Manual for FunSiP (Functional Site Prediction)

Michiel Van Bel, Yvan Saeys, Yves Van de Peer

12th December 2007

1 Overview

FunSiP is a software package which bundles several programs that can be used for the prediction of genes and the annotation of genomes. Also, several associated programs are included for pattern discovery and detection of regulatory features inside sequences.

Since the software relies on the use GPLv2 software, we also publish both the software and the documentation as GPLv3. Some programs inside the software package rely on an external software tool for predicting the secondary structure of RNA (standard: RNAfold). This software is also freely availbale as GPL'ed software, but is not included. It can however easily be obtained from various internet sources.
The entire software package was written in java, using Java 5 features. As such, all programs can be run with either a Java 5 or later Runtime Environment.

1.1 Description of the software package

The software package consists of various programs sharing a single library. The main different programs (and their associated uses are):

- **Functional site recognition:**
  The program supports the detection of both splice sites and start sites (and others, if properly configured) in any sequence of an organism for which a trained model (or training data) exists. The idea is the same as in SpliceMachine [1]: extract high-dimensional data from the sequence, build a classifier with this data and then classify each possible site (this is done by considering fixed motifs: AG for acceptor, GT/GC for donor, ATG for start) into either a pseudo site or a true site. See Figure 1 for an example, applied to acceptor site recognition. We have extended the expressive power of these ideas by generalizing the possible site detection. This is done by allowing the user to provide the fixed motif by which the initial detection occurs. For further information, see section 4.

- **Feature selection:**
  By including native support for feature selection techniques, we allow a more precise detection of functional sites. The feature selection is done by applying the feature selection techniques to the training data, which then provides a list of scores (each score is associated with a single feature). This list then acts as a filter (e.g. use only the features with non-zero scores) for both the training and future tests. It has been shown that the feature selection techniques can significantly improve the performance of classification methods, see section 4. Feature selection of training data can run either as a standalone program, or it can be used inside other programs such as the functional site prediction program. For further information, see section 3.4.

- **EuGene Output code Generation:**
  FunSiP was built upon the same theoretical foundations as SpliceMachine. The tight coupling between the EuGene genome annotation platform and SpliceMachine made it imperative that FunSiP was implemented in such a way that the conversion from SpliceMachine to FunSiP is done as painless as possible.

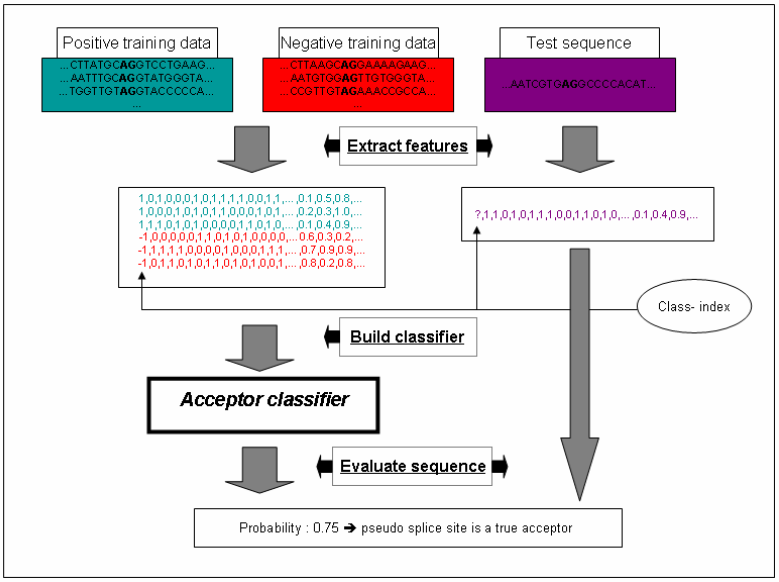

Figure 1: Framework of FunSiP, applied to acceptor site recognition

2 General options

The runtime process contains options that are applicable for each of the different programs. These options are mostly defined in the general-options part of the configuration files (see section 4).

2.1 Multi-threading and cluster support

As usual, most processes are parallel in nature. This also applies to the *functional site recognition* program. We have chosen for a coarse grained approach of the parallelization problem. This implies that only rather large building blocks are used for parallelization. For example, when considering splice site recognition: normally one would expect both a donor prediction and an acceptor prediction for a certain sequence. Our parallization just considers these two as the parallel building blocks, which are called **ClassificationActions**. The finer grained approach (e.g. different threads for each seperate string of sequence to be examined) provided more overhead then it saved. Thus we opted for the coarse grained approach. At startup the user can define three possible modi of operandi:

- No multithreading or cluster support. This is the default setting. All possible classificationactions are executed in a serial way.
- Single pc multithreading. This setting executes different local threads, with each thread containing a different classificationaction. This approach calls for some changes, considering race conditions for differerent files etc. However, on a multi-core computer this type of execution should result in drastically reduced computation times. When parallelizing two classification actions the runtime is not split in two though (rather something like 1.8). I/O bottlenecks are the main reason for this, as both threads will be busy reading and writing files at the same time.
- Cluster support. Our package has support for the Grid Engine clustersoftware [5]. Each classification action can thus be launched on a different node of a cluster. The launch of the initial program should however be done from the masternode, as not all grid engines support the launching of new jobs from within the cluster itself.

3 Functional site Prediction

3.1 Overview

FunSiP was originally conceived as a replacement for the SpliceMachine program. Thus, one of the basic abilities of the program is the capability to predict splice sites in eukaryotic genes. FunSiP is based on the same general idea of SpliceMachine: extracting high-dimensional features from sequences in order to build a classification model, then use this classification model to predict splice sites in a genomic sequence. By rewriting the code we were able to design a more modular and extensible functional site prediction platform.

FunSiP consists of a series of classification actions that are created after parsing the configuration file (see Figure 2). These actions may progress in a number of different ways: they can either process training data and build a classification model, or they can process genome sequence data and predict functional sites (see Figure 3).

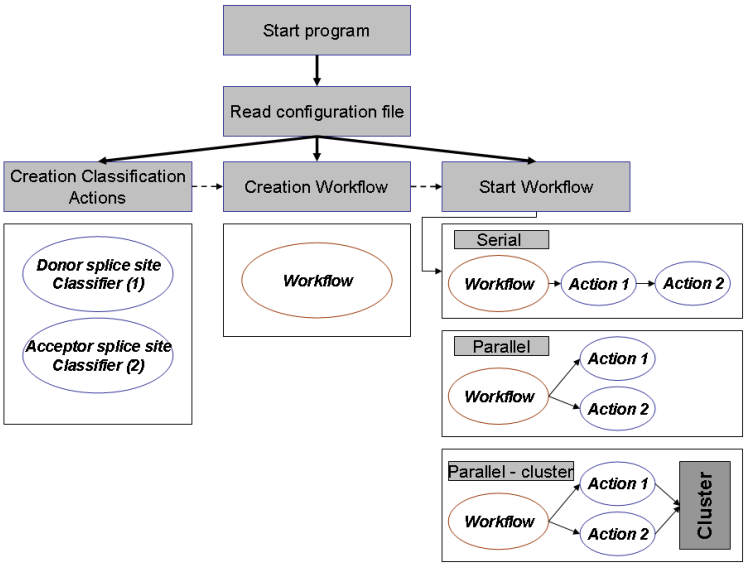

Figure 2: General workflow in FunSiP

3.2 Classification Features

3.2.1 Introduction to Classification Features

When classifying functional sites in a DNA/RNA-sequence, the classifier (most likely a Support Vector Machine) needs to be supplied with consistent data that is extracted from both the training sequences (in order to build the classification model) and from the test sequences (in order to evaluate the functional sites). The way in which the data is extracted from the training sequences is highly important because it is one of the main influences on the predictive performance of the classifier.
The extraction of data from sequences is done by a system of so-called *Classification Features*. These *Classification Features* operate in an independant way from each other, and extract a specialized type of data from the sequences. Later on, the data is concatenated and fed to the classifier in order to build the classification model. Examples of *Classification Features* are (a full list can be found later on) :

- **Positional Features**: These features see what type of k-mer is present at each position in the sequence. This type of feature performs very well, especially when the data extraction is limited to a small upstream and downstream range from the functional site (see further).
- **Compositional Features**: These features count the number of occurences of each k-mer, and supply this data to the classifier. This way a certain codon or other compositional bias that is non-positional can be detected.

It is clear that a huge amount of different *Classification Features* can be defined. However, it should also be clear that only a pretty limited number of *Classification Features* can extract meaningful data from the sequences, that can be used to discriminate between positives and negatives (i.e. classification). It requires a great

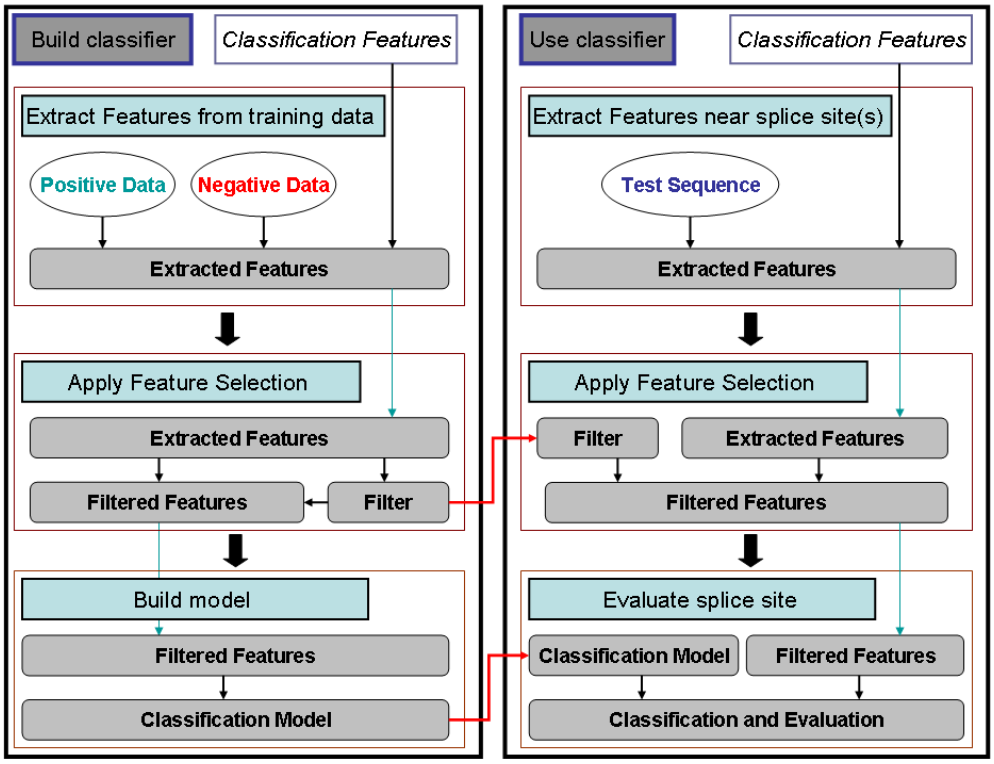

Figure 3: Classification workflow in FunSiP (applied to splice site classification)

amount of deduction, testing and insight to produce good *Classification Features*. Furthermore, most *Classification Features* have parameters that influence their impact on the classification process. Examples of these are:

- **Upstream and downstream ranges**: When classifying a functional site, we can consider both data from upstream or downstream regions around the functional site. Either way, it is also a fact that the range itself has an influence.
- **Length**: When we extract compositional (or positional) features from the sequence, it is clear that a different amount and type of influential data is extracted when we consider k-mers of varying length. For example (when considering compositional features): we can count the number of occurences of single nucleotides in a certain window of the sequence, and this will reveal perhaps some bias towards the use of certain nucleotides. However, when we extend this to dinucleotides, codons, ... we see that we gain additional information, because the neighbouring nucleotides are also perceived as being influential.

As such, we combine different types of *Classification Features* in order to capture as much (and different) information as possible in order to maximize the predictive performance. This selection of *Classification Features* is done by iterating across a number of upstream and downstream ranges for each (supplied) *Classification Feature*, crossvalidating the training data using the supplied parameters. While this

procedure does not try each and every possible combination of features and their parameters, it is a sufficient approximation.

3.2.2 Conversions

By using *Classification Features* we can hope to extract enough information. However, it is fairly difficult to capture dependencies among nucleotides that are not adjacent. An example of this behaviour is the supposed influence of the RNA secondary structure on the splicing process (see [2]). We are also able to extract information and data from this RNA secondary structure, by the use of exactly the same *Classification Features* as used for the primary RNA/DNA sequence, or by the use of specially designed *Classification Features*. In order to accomodate for these dependencies, we use the notion of *Conversions*. A *Conversion* is a string of characters that is produced by converting the original DNA sequence. Technically, all this could also be done by supplying parameters to certain *Classification Features*. There are however several reasons why we have opted not to use this approach:

1. The computational cost required for deducting certain *Conversions*, such as the RNA secondary structure, is prohibitively high. Because there is no communication between the different *Classification Features*, the same *Conversion* would be computed several times again, leading to a severe performance penalty. Therefore, *Conversions* that are defined to be used (by the configuration file) are computed at the beginning of the program and supplied to the *Classification Features*.

2. By seperating the *Conversions* from the *Classification Features* we both increase the reusability of the code and adhere to a decent object oriented design.

3. By seperating the *Conversions* from the *Classification Features*, we make it possible to add new *Conversions* in a plug-and-play way without having to change *Classification Features*.

Thus, certain *Classification Features* have the additional option of receiving a *Conversion* parameter. This can lead to an increase in the amount of useful extracted information. However, it should be noted that the creation of useful *Conversions* which capture information from (non-adjacent nucleotides), is not an easy task. A list of currently usable *Conversions* is described in section 4.

3.2.3 Creating your own Classification Features and Conversions

The modularity of the platform allows the user to easily create new *Classification Features* and *Conversions*, and have them used by the program with minimal effort. Indeed, the only requirements are that the necessary interfaces are extended (ClassificationFeature.java or Conversion.java) and that the implementation is placed in the correct directory:

- ./util/classificationFeatures/implementations
- ./util/conversions/implementations

There is however one small caveat: each *Classification Feature* or *Conversion* is recognized by an identifier (the same used as by the configuration files). This identifier needs to be unique, otherwise an incorrect *Classification Feature* or *Conversion* might be loaded and used.

3.2.4 Testing new classification features

Ease of use when applying newly defined classification features to data is one of the key advantages of FunSiP. Certain care needs to be taken care however when testing these new features and interpreting their results.
Testing new features is typically done by crossvalidating the data that is acquired by extracting these features from the training data. It is however highly recommanded that you follow the steps below:

1. Run a test with standard positional features of limited length and ranges (e.g. P 1 20 20) to gain a general view of how well these features are able to discern between between positive data and negative data. Other results of crossvalidation should be compared to these "standard" results. Take care to use a non-trivial amount of training data. 10-fold crossvalidation with 10 positives and 100 negatives that are extracted from a total amount of 10000 positives and 500000 negatives will be totally biased! Using all training data is not necessary (and even counter productive, as it might lead to computer memory problems and overfitting), but consider using a decent amount of training data.

2. Compute how many features will be extracted by the new classification feature. If this number is low to very low, in comparison to the amount of training data (ratio is smaller then 0.1), the classifier will have trouble seperating the positive and negative data, mostly resulting in the fact that all trainingdata will be labeled as negatives during the crossvalidation phase. If this is the case, go to step 3, otherwise go to step 4.

3. Crossvalidating the data with only the new features will not work. Therefore, in order to test the predictive performance of the new features, it is necessary to use two classification features. The first one is the standard (see point 1), the second one is the newly defined classification feature. The results of the crossvalidation can hereafter be compated to the results of the standard crossvalidation. After this phase, go to step 5.

4. If the amount of extracted features is sufficient, then they might be used for crossvalidation. The results of this crossvalidation can then be an indication to the predictive performance of the new type of classification feature. Go to step 5.

5. It remains however to be seen whether or not this new classification feature adds any "new" information to the amount of information that is already gathered by the typical classification features used (positional and compositional

features of various length and ranges). Therefore, as a final test towards the feasibility of the new classification feature, it might prove usefull to perform a last test:

(a) Use a combination of positional and compositional features as a standard to retrieve crossvalidation results.

(b) Use the same combination of positional and compositional features, and add the newly designed classification feature this list. Then use all these features for crossvalidation.

(c) Compare the results of both crossvalidations.

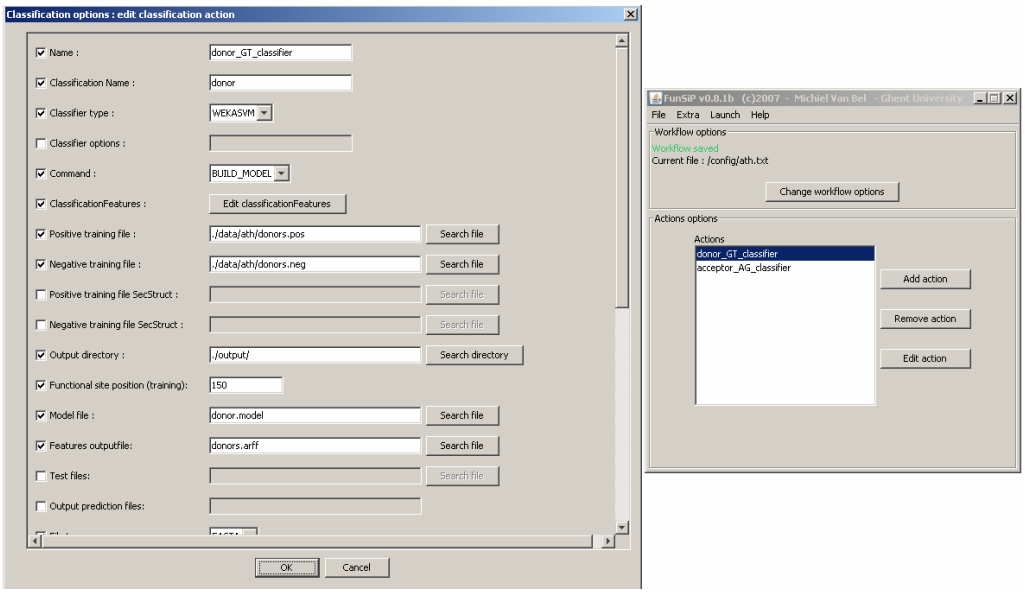

Figure 4: Various options available in the user interface of funsip.

3.3 Usage of the FunSiP program

3.3.1 Starting the program

The FunSiP program can be used either from the commandline or from the Graphical User Interface (GUI in short). The program is supplied as a jar-file, and the necessary batch-files / script-files are provided to start the program in GUI-mode. If necessary, the program can also be started manually as:

- java -jar FunSiP.jar -c [configuration-file] <sequencefile(s)> : This will start the FunSiP program in non-interactive mode and launch the supplied configurationfile (and - if sequencefiles are provided and the configurationfile is correct - annotate the sequencefile(s)). The program will run in commandline mode (hence the -c option) if started this way.

- java -jar FunSiP.jar : This will start the FunSiP program with a default configurationfile loaded. The program will run in GUI mode. The program will be started this way if the user double-clicks the jar-file in his window manager (explorer for windows).
- java -jar FunSiP.jar [configuration-file] : This will start the FunSiP program in GUI mode with the supplied configurationfile preloaded. The user then has the ability to change several settings, or launch the configurationfile right away.

If the jar-file is unpacked or you did not receive the jar-file and received only the sourcefiles, then you can launch the program in a different way. Some scriptfiles (*nix) and batchfiles (Windows) are provided with the source.

- FunSiP.sh [configurationfile] <sequencefile(s)>: *nix shellscript which launches the FunSiP program in commandline mode, with the supplied configurationfile and it annotates (if applicable) the supplied sequencefile(s).
- FunSiPGui.sh <configurationfile> : *nix script which launches the FunSiP program in gui mode, and preloads the configurationfile (if supplied, otherwise it loads the standard config file).
- FunSiP.bat [configurationfile] <sequencefiles(s)> : batchfile which launches the FunSiP program in commandline mode, with the supplied configurationfile and it annotates (if applicable) the supplied sequencefile(s).
- FunSiPGui.bat <configurationfile> : batchfile which launches the FunSiP program in gui mode, and preloads the configurationfile (if supplied, otherwise it loads the standard config file).

3.3.2 Using the Graphical User Interface

The Graphical User Interface (GUI) has several useful extra properties compared to the standard commandline interface. The GUI supports both the creation and adaptation of new configurationfiles, with some supplied wizards to help inexperienced users with the creation of configuration files. Indeed, while the base system is not very complicated, there are numerous options, which can easily confuse new users. Of course, the user still has the ability to just change the configurationfile by hand an launch it by using the commandline.

3.4 Feature selection

3.4.1 Introduction to feature selection

Feature selection is a complex topic in the machine learning. Feature selection techniques aim to optimize the performance of classification methods by selection the optimal features, prior to the actual building of the classification model. Of

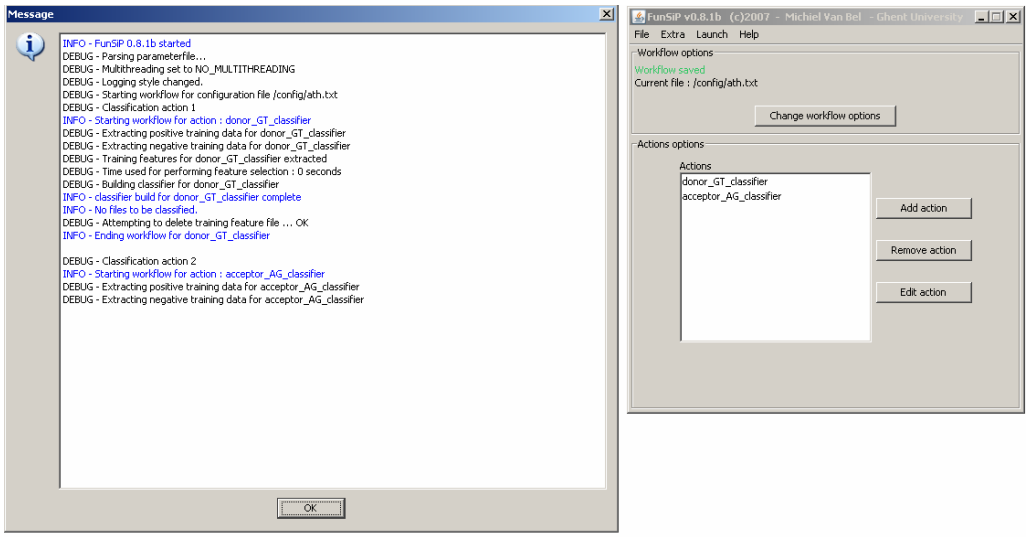

Figure 5: Runtime building of classification model in the user interface.

course, defining which features are optimal is the key problem here, and a lot of different approaches have been suggested, with varying succes and with difficulties to compare the approaches since some approaches perform very good for a very specific type of features, while others are more general in nature. Another fact is that some feature selection techniques can take care of dependencies among the different features (multivariate), while others do not (univariate). One would expect that multivariate techniques would result in better performance, since they would gather additional information. We have however tested 2 different multivariate techniques (CFS and FCBF), and the results were very disappointing [3].

3.4.2 Using feature selection

When the user opts to use feature selection techniques in order to increase the predictive performance of the program, he has to define several things:

1. Build a classification model in the normal way, while at the same time setting the feature selection mode to USE_AND_WRITE_FULL_FILTER . Of course, the feature selection type should be set to a valid feature selection algorithm, and a valid file indicator for feature selection output file. Setting this feature selection mode will result in the feature selection algorithm being applied to the extracted features. The resulting filter will then be applied to the training data - resulting in a smaller and more performant set of training data - and the filter will be written to an output file. After this, the classification model will be build.

2. Evaluate a sequence-file with the classification model. This is done by setting the feature selection mode to LOAD_FILTER and by providing the feature selection filter in the feature selection input file variable (this location should be the same as the output file variable defined above). This way, the filter is

read from the file and applied to the extracted features from the sequence-file, resulting (hopefully) in a better computational and predictive performance.

4 Configuration File

Normally one would use the GUI for creating configuration files. However, some might want a better insight into the inner workings and options that are available through the configuration file. This section defines all the possible parameter options that can be used in the configuration files used by FunSiP. Normally, this list should have become pretty stable, and not change that much anymore during future updates.

4.1 General remarks

- Comments can be made by placing a # -character in front at the beginning of the line. The parser works on a line-by-line basis, so a line starting with #--character is a comment, but a line with a # -character in the middle is either malformed or the # is interpreted as a character.
- There are always at least two sections in the configuration file. Each section is started with a specific keyword (see also 4.2.1 and 4.2.2) and ended with the END keyword.
- The configuration options consist of both a keyword (indicating what the option is supposed to do) and a value. The keyword and the value are seperated by at least one tab (i.e. one or more). If the value consists of multiple subvalues, these subvalues are seperated by one or more spaces. It is important to keep this policy consistent.
- Of course, some options are dependent on others (and so on). We have also provided the preferred multiplicity (the number of times an option can appear inside a single configuration-file block) of every option.
- Configuration files which contain options or values that cannot be parsed report an error indicating (possibly, this is not always feasable) where and what is wrong. The program will terminate, instead of continuing with default or wrong options.

4.2 Overview of the different sections.

4.2.1 General Information

Keyword : GENERAL_OPTIONS
This section contains the parameters that are independent of the actions (see later). Only one GENERAL_OPTIONS section should be defined in the configuration file. Possible options (with their respective possible values) in this section are:

1. multithreading: indicates how the different actions (and possibly different to be examined sequence files) need to be scheduled by the Java VM.
   - Keyword : MULTITHREADING
   - Multiplicity : 0 or 1
   - Possible values :
     - NO_MULTITHREADING : default setting, runs everything in serial way.
     - SINGLE_PC_MULTITHREADING : creates a number of local Java threads for every action and every sequence file. Should only be used on multi-core machines, since FunSiP is mostly CPU dependent and not IO dependent.
     - CLUSTER_MULTITHREADING : uses the Grid Engine cluster (qsub commands and such) to spread the workload. Possibly only available from unix/linux systems.
2. logging style: indicates the amount of output that is produced by the SPR system. It is based on log4j [6], so the logging style indicators are identical. The indicators (as shown below) show all data of their setting, AND those below in the list. E.g. setting the style to WARN will show WARN messages, ERROR messages and FATAL messages.
   - Keyword : LOGGING_STYLE
   - Multiplicity : 0 or 1
   - Possible values :
     - DEBUG : prints every output statement, for development purposes or those who like verbose output.
     - INFO : prints informative output only, default setting.
     - WARN : prints warning statements (rarely used).
     - ERROR : prints error messages, normally generated by java exceptions.
     - FATAL : prints error messages, followed by the termination of the program. These error messages are generated by exceptions.
3. logging output: instead of directing the generated output to screen, the output can also be written to a file. WARNING (due to some confusion about terminology): the 'output' indicator used above only applies to the workflow output. Data from either crossvalidation or sequence evaluation are written to their respective files automatically. These files can be located in the designated outputdirectory (see further).
   - Keyword : LOGGING_OUTPUT
   - Multiplicity: 0 or 1
   - Possible values :

  - CONSOLE : default, redirects output to screen (on commandline prompt).
  - FILE [fileName] : redirects output to a file (indicated by fileName).

4. Merging of prediction output files: This is necessary if more than one 'action' (see below) is defined. E.g. Donor-recognition 'action' and acceptor-recognition 'action'. By setting this option, both prediction files can be merged into one resulting file.

   - Keyword : MERGE_PREDICTION_OUTPUT
   - Multiplicity: 0 or more
   - Possible values:
     - [fileName1][fileName2][...][resultFileName] : at least 3 filenames need to be defined. All but the last are thought of as sourcefiles, the last one as the file to which the merged prediction should be written. Normally, the prediction outputfiles contains data that is sorted by a number in the first column. This sorting is then applied to the resulting file.

5. Optimization of the program parameters : The identification and prediction of various biological sites (donor, acceptor,...) is done by creating a classifier that takes features which are extracted from various sequences and by building a model with these features (see manual for a more detailed explanation). The various types of feature-extraction are dependent on their own parameters. Setting the optimization option will try to create the optimal settings for this particalur type of classifier. WARNING: These computations can take a VERY long time (as in several days) if it is done with:

   (a) a small interval parameter and/or a large maximum_range (see Possible Values).

   (b) a large number of classificationFeatures (these extract the features)

   (c) feature selection algorithms

   (d) a large number of training examples

   This is due to the fact that every possible optimal setting is tested, which can of course quickly ramp up to significant numbers.

   - Keyword : OPTIMIZER
   - Multiplicity: 0 or 1
   - Possible values: (all integer)
     - [minimum_range][maximum_range][interval] : every (well, with some exceptions which should not be optimized this way) classificationFeature is dependend on both an upstream-range and a downstream-range from the site that needs to be examined. These 3 integers define the ranges that will be examined for both upstream - and downstream use.

EXAMPLES:

```
# Example which optimizes features, with virtually no output
GENERAL_OPTIONS
     MULTITHREADING          NO_MULTITHREADING
     LOGGING_STYLE           FATAL
     OPTIMIZER               10 80 10
END

# Example which uses the cluster to predit some files and merge them
GENERAL_OPTIONS
   MULTITHREADING          CLUSTER_MULTITHREADING
   LOGGING_STYLE          DEBUG
   MERGE_PREDICTION_OUTPUT ./output/donor.pred ./output/acceptor.pred ./output/bo
END
```

4.2.2 Action Information

Keyword : ACTION
These sections (multiplicity is one or greater than one) define the different possible actions that can be used. E.g. donor site prediction and classification. Now, as it is, every action is independent of every other action, thus allowing everything to become parallelizable. It may become difficult for new users to properly grasp the possibilities of an *ACTION* due to the high number of possible options. We therefore provide first a short list with the options grouped by purpose:

1	NAME	Standard information
2	CLASSIFICATION_NAME	Standard information
3	CLASSIFICATION_FEATURE	Standard information
4	OUTPUT_DIRECTORY	Standard information
5	COMMAND	Standard information
6	MODEL_FILE	Standard information
7	CLASSIFIER_TYPE	Classification mechanisms information
8	CLASSIFIER_OPTIONS	Classification mechanisms information
9	POSITIVE_TRAINING_FILE	Classification model building information
10	NEGATIVE_TRAINING_FILE	Classification model building information
11	POSITIVE_TRAINING_FILE_SECSTRUCT	Classification model building information
12	NEGATIVE_TRAINING_FILE_SECSTRUCT	Classification model building information
13	SPLICESITE_POSITION	Classification model building information
14	POSITIVE_TRAINING_AMOUNT	Classification model building information
15	NEGATIVE_TRAINING_AMOUNT	Classification model building information
16	FEATURES_OUTPUT_FILE	Classification model building information
17	TEST_FILES	Classification model usage information
18	OUTPUT_PREDICTION_FILES	Classification model usage information
19	FILE_TYPE	Classification model usage information
20	STRAND	Classification model usage information
21	CLASSIFICATION_PATTERN	Classification model usage information
22	CLASSIFICATION_PATTERN_LOCATION	Classification model usage information
23	CLASSIFICATION_PATTERN_OUTPUT	Classification model usage information
24	CROSSVALIDATION	Crossvalidation information
25	COMPLEXITY_CROSSVALIDATION	Crossvalidation information
26	MAXIMUM_CROSSVALIDATION	Crossvalidation information
27	FEATURE_SELECTION_TYPE	Feature selection information
28	FEATURE_SELECTION_PURPOSE	Feature selection information
29	FEATURE_SELECTION_INPUT_FILE	Feature selection information
30	FEATURE_SELECTION_OUTPUT_FILE	Feature selection information
31	FEATURE_SELECTION_MAX_FEATURES	Feature selection information

A full explanation of the possible options (with their respective possible values):

1. Name of the 'action' in the workflow. Will only be used throughout the workflow and possibly appear in the logging output.
   - Keyword : NAME
   - Multiplicity : 1
   - Possible Values: (String)
     - [name] : a string indicating the name.
2. Name of the 'action' that is used in the results. Certain files will get this name as prefix to make the distinction with the output of other 'actions'. Normally, this name will be shorter than the one provided above.

- Keyword : CLASSIFICATION_NAME
- Multiplicity : 1
- Possible Values : (String)
  - [name] : a string indicating the name used for output purposes

3. The type of features that will be extracted from sequences: classificationFeatures are one of the main backbones of the program. Each classificationFeature extracts a certain amount of features (floating point numbers mostly) from a sequence, and adds these to a list. Each classificationFeature always produces the SAME amount of features for a different sequence, so that all features are properly aligned. This is necessary in order for the classifiers to function properly. All classificationFeatures are identified by a unique name (first part of space-separated value), and (eventually) a number of parameters for this classificationfeature, in order to optimize the prediction results. During the design phase of the program, the decision was made to make the different classificationFeatures dynamically loadable. This means that one person only needs to implement a certain interface and place the resulting class in the correct directory. While this approach is very usefull for researchers, there always exists the possiblitiy of name-clashes between classificationfeatures : the unique name can be shared, with all resulting problems. Researchers should thus be carefull when assigning new names to newly build classificationfeatures.

   - Keyword : CLASSIFICATION_FEATURE
   - Multiplicity : 1 or higher
   - Possible Values : (we will give an overview of currently available classificationfeatures with their respective parameters. Applied to splice site prediction to make a good example. For further explanation of the different types of classification features, one should look at the manual.)
     - P [length][up][down] : this feature extracts positional features of length *length* in the interval {splice site-up,splice site+down} of a sequence. *Length, up* and *down* are all integers. Care should be taken not to choose a to great *length*, since this will result in a very sparse feature matrix.
     - C [length][up][down] : this feature extracts compositional (occurence based) features of length *length* in the interval {splice site-up,splice site+down}. A difference is made between the number of occurences upstream and downstream, in order to make a decent distinction between e.g. exons and introns. *Length, up* and *down* are all integers.
     - PG [length][up][down][conversion] : This is the same as the positional feature detailed above. However, the extra parameter "conversion" details a conversion made to the DNA/RNA-sequence to another sequence of the same length. Examples of conversions are $DNA \mapsto$ AminoAcid and $RNA \mapsto$ secondary structure. In the extra addendum (see below) a list of conversions is described, and ways of creating your own conversions is also detailed.

- CG [length][up][down][conversion] : This is the same as the compositional feature detailed above, with the same extra notion as with PG-features.
- BP [min_up][max_up] : this feature extracts branchpointfeatures from a DNA/RNA sequence (can only be used for acceptor splicesite recognition). This includes distance to the splicesite, binding potential to U2RNA and pyrimidinetract.
- RF [up][down] : Extracts occurence based features from all 3 reading frames, with a difference between upstream and downstream from the site to be examined. The features are extracted from the interval {site-up, site+down}. Both *up* and *down* are integers.
- GC [length][up][down] : Extracts the GC-content in a certain shifting frame (indicated by *length*) and thus creates a series of features of the same length as the sequence. *Length, up* and *down* are all integers. The shifting frame will move in the interval {site-up,site+down}.

- Addendum : Conversions. Conversions are functions which map the sequence to another sequence. The original sequence is DNA/RNA and thus consist of a 4-letter alphabet, while the resulting sequence can have any type of alphabet. When translating the sequence to aminoacids for example, the new alphabet exists of the 20 different possible types of aminoacids. Conversions are very usefull since they can be used to capture dependencies between nucleotides which would otherwise be unnoticed (for example RNA secondary structure). Since the implementation of these conversions is also a lot more easy than the classification features, and since conversions can also be dynamically added, experimentation should be fairly simple with these. As is the same case with classificationfeatures, conversions are also identified by a unique name, thus having the possibility of name clashes (because of careless researchers). Possible Conversions are:
  - DNA_AA : translates every three nucleotides into their respective amino acid.
  - DNA_MK : translates every nucleotide into his respective amino/keto part. $A, C \mapsto$ M and $G, T \mapsto$ K
  - DNA_RY : translates every nucleotide into his respective pyrimidine/purine. part. $A, G \mapsto$ R and $C, T \mapsto$ Y
  - DNA_SW : translates every nucleotide into his respective strong/weak interaction counterpart. $A, T \mapsto$ W and $G, C \mapsto$ S

4. Name of the directory in which all output files should be placed. If the indicated directory does not exist, then it is created.

   - keyword : OUTPUT_DIRECTORY
   - multiplicity : 1
   - Possible values :

  - [dirName] : string with the path to the outputdirectory

5. Indicator of the job that needs to be performed. This indicates whether a classification model should be build or whether it should be loaded. The first case is mainly for researchers who want to test new things. The second case is for general use by the public.

   - keyword : COMMAND
   - multiplicity : 0 or 1
   - Possible values :
     - BUILD_MODEL : default, builds a new model from training data
     - LOAD_MODEL : loads a model and uses it for classification

6. Name of the file containing the classification model. There is a small catch here: if the COMMAND is BUILD_MODEL, then you can just provide a name and the model will be created and be placed in the OUTPUT-directory. However, if the COMMAND is LOAD_MODEL, then the user should provide the exact path to the model-file (absolute path as in C:/foobar or /foobar/ or relative path as in ./foobar/).

   - keyword : MODEL_FILE
   - multiplicity : 1
   - possible values : (string value)
     - [fileName] : name of the model-file

7. Classifiertype used for building and evaluating models: There are a great number of different classifiers that can be used as a basis for evaluating sequences. However, there is a lot of extra work that needs to be done (specific to this project) by each classifier, thus limiting somewhat their usability because of coding work. EDIT : only one classifier working currently : WEKASVM

   - Keyword : CLASSIFIER_TYPE
   - Multiplicity : 1
   - Possible Values:
     - WEKASVM : uses the SMO implementation of the Weka machine learning platform.

8. Each classifier has a number of different options, of course dependant on their implementation. When using SVM's as classifier for example, there is always the complexity-option (mostly indicated by -C or -c) that gives the penalty for having datapoints on the wrong side of the hyperplane. As said, these options are dependent on the classifier and are passed directly to the classifier that is used.

   - Keyword : CLASSIFIER_OPTIONS

- Multiplicity : 0 or 1
- Possible Values :
  - [dependent on the classifier type]

9. Name of the file with the positive training data: this data should of course be properly aligned (see manual for an example).
   - Keyword : POSITIVE_TRAINING_FILE
   - Multiplicity : 0 or 1 (depending on COMMAND, see below)
   - Possible values :
     - [fileName] : just a string with the path to the positive training file
10. Name of the file with the negative training data: this data should of course be properly aligned (see manual for an example).
    - Keyword : NEGATIVE_TRAINING_FILE
    - multiplicity : 0 or 1 (depending on COMMAND, see below)
    - Possible values :
      - [fileName] : just a string with the path to the negative training file.
11. Name of the file containing the secondary structures of the positive training data. Because the computation of the secondary structures can take a pretty long time, we have opted to include this option to reduce computation time.
    - keyword : POSITIVE_TRAINING_FILE_SECSTRUCT
    - multiplicity : 0 or 1
    - Possible values :
      - [fileName] : just a string with the path to the positive secstruct training file
12. Name of the file containing the secondary structures of the negative training data. Because the computation of the secondary structures can take a pretty long time, we have opted to include this option to reduce computation time.
    - keyword : NEGATIVE_TRAINING_FILE_SECSTRUCT
    - multiplicity : 0 or 1
    - Possible values :
      - [fileName] : string with the path to the negative secstruct training file
13. Position of the functional site in the trainingdata.
    - keyword : SPLICESITE_POSITION
    - multiplicity : 1

- possible values : (integer value)
  - [position] : position of the splice site (typically something like 150)

14. Indicator for the amount of positive training data that is extracted from the positive training file. If this option is omitted, the total content of the training file will be used.

    - keyword : POSITIVE_TRAINING_AMOUNT
    - multiplicity : 0 or 1
    - Possible values : (integer value)
      - [num] : The amount of examples to be extracted from the provided POSITIVE_TRAINING_FILE.

15. Indicator for the amount of negative training data that is extracted from the negative training file. If this option is omitted, the total content of the training file will be used.

    - keyword : NEGATIVE_TRAINING_AMOUNT
    - multiplicity : 0 or 1
    - Possible values : (integer value)
      - [num] : The amount of examples to be extracted from the provided NEGATIVE_TRAINING_FILE.

16. Name of the file containing the extracted features (can be usefull to examine, therefore we have opted to place it somewhere). The file will automatically be placed inside the OUTPUT directory.

    - keyword : FEATURES_OUTPUT_FILE
    - multiplicity : 0 or 1 (depending on COMMAND).
    - possible values : (String value)
      - [fileName] : Name of the file which will be used to write the extracted features (and possible some headerdata) to. If it has the wrong extension, the file extension provided above can be used to elonge the filename.

17. Names of the files that needs to be examined by the classifier. E.g. a couple of fasta-files for which the splicesite need to be examined. It is possible here to provide the *-wildcard so all files in a certain directory will be examined.

    - keyword : TEST_FILES
    - multiplicity : 0 or 1 (depending on COMMAND and CROSSVALIDATION)
    - Possible values :

  - [fileName][...] : list of space-seperated paths to the files that need to be examined.

18. Names of the files to which the output of the prediction of the classification should be written. These files will then be place inside the OUTPUT_DIRECTORY. Normally the amount of outputfiles should match the amount of testfiles, but if this is not the case, the outputfilenames will be created on the fly. However, in this case the MERGE_OUTPUT control will most likely not work.

    - keyword : OUTPUT_PREDICTION_FILES
    - multiplicity : 0 or 1 (depending on TEST_FILES of course)
    - Possible values : (String values)
      - [fileName][...] : list of space-seperated filenames for the output

19. Filetype of the test files. Because each filetype (fasta, embl) has its own internal structure, this indicator is needed.

    - keyword : FILE_TYPE
    - multiplicity : 0 or 1
    - Possible values : (String values)
      - FASTA : Fasta file type
      - EMBL : EMBL file type

20. Indicator for the strand of the testfiles on which the classification action should be performed.

    - keyword : STRAND
    - multiplicity : 0 or 1
    - Possible values :
      - FORWARD : forward strand only
      - REVERSE : reverse strand only
      - BOTH : default, both strands

21. Pattern that is used for primary selection, prior to the actual classification. Using the acceptor splicesite recognition as an example, we see that every acceptor ends with the AG-pattern. Thus, when evaluating a sequence file, the first selection is done on this pattern: a list of all positions in the sequence that have the AG-pattern is compiled, and then the contents of this list are used for classification.

    - keyword : CLASSIFICATION_PATTERN
    - multiplicity : 1
    - Possible values : (String value)

  - [pattern] : A String (not null, length greater or equal than 1 – although length 1 really is too small, as it will result in an enormous amount of selected positions.) that indicates the pattern to be used for selection prior to classification.

22. An extra indicator for locating the pattern inside a sequence. The basic rationale behind this is the fact that when we do pattern matching to find the primary selection, then the program has no way to determine whether the site of interest is before the pattern (e.g. donor splice site recognition – GT -pattern) or behind the pattern (e.g. acceptor splice site recognition – AG-pattern) or even somewhere else. Thus, this indicator adds or subtracts a certain number from the position of the located pattern. The initial position of the located pattern is the position of the first character of the pattern. If this option is omitted, then no increase/decrease is performed on the pattern location.

    - keyword : CLASSIFICATION_PATTERN_LOCATION
    - multiplicity : 0 or 1
    - Possible values : (integer value, can be negative)
      - [extra] : The extra number that is added (or subtracted, in case the number is negative) to the position of the pattern location.

23. During the evaluation of a sequence, every position that conforms to a certain pattern (see above) is selected and then classified. However, these selected positions do not always match the positions that are needed in the outputfiles. More often than not, there is an offset of 1 or more nucleotides (even despite the CLASSIFICATION_PATTERN_LOCATION option), caused for example by the difference in counting: some start counting from 0 and others from 1. So in order to correct for these offsets in the output, we have created this option. This option adds (or subtracts) an extra number to the output locations that conform to the pattern.

    - keyword : CLASSIFICATION_PATTERN_OUTPUT
    - multiplicity : 0 or 1
    - Possible values : (Integer values, can be negative)
      - [forward][reverse] : Two extra numbers that correct the output so it conforms to the expectations. The first one is the correction for the classification of the forward strand, the second one is the correction for the classification of the reverse strand. Both are always needed when using this option. When only one is needed (e.g. we only want to classify the sequence on the reverse strand), is suffices to fill in 0 for the correction of the strand that is not used. Of course, when classification of both strands is required, then both numbers should be filled in meaningfully.

24. Indicator whether or not crossvalidation should occur. This option cannot be used together with the TEST_FILES option, and it should be used with the COMMAND option set to BUILD_MODEL. The output of the crossvalidation is put in a file and placed in the OUTPUT_DIRECTORY.

    - keyword : CROSSVALIDATION
    - multiplicity : 0 or 1
    - possible values : (integer)
        - [crossvalidation_fold] : this parameter indicates the fold of the crossvalidation. Normal numbers are 5 and 10.

25. Indicator to perform a series of crossvalidations in a row, each with a different complexity for the SVM that is used (complexity ranging from 2^{-10} till 2^4). This way the best complexity can be searched for a certain classification feature or set of classification features. This method is used by the optimalization procedure.

    - keyword : COMPLEXITY_CROSSVALIDATION
    - multiplicity : 0 or 1 (depending on CROSSVALIDATION)
    - Possible values :
        - TRUE : Use this method
        - FALSE : don't use this method (default)

26. Indicator (used for developpement and research only!!) for the maximum ratio between positive and negative training examples. This can be used to examine the optimum ratio between positive and negative training examples.

    - keyword : MAXIMUM_CROSSVALIDATION
    - multiplicity : 0 or 1 (depending on CROSSVALIDATION)
    - possible values : (integer value)
        - [ratio] : a ratio indicating how many more negative examples than positive examples must be used.

27. Applying feature selection prior to classification can drastically increase the prediction correctness (see manual for further details). This option allows the user to set the algorithm to be used by the featureselection - stub of the FunSiP program. When a valid algorithm is selected, feature selection is automatically activated. Feature selection is applied to the features extracted from the training data. After transforming the results of the feature selection algorithm to a filter (most FS algorithms just give a number between 0 and 1 to each feature), the filter will be used on both the training files themselves (and thus influence the model building and the crossvalidation) and the extracted features of the sequence to be evaluated. All the current Feature selection algorithms are taken from the WEKA machine learning library.

- keyword : FEATURE_SELECTION_TYPE
- multiplicity : 0 or 1
- Possible values : (the names of the available algorithms are provided)
  - NO_FEATURE_SELECTION : No feature selection to be used. Default setting.
  - PRECOMPUTED_FILTER : to be used when a filter is written to file and this filter should be extracted from the file and be applied to other features (be it training data or sequence data).
  - SYMMETRICAL_UNCERTAINTY : Applies the *Symmetrical uncertainty* algorithm. This algorithm is univariate.
  - CFS : Applies the *CFS* algorithm. This algorithm is multivariate.
  - INFORMATION_GAIN : Applies the *Information gain* statistical algorithm to the features. This algorithm is univariate.
  - CHI_SQUARED : Applies the *Chi squared* statistic to the features. This algorithm is univariate.
  - GAIN_RATIO : Applies the *Gain ratio* statistic to the features. This algorithm is univariate.
  - RELIEF : Applies the *Relief* algorithm. This algorithm is univariate.
  - FCBF : Applies the *FCBF* algoritgm. This algorithm is multivariate.

28. By using the FEATURE_SELECTION_TYPE option (see above), the user has the ability to decide what a algorithms to use. This option gives the user the ability to decide what to do with the filter, after the feature selection has been carried out. This may include using the filter by applying it to both the training data and the sequence data, writing the filter to a file, etc.

    - keyword : FEATURE_SELECTION_PURPOSE
    - multiplicity : 0 or 1
    - Possible values :
      - USE_FILTER : Use the filter, by applying it to both training data and sequence data.
      - WRITE_MINIMAL_FILTER : Write a filter to file. Only a minimal filter is considered: this means that only features with a non-zero score (generated by the FS algorithm) will be written to the file.
      - WRITE_FULL_FILTER : Write a filter to file. The full filter is considederd : this means that every feature and its associated score (as generated by the FS algoritm) will be written to the file.
      - LOAD_FILTER : Loads a filter from a file and applies this filter to the training data and test data.
      - USE_AND_WRITE_FULL_FILTER : A combination of USE_FILTER and WRITE_FULL_FILTER (mainly for development purposes).

29. This option allows the user to define the input-file to be used when the LOAD_FILTER value is selected in the FEATURE_SELECTION_PURPOSE option.
    - keyword : FEATURE_SELECTION_INPUT_FILE
    - multiplicity : 0 or 1 (dependent on FEATURE_SELECTION_PURPOSE)
    - Possible values : (String value)
        - [name] : name of the file containing the feature selection filter.
30. This option allows the user to define the output-file to be used when the WRITE_MINIMAL_FILTER value or WRITE_FULL_FILTER value is selected in the FEATURE_SELECTION_PURPOSE option.
    - keyword : FEATURE_SELECTION_OUTPUT_FILE
    - multiplicity : 0 or 1 (dependent on FEATURE_SELECTION_PURPOSE)
    - Possible values : (String value)
        - [name] : name of the file to be used when writing a filter to a file.
31. When applying feature selection algorithms, it may be in the best interests of the user to set an upper limit to the amount of features that will be retained by the feature selection algorithm and its generated scores. Therefore, this option was designed and it allows the user to set this upper limit.
    - keyword : FEATURE_SELECTION_MAX_FEATURES
    - multiplicity : 0 or 1 (dependent on FEATURE_SELECTION_PURPOSE)
    - Possible values : (integer value)
        - [num] : the maximum number of features to be retained after the feature selection algorithm has run.

References

[1] Sven Degroeve, Yvan Saeys, Bernard De Baets, Pierre Rouze, Yves Van de Peer, *SpliceMachine: predicting splice sites from high-dimensional local context representations*, Bioinformatics **21**, No. 8, 1332-1338 (2005).

[2] Emanuele Buratti, Francisco E. Baralle, *Influence of RNA Secondary Structure on the Pre-mRNA Splicing Process*, Molecular and Cellular Biology **24**, No. 24, 10505-10514 (2004).

[3] Michiel Van Bel, Yvan Saeys, Yves Van de Peer, *Increasing the performance of splice site prediction: A feature selection approach*, Poster at MSLB07, available at http://bioinformatcs.psb.ugent.be

[4] Saeys,Y.,Inza,I.,Larranaga,P., *A review of feature selection techniques in bioinformatics*, Bioinformatics (In Press) (2007).

[5] http://gridengine.sunsource.net/

[6] http://logging.apache.org/log4j/1.2/index.html