typically proportional to the log-likelihood) of a sequence for this DifferentiableStatisticalModel
diffSM.getLogScoreAndPartialDerivation(sequence,indices,partialDers): Compute the partial derivations wrt. all parameters of this DifferentiableStatisticalModel for the given sequences and store the parameter indexes and corresponding partial derivations in given lists

Classifiers

AbstractClassifier classif: Abstract class of a classifier
new TrainSMBasedClassifier(models): Create a classifier from TrainableStatisticalModels that is learned by ML or MAP
new MSPClassifier(params,prior,models): Create a classifier from DifferentiableStatisticalModels that is learned by MCL or MSP
new GenDisMixClassifier(params,prior,learnPrinc,models): Classifier that learns DifferentiableStatisticalModels using a unified learning principle

classif.train(dataSets): Train a classifier from training data sets
classif.classify(sequence): Classify a sequence
classif.evaluate(performanceMeasures,exc,dataSet): Evaluate performance measures for a given classifier on test data sets

AbstractPerformanceMeasure: Abstract class of all performance measures
new NumericalPerformanceMeasureParameterSet(): Create a set of scalar standard performance measures that are applicable to two-class problems (binary classification)
new PerformanceMeasureParameterSet(measures): Create a set of performance measures
PerformanceMeasureParameterSet.createFilledParameters(): Create a set of scalar standard performance measures for binary classification problems that can immediately be used

Utilities

Storable: Interface of objects that can be stored to XML
XMLParser.appendObjectWithTags(buffer,storable,tag): Append storable object to StringBuffer with given tags
XMLParser.extractObjectForTags(buffer,tag): Extract storable object within tags from StringBuffer

Alignment align: Class for optimal alignments of sequences
new Alignment(type,costs): Create an object for alignments of sequences
align.getAlignment(seq1,seq2): Align two sequences

ArrayHandler.clone(array): Deep clone a multi-dimensional array
ArrayHandler.createArrayOf(template,num): Create an array containing num clones of a template

ToolBox.sum(doubles): Compute the sum of all elements in an array
ToolBox.getMaxIndex(doubles): Get the index of the maximum value in an array

Normalisation.getLogSum(doubles): Compute the logarithm of a sum of values given as their logs
Normalisation.sumNormalisation(double): Normalize a given array to probabilities