Submitted to *Challenges*. Pages 1 - 21.

OPEN ACCESS

Communication

An Evolutionary Optimizer of *libsvm* Models

Dragos Horvath [1,*], J.B. Brown [2], Gilles Marcou [1] and Alexandre Varnek [1]

[1] Laboratoire de Chémoinformatique, UMR 7140 CNRS – Univ. Strasbourg, 1, rue Blaise Pascal, 6700 Strasbourg, France

[2] Kyoto University Graduate School of Pharmaceutical Sciences, Department of Systems Bioscience for Drug Discovery, 606-8501, Kyoto, Japan

* Author to whom correspondence should be addressed; dhorvath@unistra.fr

Version August 7, 2014 submitted to Challenges. *Typeset by LaTeX using class file mdpi.cls*

Abstract: This User Guide describes the rationale behind, and the *modus operandi* of a Unix script-driven package for evolutionary searching of optimal *libsvm* operational parameters, leading to Support Vector Machine models of maximal predictive power and robustness. Unlike common *libsvm* parameterizing engines, the current distributions includes the key choice of best-suited sets of attributes/descriptors, next to the classical *libsvm* operational parameters (kernel choice, cost, *etc.*), allowing a unified search in an enlarged problem space. It relies on an aggressive, repeated cross-validation scheme to ensure a rigorous assessment of model quality. Primarily designed for chemoinformatics applications, it also supports the inclusion of decoy instances – for which the explained property (bioactivity) is, strictly speaking, unknown, but presumably "inactive". The package supports four different parallel deployment schemes of the Genetic Algorithm-driven exploration of the operational parameter space, for workstations and computer clusters.

Keywords: chemoinformatics; QSAR; machine learning; *libsvm* parameter optimization

1. Introduction

Support Vector Machines (SVM), such as the very popular *libsvm* toolkit [1], are a method of choice for machine learning of complex, non-linear patterns of dependence of an explained variable – here termed the "property" P – and a set (vector) of attributes (descriptors $\vec{D}$) thought to be determinants of the current P value of the instance/object they characterize. The ultimate goal of machine learning

is to unveil the relationship between the property of an object and its descriptors: $P = P(\vec{D})$, where this relationship may ideally represent a causal relationship between the attributes of the objects that are responsible for its property, or, at least, a statistically validated covariance between attributes and property. Following good scientific practice (Occam's razor), this relationship must take the simplest form that explains so-far available experimental observations. Complexifying the relationship is only justified if this leads to a significant increase of its predictive/extrapolation ability. Support Vector Machines may describe relationships of arbitrary complexity. However, controlling the underlying complexity of an SVM model may be only indirectly achieved, by fine tuning its operational parameters. Unfortunately, this is a rather counterintuitive task (one cannot easily "guess" the optimal order of magnitude for some of these parameters, not the mention the actual values), and cannot be addressed by systematic scanning of all potentially valid combinations of control parameters. Furthermore, in various domains, such as mining for Quantitative (Molecular) Structure – Property Relationships (QSPR), for which this package has been primarily designed, there are many (hundreds) of *a priori* equally appealing, alternative choices for the molecular descriptor set to be used as the input $\vec{D}$. There are many ways in which the molecular structure can be encoded under the form of numeric descriptors. While some problem-based knowledge may orient the user towards the most likely useful class of descriptors to use in order to model a given molecular property P, tens to hundreds of empirical ways to capture the needed molecular information in a bit-, integer- or real-number vector remain, and it cannot be known beforehand which of these alternative "Descriptor Spaces" (DS) would allow for the most robust learning of $P = P(\vec{D})$.

Moreover, vectors $\vec{D}$ in these various DS may be of widely varying dimension, sparseness and magnitude, meaning that some of the SVM parameters (in particular the γ coefficient controlling Gaussian or Sigmoid kernel functions) will radically differ by many orders of magnitude with respect to the employed DS. Since γ multiplies either a squared Euclidean distance, or a dot product value between two vectors $\vec{D}$ and $\vec{d}$ in the considered DS in order to return the argument of an exponential function, fitting will not focus on γ *per se*, but on a preliminary "gamma factor" `gamma` defined such that $\gamma = \texttt{gamma} / < |\vec{D} - \vec{d}|^2 >$ or, respectively $\gamma = \texttt{gamma} / < \vec{D}.\vec{d} >$. Above, the denominators represent means, over training set instance pairs, of Euclidean distance and dot product values, as required. In this way, γ is certain to adopt a reasonable order of magnitude if the dimensionless `gamma` parameters is allowed to take values within (0,10).

The goal of the herein presented software distribution is to provide a framework for simultaneous "meta"-optimization of relevant operational SVM parameters, including – and foremost – the choice of best suited descriptor spaces, out of a list of predefined choices. This assumes simultaneous fitting of categorical degrees of freedom (descriptor choice) and "classical" numerical SVM parameters, where the acceptable orders of magnitudes for some of the latter directly depend on the former DS choice. Such an optimization challenge – unfeasible by systematic search – clearly requires stochastic, nature-inspired heuristics. A simple Genetic Algorithm (GA) has been implemented here. Chromosomes are a concatenation (vector) of the currently used operational parameter values. To each chromosome, a fitness score is associated, reporting how successful SVM model building was when the respective choice of parameters was employed. Fitter setups are allowed to preferentially generate "offspring", by cross-overs and mutations, and generate new potentially interesting setup configurations. Since this

is perfectly able to deal with both categorical and continuous degrees of freedom, additional toggle variables were considered: descriptor scaling (if set, initial raw descriptors of arbitrary magnitudes will first be linearly rescaled between zero and one) and descriptor pruning (if set, the initial raw descriptor space will undergo a dimensionality reduction, by removing one of pairwise correlated descriptor columns, respectively).

Model optimization is so-far supported for both ϵ Support Vector Regression (SVR) and Support Vector Classification (SVC) problems. The definition of SVR and SVC chromosome slightly differs, since the former also includes the ϵ tube width, within which regression errors are ignored. The nature of the problem may be indicated by command line parameter.

The fitness (objective) function (SVM model quality score) to be optimized by picking the most judicious parameter combination should accurately reflect the ability of the resulting model to extrapolate/predict instances not encountered during the training stage, not its ability to (over)fit training data. In this sense, "classical" leave-$1/N$-out cross-validation (XV) already implemented in libsvm was considered too lenient, because its outcome may be strongly biased by the peculiar order of instances in the training set, dictating the subsets of instances that are simultaneously being left out. A M-fold repeated leave-$1/N$-out XV scenario, where each M-fold repeat proceeds on a randomly reordered training set item list, has been retained here. As a consequence, $M \times N$ individual SVM models are by default built at any given operational parameter choice as defined by the chromosome. However, if any of these $M \times N$ attempts fails at fitting stage (quality of fit below a user-defined threshold), then the current parameterization is immediately discarded as unfit, in order to save computer time. Otherwise, the quality score used as fitness function of the GA not only reflects the average cross-validation propensity of models at given operational setup, but also accounts for its robustness with respect to training set item ordering. At equal average XV propensity over the M trials, models that cross-validate roughly equally well irrespective of the order to of training set items are to be preferred over models which achieve sometimes very strong, other times quite weak XV success depending on how left-out items were grouped together.

For SVR models, the cross-validated determination coefficient Q^2 is used to define fitness, as follows:

1. For each XV attempt between $1...M$, consider the N models generated by permuting the left-out $1/N$ of the training set.

2. For each training instance i, retrieve the predicted property $\hat{P}_M(i)$ returned by the one specific model among the above-mentioned N which had not included i in its training set.

3. Calculate the cross-validated Root-Mean-Squared Error $RMSE_M$ of stage M, based on the deviations of $\hat{P}_M(i)$ from the experimental property of instance i. Convert this into the determination coefficient $Q^2_M = 1 - RMSE^2_M/\sigma^2(P)$, by reporting this error to the intrinsic variance $\sigma^2(P)$ of property P within the training set.

4. Calculate the mean $< Q^2_M >$ and the standard deviation σ of Q^2_M values over the M attempts. Then, fitness is defined as $< Q^2_M > -\kappa \times \sigma(Q^2_M)$, with a user-defined κ (2, by default).

In SVC, the above procedure is applied to the Balanced Accuracy (BA) of classification, defined as the mean, over all classes, of the fractions of correctly predicted members of class C out of the total number

of instances in class C. This is a more rigorous criterion than the default fraction of well-classified instances reported by libsvm (sum of well-classified within each class/training set size), in particular with highly unbalanced sets, as often the case in chemoinformatics.

Eventually, the set of $M \times N$ individual SVM models may serve as final consensus model for external predictions if the fitness score exceeds a user-defined threshold and if, of course, external descriptor sets are being provided. Furthermore, if these external sets also feature activity data, external prediction challenge quality scores are calculated on-the-fly. However, the latter are not used in the Darwinian selection process of the optimal parameter chromosome, but merely generated for further exploitation by the user. External "on-the-fly" validation also allows to eventually delete the $M \times N$ temporarily generated models, avoiding the potential storage problems that would be quick to emerge if several tens of bulky SVM model files were to be kept for each of the thousands of attempted parameterization schemes. The user is given the option to eventually explicitly regenerate the model files for one or more of the most successful parameterization schemes "discovered" by the Darwinian process, and use them outside of the context of this package.

Both the expansion of parameter space to include "meta"-variables such as the chosen DS and the pretreatment protocol of descriptors and the aim for a robust model quality estimated based on an M-fold more expensive approach than classical XV may render the mining for the optimal model quite effort-consuming. Therefore, the current distribution supports both a multi-CPU workstation version for "easy" problems (hundreds of training items within tens of candidate DS), but also parallel deployment-ready pilots using Torque, Slurm or LSF (Load Share Facility) on clusters.

2. Installation and Use

This distribution includes a set of tcsh scripts, relying on various subscripts (tcsh, awk and perl) and executables in order to perform specific tasks. This User guide assumes basic knowledge of Linux – it will not go into detailed explanations concerning fundamentals such as environment, paths, *etc.*

2.1. Installation and Environment Setup

2.1.1. Prerequisites

As a prerequisite, check whether `tcsh`, `awk` and `perl` are already installed on your system. Also, the `at` job scheduler should be enabled, both on the local workstations and on cluster nodes, as needed. In principle, our tools should not be tributary to specific versions of these prerequisite standard packages, and such dependence was never observed so far. Note, however, that `awk` (unfortunately) displays a language-dependent behavior, and may be coaxed to accept a comma as decimal separator instead of the dot, which would produce weird results (as awk always does something, and never complains). Therefore, we explicitly recommend to check the language setup of your machine – if in doubt, set your `LANG` environment variable to "en_US". Also, if you plan to machine-learn from DOS/Windows files and forgot to convert them to Unix-style, expect chaotic behavior. Install the `dos2unix` utility now, so you won't have to come back to this later.

2.1.2. Installation

Download the .tar.gz archive and extract files in a dedicated directory. It does not include the *libsvm* package, which you must install separately. Two environment variables must be added to your configuration: `GACONF`, being assigned the absolute path to the dedicated directory, and `LIBSVM`, set to the absolute path of the residence directory of *libsvm* tools. An example (for `tcsh` shells) is given in the file **EnvVars.csh** of the distribution, to be edited and added to your **.cshrc** configuration file. Adapt as needed (use *export* instead of *setenv*, in your **.profile** configuration file) if your default shell is other than `(t)csh`.

Specific executables shipped with this distribution are compiled for x86_64 Linux machines. They are all regrouped in the subdirectory `$GACONF/x86_64`, since it is assumed that your machine type, as encoded by the environment variable `$MACHTYPE`, is "x86_64". If this is not your case, you will have to create your own executable subdirectory `$GACONF/$MACHTYPE` and generate the corresponding executables in that directory. To this purpose, source codes are provided for all Fortran utilities. They can be straightforwardly recompiled:

```
foreach f ($GACONF/*.f)
g77 -O3 -o $GACONF/$MACHTYPE/$f:r $f
end
```

– please adapt to your favorite Unix shell syntax. Note – `gfortran, f77` or other Fortran compilers may do the job as well: those snippets of code are fairly robust, using standard commands.

However, *simScorer*, the executable used to calculate mean Euclidean distance and dot product values of training set descriptor vectors is a full-blown, strategic virtual screening tool of our laboratory. Its FreePascal source code cannot be released. However, we might perhaps be able to generate, upon request, the kind of executable you require.

2.1.3. Deployment-Specific Technical Parameters

As hinted in the Introduction, this distribution supports a parallel deployment of the fitness scoring of operational parameter combinations encoded by the chromosomes. Four scenarios are supported:

1. **local:** several CPUs of a workstation are used in parallel, each hosting an independent machine learning/XV attempt (the "slave" job) with parameters encoded by the current chromosome , and returning, upon completion, the fitness score associated to that chromosome to the master "pilot" script overseeing the process. Master (pilot) and slave (evaluators of the parameterization scheme quality) scripts run on the same machine – however, since the pilot sleeps for most of the time, waiting for slaves to return their results, the number of slave jobs may equal the number of CPU cores available. As the number of active slave processes decreases, the pilot will start new ones, until the user-defined maximal number of parameterization attempts has been reached.

2. **torque:** slaves estimating the quality of a parameterization scheme are being submitted on available nodes of a cluster running the torque job scheduler. The pilot runs on the front-end

(again, without significantly draining computer power) and schedules a novel slave job as soon as the number of executing or waiting slave jobs has dropped below the user-defined limit, until the user-defined maximal number of parameterization attempts has been reached. One slave job is scheduled to use a single CPU core (meaning that, like in the local version, successive XV runs are executed sequentially). Slave jobs are being allotted explicit time lapses (walltime) by the user and those failing to complete within allotted time are lost.

3. **slurm:** unlike in torque-driven deployment, the pilot script running on the front-end of the slurm-operated batch system reserves entire multi-CPU nodes for the slave jobs, not individualCPUs. Since slave jobs were designed to run on a single CPU, the slurm pilot does not directly manage them, but deploys, on each reserved node, a slave manager mandated to start the actual slave jobs on each CPU node, then wait for their completion. When the last of the slaves finished and returned results to the front-end pilot, the manager completes as well, and frees the node. The front-end pilot observes this, and reschedules another session on a free node, until the user-defined maximal number of parameterization attempts is reached. A runtime limit per node (walltime) must be specified: when reached, the manager and all so-far incomplete slave jobs will be killed.

4. **bsub:** this batch system (officially known as **LSF**, but practically nicknamed **bsub** by the name of its key job submission command) is similar to slurm, in the sense that the front-end pilot is managing multi-CPU nodes, not individual cores. However, there is no mandatory time limit per node involved. The task of the front-end pilot is to start a user-defined number of "local" workstation pilot jobs on as many nodes, keep track of incoming results, potentially reschedule local pilots if some happened to terminate prematurely, and eventually stop the local pilots as soon as the user-defined maximal number of parameterization attempts has been reached. Local pilots will continue feeding new slave jobs to the available CPUs of the node, not freeing the current node until terminated by the front-end pilots (or by failure).

IMPORTANT! Cluster-based strategies may require some customization of the default parameters. These may be always modified at command-line startup of the pilot scripts, but permanent modifications can be hard-coded by editing the corresponding ***.oppars** files ("*" meaning **torque, slurm, bsub**, respectively) in `$GACONF`. The files contain commented set instructions (`tcsh`-style, do *not* translate to your default shell syntax) and should be self-explanatory. **Some adjustments – account and queue specifications, when applicable – are mandatory.** Moreover, you might want to modify the `tmpdir` variable, which points to the local path on a node-own hard drive, on which temporary data may be written. Unless it crashed, the slave job will move relevant results back to your main work directory, and delete the rest of temporary data. Do not modify the `restart` parameter (explained further on).

2.2. User Guide: Understanding and Using the libsvm Parameter Configurator The *libsvm* parameter configurator is run by invoking, at command line, the pilot script dedicated to the specific deployment scheme supported by your hardware:

```
$GACONF/pilot_<scheme>.csh data_dir=<directory containing input
data> workdir=<intended location of results> [option=value...]  >&
logfile.msg
```

where `<scheme>` is either of the above-mentioned `local, torque, slurm, bsub`. The script does not attempt to check whether it is actually run on a machine supporting the envisaged protocol – calling `pilot_slurm.csh` on a plain workstation will result in explicit errors, while calling `pilot_local.csh` on the front-end of a slurm cluster will charge this machine with machine learning jobs, much to the disarray of other users and the ire of the system manager.

2.2.1. Preparing the Input Data Directory

The minimalistic input of a machine learning procedure is a property-descriptor matrix of the training set instances. However, one of the main goals of the present procedure is to select the best suited set of descriptors, out of a set of predefined possibilities. Therefore, several property-descriptor matrices (one per candidate descriptor space) must be provided. As a consequence, it was decided to regroup all the required files into a directory, and pass the directory name to the script, which then will detect relevant files therein – following naming conventions. An example of input directory, **D1-datadir**, is provided with the distribution. It is good practice to keep a copy of the list of training set items in the data directory, even though this will not be explicitly used. In **D1-datadir**, this is entitled **ref.smi_act_info**. It is a list of SMILES strings (column #1) of 272 ligands for which experimental affinity constants (pK_i, column #2) with respect to the D1 dopamine receptor were extracted from the ChEMBL database [2], forming the training set of a D1 affinity prediction model.

Several alternative schemes to encode the molecular information under numeric form were considered, *i.e.* these molecules were encoded as position vector in several DS. These DS include pharmacophore (PH)- and atom-symbol (SY) labeled sequences (seq) and circular fragment counts (tree,aab), as well as fuzzy pharmacophore triplets [3,4]. For each DS, a descriptor file encoding one molecule per line (in the order of the reference list) is provided: **FPT1.svm, seqPH37.svm, seqSY37.svm, aabPH02.svm, treeSY03.svm, treePH03.svm** in .svm format. Naming of these descriptor files should consist of an appropriate DS label (a name of letters, numbers and underscores – *no spaces, no dots* – unequivocally reminding you how the descriptors were generated), followed by the **.svm** extension. For example, if you have three equally trustworthy/relevant commercial pieces of soft, each offering to calculate a different descriptor vector for the molecules, let each generate its descriptors and store them as **Soft1.svm, Soft2.svm, Soft3.svm** in the data directory. Let us generically refer to these files as **DS.svm**, each standing for a descriptor space option. These should be plain Unix text files, following the SVM format convention, *i.e.* representing each instance (molecule) on a line, as:

```
SomeID 1:Value_Of_Descriptor_Element_1 2:Value_Of_Descriptor_Element_2
...  n:Value_Of_Descriptor_Element_n
```

where the values of vector element D_i refer, of course, to the current instance described by the current line. The strength of this way to render the vector $\vec{D}$ is that elements equal to zero may be omitted, so that in terms of total line lengths the explicit citing of the element number ahead of the value is more than compensated for by the omission of numerous zeros in very sparse vectors. If your software does not support writing of .svm files, it is always possible to convert a classical tabular file to .svm format, using:

```
awk '{printf "%s",$1;for (i=2;i<=NF;i++) {if ($i!=0) printf
"%d:%.3f",i-1,$i};print ""}' tabular_file_with_first_ID_column.txt
> DS_from_tabular.svm
```

where you may tune the output precision of descriptors (here %.3f). Note – there are no title/header lines in .svm files. If the tabular file does contain one, make sure to add an `NR>1` clause to the above `awk` line.

By default, .svm files do not contain the modeled property as the first field on the line, as expected from a property-descriptor matrix. Since several .svm files, each representing another DS – and presumably generated with another piece of soft, that may or may not allow the injection of the modeled property parameter into the first column – our software tools allow .svm files with first columns of arbitrary content to be imported as such into the data directory. The training set property values need to be specified separately, in a file having either the extension `.SVMreg` – for continuous properties to be modeled by regression, or `SVMclass`, for classification challenges. You may have both types of files, but only one file of a given type per data directory (their name is irrelevant, only the extension matters). For example, in **D1-datadir**, the file **ref.SVMreg** is a single-column file of associated D1 pK_i affinity values (matching the order of molecules in descriptor files). The alternative **ref.SVMclass** represents a two-class classification scheme, distinguishing "actives" of $pK_i > 7$ from "inactives". The provided scripts will properly merge property data with corresponding descriptors from .svm files during the process of cross-validated model building. Cross-validation is an automated protocol – the user needs not to provide a split of the files into training and test sets: just upload the global descriptor files to the data directory.

In addition, our package allows on-the-fly external predictions and validation of the models. This is useful in as far as, during the search stage of the optimal parameters, models generated according to a current parameter set are evaluated, then discarded, in order not to fill the disks with bulky model files that seemed promising at the very beginning of the evolutionary process, but were long since outperformed by later generations of parameter choices. Do not panic: the model files corresponding to user-picked parameter configurations may alway be recreated and kept for off-line use. However, if descriptor files (or property-descriptor matrices) of external test sets are added to the data directory, models of fitness exceeding a user-defined threshold will be challenged to predict properties for the external instances before deletion. The predictions, of course, will be kept and reported to the user, but never used in parameter scheme selection – the latter being strictly piloted by the cross-validated fitness score. Since the models may be based on either of the initially provided **DS.svm**, all the corresponding descriptor sets must also be provided for each of the considered external test sets. In order to avoid confusion with

training files, external set descriptors must be labeled as **ExternalSetName.DS.psvm**. Even if only one external set is considered, the three-part dot-separated syntax must be followed (therefore, no dots within ExternalSetName or DS, please). For example, check the external prediction files in **D1-datadir**. They correspond to an external set of dopamine D5-receptor inhibitors, encoded by the same descriptor types chosen for training: **D5.FPT1.psvm, D5.seqPH37.psvm, ...** Different termination notwithstanding, **.psvm** files are in the same **.svm** format. Whatever the first column of these files may contain, our tool will attempt to establish a correlation between the predicted property and this first column (forcibly interpreted as numeric data). If the user actually provided the experimental property values, then external validation statistics are automatically generated. Otherwise, if these experimental values are not known – this is a genuine prediction exercise – then the user should focus on the returned prediction values. Senseless external validation statistics scores may be reported if your first column may be interpreted as a number – just ignore. In the sample data provided here, the first column reports pK_i affinity constants for the D5 receptor, whereas the predicted property will be the D1 affinity. This is thus not an actual external validation challenge, but an attempt to generate computed D1 affinities for D5 ligands.

To wrap up, the input data directory must/may contain the following files:

1. A **list** of training items (optional)
2. A one-column **property** file with extensions .SVMreg (regression modeling of continuous variables) or SVMclass (classification modeling) respectively (**compulsory**)
3. **Descriptor** files, in .svm format, one per considered DS, numerically encoding one training instance per line, ordered like in the property file (**at least one DS.svm required**). The contents of their first column is irrelevant, as the property column will be inserted instead.
4. Optional **external** prediction files – for each considered external set, all the associated descriptors must be provided as **ExternalSetName.DS.psvm** files. The user has the option of inserting experimental properties to be automatically correlated with predictions in the first column of these .psvm files

2.2.2. Adding Decoys – The Decoy Directory

In chemoinformatics, it is sometimes good practice to increase the diversity of a con-generic training set (based on a common scaffold) by adding very different molecules. Otherwise, machine learning may fail to see the point that the common scaffold is essential for activity and learn the "antipharmacophore" – features that render some of these scaffold-based analogues *less* active than others. Therefore, when confronted to external predictions outside of the scaffold-based families, such models will predict all compounds *not* containing this antipharmacophore (including cosmic vacuum, matching the above condition) to be active. It makes sense to teach models that outside the scaffold-based family activity will be lost. This may be achieved by adding *presumed* diverse inactives to the training set. However, there is no experimental (in)activity measure for these – empirically, a value smaller than the measured activity of the less active genuine training set compound is considered. This assumption may, however, distort model quality assessment. Therefore, in this approach we propose an optimal compromise scenario in which, if so desired by the user, the training set may be augmented by a number of decoy instances

equal to half of the actual training set size. At each XV attempt, after training set reshuffling, a (every time different) random subset of decoys from a user-defined **decoy repository** will be added. The SVM model fitting will account for these decoys, labeled as inactives. However, the model quality assessment will ignore the decoy compounds, for which no robust experimental property is known: it will focus on the actual training compounds only. Therefore, the user should *not* manually complete the training set compounds in the data directory with decoy molecules. An extra directory – to be communicated to the pilot script using the `decoy_dir=<path of decoy repository>` command line option – should be set up, containing **DS.svm** files of an arbitrary large number of decoy compounds. Again, all the DS candidates added to the data directory must be represented in the decoy directory. The software will therefrom extract random subsets of size comparable to the actual training set, automatically inject a low activity value into the first column, and join these to the training instances, creating a momentarily expanded training set.

In case of doubt concerning the utility of decoys in the learning process, the user is encouraged to run two simulations in parallel – one employing decoys, the other not (this is the default behavior unless `decoy_dir=<path to decoy repository>` is specified). Unless decoy addition strongly downgrades model fitness scores, models employing decoys should be preferred because they have an intrinsically larger applicability domain, being confronted with a much larger chemical subspace at the fitting stage.

2.2.3. Command-line Launch of the libsvm Parameter Configurator: the Options and their Meaning

The following is an overview of the most useful options that can be passed to the pilot script, followed by associated explanations of underlying processes, if relevant:

1. `workdir=<intended location of results>` is always mandatory, for both new simulations (for which the working directory must not already exist, but will be created) and simulation restarts, in which the working directory already exists, containing preprocessed input files and so-far generated results.

2. `cont=yes` must be specified in order to restart a simulation, based on an existing working directory – in which data preprocessing is completed. Unless this flag is set (default is new run), specifying an existing working directory will result in an error.

3. `data_dir=<directory containing input data>` is mandatory for each new simulation, for it refers to the repository of basic input information, as described above. Unless `cont=yes`, failure to specify the input repository will result in failure.

4. `decoy_dir=<path of decoy repository>` as described in the previous subsection is mandatory only for new starts (if `cont=yes`, preprocessed copies of decoy descriptors are assumed to already reside in the working directory).

5. `mode=SVMclass` toggles the pilot to run in classification mode. Default is `mode=SVMreg`, *i.e.* *libsvm* ϵ-regression. Note: the extension of the property data file in the input data directory must strictly match the `mode` option value (it is advised *not* to mistype "SVMclass").

6. `prune=yes` is an option concerning the descriptor preprocessing stage, which will be described in this paragraph. This preprocessing is the main job of `$GACONF/svmPreProc.pl`, operating within `$GACONF/prepWorkDir.csh`, called by the pilot script. It may perform various operations on the brute **DS.svm**. By default, training set descriptors from the input directory are scanned for columns of near-constant or constant values. These are discarded if the standard deviation of vector element *i* over all training set instances is lower than 2% of the interval $maxD_i - minD_i$ (near-constant), or if $maxD_i = minD_i$ (constant). Then, training set descriptors are Min/Max-scaled (for each descriptor column *i*, $maxD_i$ is mapped to 1.0, while $minD_i$ maps to 0.). This is exactly what the *libsvm*-own `svm-scale` would do – now it is achieved on-the-fly, within the larger preprocessing scheme envisage here. Training set descriptors are the ones used to fix the reference minimum and maximum, further used to scale all other terms (external set .psvm files and/or decoy .svm files). These reference extremes are stored in **.pri** files in the working directory, for further use. In this process, descriptor elements ("columns") are sorted with respect to their standard deviations and renumbered. If, furthermore, the `prune=yes` option has been invoked, a (potentially time-consuming) search for pairwise correlated descriptor columns (at $R^2 > 0.7$) will be performed, and one member of each concerned pair will be discarded. However, it is not *a priori* granted that either Min/Max scaling or pruning of correlated columns will automatically lead to better models. Therefore, scaling and pruning are considered as degrees of freedom of the GA – the final model fitness is the one to decide whether scaled, pruned, scaled and pruned or plain original descriptor values are the best choice to feed into the machine learning process. At preprocessing stage, these four different versions (scaled, pruned, scaled and pruned or plain original) are generated, in the working directory, for each of the .svm and .psvm files in input or decoy folders. If pruning is chosen, but for a given DS the number of cross-correlated terms is very low (less than 15% of columns could be discarded in the process), then pruning is tacitly ignored – the two "pruned" versions are deleted because they would likely behave similarly to their parent files. For each of the processed descriptor files – now featuring property values in the first column, as required by `svm-train` – the mean values of descriptor vector dot products, respectively Euclidean distances are calculated and stored in the working directory (file extensions **.EDprops**). For large training sets (> 500 instances), these means are not taken over all the $N(N-1)/2$ pairs of descriptor vectors, but are calculated on hand of a random subset of 500 instances only.

   IMPORTANT! The actual descriptor files used for model building – and expected as input for model prediction – are, due to preprocessing, significantly different from the original **DS.svm** generated by yourself in the input folder. Therefore, any attempt to use generated models in a stand-alone context, for property prediction, must take into account that any input descriptors must first be reformatted in the same way in which an external test .psvm file is being preprocessed. Calling the final model on brute descriptor files will only produce noise. All the information required for preprocessing is contained in .pri files. Suppose, for example, that Darwinian evolution in *libsvm* parameter space showed that the so-far best modeling strategy is to use the pruned and scaled version of DS.svm. In that case, in order to predict properties of external instances described in **ExternalDS.svm**, one would first need to call:

```
$GACONF/svmPreProc.pl ExternalDS.svm selfile=DS_pruned.pri
scale=yes output=ReadyToUseExternalDS.svm
```

where the required **DS_pruned.pri** can be found in the working directory. If original (not scaled) descriptors are to be used, drop the `scale=yes` directive. If pruning was not envisaged (or shown not to be a good idea, according to the undergone evolutionary process), use the **DS.pri** selection file instead. The output .svm is now ready to used for prediction by the generated models.

7. `wait=yes` instructs the pilot script to generate the new working directory and preprocess input data, as described above, and then stop rather than launching the GA. The GA simulation may be launched later, using the `cont=yes` option.

8. `maxconfigs=<maximal number of parameter configurations to be explored>` – by default, it is set to 3000. It defines the amount of effort to be invested in this search of the optimal parameter set, and should take into account the number of considered descriptor spaces impacting on the total volume of searchable problem space. However, not being able to guess the correct `maxconfigs` values is not a big problem. If the initial estimate seems to be too high, no need to wait for the seemingly endless job to complete: one may always trigger a clean stop of the procedure by creating a file (even an empty one is fine) named **stop_now** in the working directory. If the estimate was too low, one may always apply for more number crunching by restarting the pilot with options `workdir=<already active working directory> cont=yes maxconfigs=<more than before>`

9. `nnodes=<number of "nodes" on which to deploy simultaneous slave jobs>` is a possibly misleading name for a context-dependent parameter. For the `slurm` and `LSF` contexts, this refers indeed to the number of multi-CPU machines (nodes) required for parallel parameterization quality assessments. Under the `torque` batch system and on local workstations, it actually stands for the number of CPU cores dedicated to this task. Therefore, the `nnodes` default value is context-dependent: 10 for `slurm` and `LSF/bsub`, 30 for `torque` and equal to the actual number of available cores on local workstations.

10. `lo=<"leave-out" XV multiplicity>` is an integer parameters greater or equal to two, encoding the number of folds into which to split the training set for XV. By default, `lo=3`, meaning that 1/3 of the set is iteratively kept out for predictions by a model fitted on the remaining 2/3 of training items. Higher `lo` values make for easier XV, as a larger part of data is part of the momentary training set, producing a model having "seen" more examples and therefore more likely to be able to properly predict the few remaining 1/lo test examples. Furthermore, since `lo` obviously gives the number of model fitting jobs needed per XV cycle, higher values translate to proportionally higher CPU efforts. If your data sets are small, so that 2/3 of it would likely not contain enough information to support predicting the left-out 1/3, you may increase lo up to 5. Do not go past that limit – you are only delusioning yourself by making XV too easy a challenge, and consume more electricity, atop of that.

11. `ntrials=<number of repeated XV attempts, based on randomized regrouping of kept and left-out subsets>` dictates how many times (12, by default) the leave-1/lo-out procedure has to be repeated. It matches the formal parameter M used in Introduction in order to define the model fitness score. The larger `ntrials`, the more robust the fitness score (the better the guarantee that this was not some lucky XV accident due to a peculiarly favorable regrouping of kept *vs.* left-out instances.) However, the more time-consuming the simulation will be, as the total number of fitted local models during XV equals `ntrials` × `lo`. A possible escape from this dilemma of quick *vs.* rigorous XV is to perform a first run over a large number (`maxconfigs`) of parameterization attempts, but using a low `ntrials` XV repeat rate. Next, select the few tens to hundreds of best-performing parameter configurations visited so far, and use them (see option `use_chromo` below) to rebuild and reestimate the corresponding models, now at a high XV repeat rate.

12. `use_chromo=<file of valid parameter configuration chromosomes>` is an option forcing the scripts to create and assess models at the parameter configurations from the input file, rather than using the GA to propose novel parameter setup schemes. These valid parameter configuration chromosomes have supposedly emerged during a previous simulation – therefore, the `use_chromo` option implicitly assumes `cont=yes`, *i.e.* an existing working directory with preprocessed descriptors. As will be detailed below, the GA-driven search of valid parameter configurations creates, in the working directory, two result files: **done_so_far**, reporting every so-far assessed parameter combination and the associated model fitness criteria, and **best_pop**, a list of the most diverse setups among the so-far most successful ones.

There are many more options available, which will not be detailed here because their default values rarely need to be tampered with. Geeks are encouraged to check them out, in comment-adorned files ending in ***pars** provided in `$GACONFIG`. There are common parameters (**common.pars**), deployment-specific parameters in ***.oppars** files, and model-specific (*i.e.* regressions-specific *vs.* classification-specific) parameters **SVMreg.pars, SVMclass.pars**. The latter concern two model quality cutoffs

1. `minlevel` – only models better than this, in terms of fitness scores, will be submitted to external prediction challenges.

2. `fit_no_go` represents the minimal performance at fitting stage, for every local model built during the XV process. If (by default) a regression model fitting attempt fails to exceed a fit R^2 value (or a classification model fails to discriminate at better BA), then hope to see this model succeed in terms of XV is low. In order to save time, the current parameterization attempt is aborted – the parameter choice is obviously wrong – and time is saved by reporting a fictitious, very low fitness score associated to the current parameter set.

Default choices for regression models refer to correlation coefficients, and are straightforward to interpret. However, thresholds for classification problems depend on the number of classes. Therefore, defaults in **SVMclass.pars** correspond not to absolute balanced accuracy levels, but to fractions of non-random BA value range (parameter value 1.0 maps to a BA cutoff value of 1, while parameter value zero maps to the baseline BA value, equaling the reciprocal of the class number).

2.2.4. Defining the Parameter Phase Space

As already mentioned, the model parameter space to be explored features both categorical and continuous variables. The former include DS and *libsvm* kernel type selection, the latter cover ϵ (for SVMreg, *i.e.* ϵ-regression mode only), cost γ and $coeff0$ values. In general, as well as in this case, GAs typically treat continuous variables like a user precision-dependent series of discrete values, within a user-defined range. The GA problem space is defined in two `mode`-dependent setup files in `$GACONF`: **SVMreg.rng** and `SVMclass.rng`, respectively. They feature one line for each of the considered problem space parameters, except for the possible choices of DS, which are not available by default. After the preprocessing stage, when standardized **.svm** files were created in the working directory, the script will make a list of all the available DS options, and write it out as the first line of a local copy (within the working directory) of the **<mode>.rng** file. Then it will concatenate the default `$GACONF/<mode>.rng` to the latter. This local copy is the one piloting the GA simulation, and may be adjusted whenever the defaults from `$GACONF/<mode>.rng` seem inappropriate. To do so, first invoke the pilot script with the data repository, a new working directory name, desired mode and option `wait=yes`. This will create the working directory, uploading all files and generating the local **<mode>.rng** file, then stop. Edit this local file, then re-invoke the pilot on the working directory, with option `cont=yes`.

The syntax of **.rng** files is straightforward. Column one designs the name of the degree of freedom defined on the current line. If this degree of freedom is categorical, the remaining fields on the line enumerate the values it may take. For example, line #1 is a list of descriptor spaces (including their optinally generated "pruned" versions) found in the working directory. The parameter "scale" chooses whether those descriptors should be used in their original form, or after Min/Max scaling (in clear, if scale is "orig", then DS.orig.svm will be used instead of DS.scaled.svm). The "kernel" parameter picks the kernel type to be used with *libsvm*. Albeit a numeric code is used to define it on the `svm-train` command line (0- linear, 1 - 3^{rd} order polynomial, 2 - Radial Basis Function, 3- Sigmoid), on the .rng file line, the options were prefixed by "k" in order to let the soft handle this as a categoric option. The "ignore-remaining" keyword on the kernel line is a directive to the GA algorithm to ignore the further options γ and $coeff0$ if the linear kernel choice k0 is selected. This is required for population redundancy control: two chromosomes encoding identical choices for all parameters except γ and $coeff0$ do stand for genuinely different parameterization schemes, leading to models of different quality – *unless* the kernel choice is set to linear, which is not sensitive to γ and $coeff0$ choices, leading to exactly the same modeling outcome. All .rng file lines having a numeric entry in column #2 are by default considered to encode continuous (discretized) variables. Such lines must obligatorily have 6 fields (besides #-marked comments): variable name, absolute minimal value, preferred minimal value, preferred maximum, absolute maximum and, last, output format definition implicitly controlling its precision. The distinction between absolute and preferred ranges is made in order to allow the user to focus on a narrowest range assumed to contain the targeted optimal value, all while leaving an open option for the exploration of larger spaces: upon random initialization or mutation of the variable, in 80% of cases the value will be drawn within the "preferred" boundaries, in 20% of cases within the "absolute" boundaries. The brute real value is then converted into a discrete option

according to the associated format definition in the last column – it is converted to a string representation using `sprintf("<format representation>", value)`, thus rounded up to as many decimal digits as mentioned in the decimal part of its format specifier. For example, the cost parameter spanning a range of width 18, with a precision of 0.1 may globally adopt 180 different values. Modifying the default "4.1" format specifier to "4.2" triggers a 10-fold increase of the phase space volume to be explored, in order to allow for a finer scan of cost.

2.2.5. The Genetic Algorithm

A chromosome will be rendered as a concatenation of the options picked for every parameter, in the order listed in the **.rng** file. They are generated on-the-fly, during the initialization phase of slave jobs, and not in a centralized manner. When the pilot submits a slave job, it does not impose the parameter configuration to be assessed by the slave, but expects the slave to generate such a configuration, based on the past history of explored configurations, and assess its goodness. In other words, this GA is *asynchronous*. Each so-far completed slave job will report the chromosome it has assessed, associated to its fitness score, by concatenation to the end of the **done_so_far** file in the working directory. The pilot script, periodically waking up from sleep to check the machine work load, will also verify whether new entries were meanwhile added to **done_so_far**. If so, it will update the population of so-far best, non redundant results in the working directory file **best_pop**, by sorting **done_so_far** by its fitness score, then discarding all chromosomes that are very similar to slightly fitter essays and cutting the list off at fitness levels of 70% of the best-so-far encountered fitness score (this "Darwinian selection" is the job of `awk` script `$GACONF/chromdiv.awk`).

A new slave job will propose a novel parameter configuration by generating offspring of these "elite" chromosomes in **best_pop**. In order to avoid reassessing an already seen, but not necessarily very fit, configuration, it will also read the entire **done_so_far** file, in order to verify that the intended configuration is not already in there. If so, genetic operators are again invoked, until a novel configuration emerges. However, neither **best_pop** nor **done_so_far** do not yet include configurations that are currently under evaluation on remote nodes. The asynchronous character of the GA does not allow absolute guarantees that it will be a perfectly self-avoiding search, albeit measures have been taken in this sense.

Upon lecture of **best_pop** and **done_so_far**, the parameter selector of the slave job (`awk` script `$GACONF/make_childrenX.awk`) may randomly decide (with predefined probability) to perform a "cross-over" between two randomly picked partners from **best_pop**, a "mutation" of a randomly picked single parent, or a "spontaneous generation" (random initialization, ignoring the "ancestors" in **best_pop**). Of course, if **best_pop** does not contain at least two individuals, the cross-over option is not available, and if **best_pop** is empty, at the begininng of the simulation, then the mutation option is unavailable as well: initially, all parameter configurations will be issued by "spontaneous generation".

2.2.6. Slave Processes – Startup, Learning Steps, Output Files and their Interpretation

Slave jobs run in temporary directories on the local file system of the nodes. A temporary directory is being assigned an unambiguous attempt ID, appended to its default name "attempt". If the slave

job successfully completes, this attempt folder will be, after cleaning of temporary files, copied back to the working directory. `$GACONF` also contains a sample working directory **D1-workdir**, displaying typical results obtained from regression-based learning from **D1-datadir** – you may browse through it in order to get acquainted with output files. The attempt ID associated to every chromosome is also reported in the result file **done_so_far**. Inspect `$GACONF/D1-workdir/done_so_far`. Every line thereof is formally divided by an equal sign in two sub-domains. The former encodes the actual chromosome (field #1 stands for chosen DS, #2 for the descriptor scaling choice, ...). Fields at right of the "=" report output data: attempt ID, fitting score and, as last field on line, the XV fitness score, $< Q^2 > -2\sigma(Q^2)$ for regression, $< BA_{XV} > -2\sigma(BA_{XV})$ for classification problems as defined in Introduction. The fitting score is calculated, for completeness, as the "mean-minus-two-sigma" of fitted correlation coefficients and fitted balanced accuracy. Please do not allow yourself to be confused by "fitting" (referring to statistics of the `svm-train` model fitting process, and concerning the instances used to build the model), and Darwinian "fitness" (defined on the basis of XV results). The fitting score is never used in the Darwinian selection process – it merely serves to inform the user about the loss of accuracy between fitted and predicted/cross-validated property values.

IMPORTANT! Since every slave job will try to append its result line to **done_so_far** as soon as it has completed calculations, chance may have different jobs on different nodes – each "seeing" the working directory on a `NFS`-mounted partition – attempt to simultaneously write to **done_so_far**. This may occasionally lead to corrupted lines, not matching the description above. Ignore them.

Yet, before results are copied back to the working directory, the actual work must be performed. This begins by generating a chromosome, unless the `use_chromo` option instructs the job to apply an externally imposed setup. After the chromosome is generated and stored in the current attempt folder, it first must be interpreted. On one hand, the chromosome contains descriptor selection information. The property-descriptor matrix matching the name reported in the first field of the chromosome, and more precisely its scaled or non-scaled version, as indicated by the second field, will be the one copied to the attempt folder, for further processing. The remaining chromosome elements must be translated into the actual *libsvm* command line options. In this sense, the actual ϵ value for regression is calculated by multiplying the `epsilon` parameter in the chromosome by the standard deviation of the training property values. In this way, ϵ, representing 10 to 100% of the natural deviation of the property, implicitly has the proper order of magnitude and proper units. Likewise, the chromosome `gamma` parameter will be converted to the actual γ value, dividing by the mean vector dot product (for kernel choices k1 or k3), or by the mean Euclidean distance (kernel choice k2), respectively. Since the cost parameter is tricky to estimate, even in terms of magnitude orders, the chromosome features a log-scale `cost` parameter, to be converted into the actual cost by taking the exponential thereof. This, and also the stripping off of the "k" prefix in the kernel choice parameter, are also performed at the chromosome interpretation stage, `mode`-dependently carried out by dedicated "decoder" `awk` scripts **chromo2SVMreg.awk** and **chromo2SVMclass.awk**, respectively. At the end, the tool creates, within the attempt folder, the explicit option line required to pilot `svm-train` according to the content of the chromosome. This option line is stored in a one-line file **svm.pars** – check out, for example, `$GACONF/D1-workdir/attempt.11118/svm.pars`, the option set that produced the fittest models so-far.

A singled-out property-descriptor matrix and a command-line option set for `svm-train` are the necessary and sufficient prerequisites to start XV. For each of the `ntrials` requested XV attempts, the order of lines in the property-descriptor matrix is randomly changes. The reordered matrix is then split into the requested `lo` folds. XV is explicit, and is not using the own facility of `svm-train`. Iteratively, `svm-train` is used to learn a local model on the basis of all but the left-out fold. If adding of decoys is desired, random decoy descriptor lines (of the same type, and having undergone the same scaling strategy as training set descriptors) are added to make up 50% of the local learning set size (the `lo`-1 folds in use). These will differ at each successive XV attempt, if the pool of decoys out of which they are drawn is much larger than the required half of training set size).

These local models are stored, and used to predict the entire training set, in which, however, the identities of local "training" and locally left-out ("test") instances are known. These predictions are separately reported into "train" and "test" files. The former report, for each instance, fitted property values by local models having used the instance for training. The latter report predicted property values by local models not having used it for training.

Quality of fit is checked on-the-fly, and failure to exceed a predefined fitting quality criterion triggers a premature exit of the slave job, with a fictitious low fitness score associated to the current chromosome – see the `fit_no_go` option. In such a case, the attempt subdirectory is deleted, and a new slave job is launched instead to the defunct one.

Results in the attempt subdirectories of **D1-workdir** correspond to the default 12 times repeated leave-1/3-out XV schemes. This means that $12 \times 3 = 36$ local models are generated. For each D1 ligand, there are exactly 12 of these local models that were not aware of that structure when they were fitted, and 24 others that did benefit from its structure-property information upon building.

Files **final_train.pred.gz** in attempt subdirectories (provided as compressed .gz) report, for each instance, the 24 "fitted" affinity values returned by models having used it for fitting (it is a 26-column file, in which column #1 reports current numbering and #2 contains the experimental affinity value). Reciprocally, the 14-column files **final_test.pred.gz** report the 12 prediction results stemming from models not encountering instances at fitting stage. Furthermore, **consens_train.pred.gz** and **consens_test.pred.gz** are "condensed" versions of the previous, in which the multiple predictions per line have been condensed to their mean (column #2) and standard deviations (column #3), column #1 being the experimental property.

Eventually, the **stat** file found in each attempt subdirectory provides detailed statistics about fitting and prediction errors in terms of root-mean-squared error, maximal error and determination coefficients. In a classification process, correctly classified fractions and balanced accuracies are reported. Lines labeled "local_train" compare every fitted value column from **final_train**, to the experimental data, whilst "local_test" represent local model-specific XV results. By extracting all the lines "local_test:r_squared", actually representing the XV coefficients of every local model, and calculating the mean and standard deviations of these values, one may recalculate the fitness score of this attempt. The **stat** file lines labeled "test" and "train" compare the consensus means as reported in the **consens_*.pred** files to the experiment. Note: the determination coefficient "r_squared" between the mean of predictions and experiment tends to exceed the mean of local determination coefficients, reporting individual

performances of local models. This is the "consensus" effect, not exploited in fitness score calculations that rely on the latter mean of individual coefficients, penalized by twice their standard deviations.

Last but not least, this model building exercise had included an external prediction set of D5 ligands, for which the current models were required to return a prediction of their D1 affinities. External prediction is enabled as soon as model fitness exceeds the user-defined threshold `minlevel`. This was the case for the best model corresponding to `$GACONF/D1-workdir/attempt.11118/`. External prediction file names are a concatenation of external set name ("D5"), the locally used DS ("treePH03_pruned"), the descriptor scaling status ("scaled") and the extension ".pred". As all these external items are, by definition, "external" to all the $12 \times 3 = 36$ local models (no checking is performed in order to detect potential overlaps with the training set), the 38-column external prediction file will report the current numbering (column #1), the data reported in the first field of the .psvm files in column #2 (here, the experimental D5 pK_i values), followed by 36 columns of individual predictions of the modeled property (the D1 affinity values), by each of the local models. Since the external prediction set .psvm files had been endowed with numeric data in the first field of each line, the present tool assumes these to be corresponding experimental property values, to be compared to the predictions. Therefore, it will take the consensus of the 36 individual D1 affinity predictions for each item, and compare them to the experimental field. The results of external prediction challenge statistics are reported in the **extval** file of the attempt subdirectory. First, the strictest comparison consists in calculating the RMSE between experimental and predicted data, and the determination coefficient – these results are labeled "Det" in the **extval** report. However, external prediction exercises may be challenging, and sometimes useful even if the direct predicted-experimental error is large. For example, predicted values may not quantitatively match experiment, but happen to be all offset by a common value or, in the weakest case, nevertheless follow some linear trend – albeit of arbitrary slope and intercept. In order to test either of these hypotheses, **extval** also reports statistics for (a) the predicted *vs.* experimental regression line at fixed slope of 1.0, but with free intercept, labeled "FreeInt", and (b) the unconstrained, optimal linear predicted-experimental correlation that may be established, labeled "Corr".

In this peculiar case, **extval** results seem unlikely to make any sense, because the tool mistakenly takes the property data associated to external compounds (D5 affinities) for experimental D1 affinities, to be confronted with their predicted alter-egos. Surprise – even the strict Determination statistics are robustly positive on the fact that predicted D1 pK_i values quantitatively match experimental D5 pK_i values. This is due to the very close biological relatedness of the two targets, which do happen to share a lot of ligands. Indeed, 25% of the instances of the D5 external set were also reported ligands of D1, and, as such, part of the model training set. Yet, 3 out of 4 D5 ligands were genuinely new. This notwithstanding, their predicted D1 affinities came close to observed D5 values – a quite meaningful result confirming the extrapolative prediction abilities of the herein built model.

At this point, local SVM models and other temporary files are deleted, the chromosome associated to it attempt ID, fitting and fitness scores is being appended to **done_so_far**, and the attempt subfolder now containing only setup information and results is moved from its temporary location on the cluster back to the working directory (with exception of the workstation-based implementation, when temporary and final attempt subdirectory location are identical). The slave job successfully exits.

2.2.7. Reconstruction and Off-package Use of Optimally Parameterized *libsvm* Models

As soon as the number of processed attempts reaches `maxconfigs`, the pilot script stops rescheduling new slave jobs. Ongoing slave processes are allowed to complete – therefore, the final number of lines in **done_so_far** may slightly exceed `maxconfigs`. Before exiting, the pilot will also clean the working directories, by deleting all the attempt sub-folders corresponding to the less successful parameterization schemes, which did not pass the selection hurdle and are not listed in **best_pop**. A trace of their statistical parameters is saved in a directory called "loosers".

The winning attempt sub-folders contain all the fitted and cross-validated/predicted property tables and statistics reports, but not the battery of local SVM models that served to generate them. These were not kept, because they may be quite bulky files which may not be allowed to accumulate in large numbers on the disk (at the end of the run, before being able to decide which attempts can be safely discarded, there should have been `maxconfigs` × `lo` *times* `ntrials` of them). However, they – or, actually, equivalent (recall that local model files are tributary to the random regrouping of kept *vs.* left-out instances at each XV step) – model files can be rebuilt (and kept) by rerunning the pilot script with the `use_chromo` option pointing to a file with chromosomes encoding the desired parameterization scheme(s). If `use_chromo` points to a multi-line file, the rebuilding process may be parallelized: as the slave jobs proceed, new attempt sub-folders – now each containing `lo` *times* `ntrials` *libsvm* model files – will appear in the working directory (but not necessarily in the listing order of `use_chromo`).

IMPORTANT! The file of preferred setup chromosomes should only contain the parameter fields, but not the chromosome evaluation information. Therefore, you cannot use **best_pop** *per se*, or ‘`head -top`‘ of **best_pop** as `use_chromo` file, unless you first remove the right-most fields, starting at the "equal" separator:

```
head -top best_pop| sed 's/=.*//' > my_favorite_chromosomes.lst
$GACONF/pilot_<scheme>.csh workdir=<working directory used for GA
run> use_chromo=my_favorite_chromosomes.lst
```

Randomness at learning/left-out splitting stages may likely cause the fitness score upon refitting to (slightly) differ from the initial one (on the basis of which the setup was considered interesting). If the difference is significant, it means that the XV strategy was not robust enough – *i.e.* was not repeated sufficiently often (increase `ntrials`).

Local model files can now be used for consensus predictions, and the degree of divergence of their prediction for an external instance may serve as prediction confidence score [5]. Remember that `svm-predict` using those model files should be called on preprocessed descriptor files, as outlined in the Command-Line Launch subsection.

Alternatively, you may rebuild your *libsvm* models manually, using the `svm-train` parameter command line written out in successful attempt sub-folders. In this case, you may tentatively rebuild the models on your brute descriptor files, or on `svm-scale`-preprocessed descriptor files – unless the successful recipe requires pruning.

3. Conclusions

This parameter configurator for *libsvm* addresses several important aspects in SVM model building, in view – but not exclusively dedicated to – chemoinformatics applications.

First, the approach co-opts an important decision making of the model building process: the choice of descriptors/attributes, and their best preprocessing strategies, into the optimization procedure. This is important, because the employed descriptor space is a primordial determinant of modeling success, and determines the optimal operational parameters of *libsvm*.

Next, an aggressive, repeated cross-validation scheme, introducing a penalty proportional to the fluctuation of cross-validation propensity on the local grouping of learning *vs.* left-out instances is used to select an operational parameter set leading to a model of maximal robustness, not to a model owing its apparent quality to a lucky ordering of instances in the training set. This fitness score is in our opinion a good estimator of the extrapolative predictive power of the model.

Furthermore, the approach allows decoy instances to be added in order to expand learned problem space zone, without however adding uncertainty to the reported statistics (statistics of decoy-free and decoy-based models are directly comparable, because they both focus on the actual training instances and their confirmed experimental properties, ignoring "presumed" inactive decoy property values).

Last but not least, the approach is versatile, covering both regression and classification problems and supporting a large variety of parallel deployment schemes. It was, for example, successfully used to generate very large (> 9000 instances) dataset-based chemogenomics models [6].

BA – Balanced Accuracy (of classification, the mean of sensitivity and specificity) DS – Descriptor Space, GA – Genetic Algorithm, SVM – Support Vector Machine, SVR – Support Vector Regression, SVC – Support Vector Classification, RMSE – Root Mean Squared Error, XV - Cross-validation.

The authors thank the High Performance Computing centers of the Universities of Strasbourg, France, and Cluj, Romania, for having hosted a part of the herein reported development work.This work was also supported in part by a Grant-in-Aid for Young Scientists from the Japanese Society for the Promotion of Science (Kakenhi (B) 25870336).This research was additionally supported by the Funding Program for Next Generation World-Leading Researchers as well as the CREST program of the Japan Science and Technology Agency.

References

1. Chang, C.C.; Lin, C.J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* **2011**, *2*, 27:1–27.
2. Gaulton, A.; Bellis, L.J.; Bento, A.P.; Chambers, J.; Davies, M.; Hersey, A.; Light, Y.; McGlinchey, S.; Michalovich, D.; Al-Lazikani, B.; Overington, J.P. ChEMBL: a large-scale bioactivity database for drug discovery. *Nucl. Ac. Res.* **2011**, *40*, D1100–D1107.

3. Ruggiu, F.; Marcou, G.; Varnek, A.; Horvath, D. Isida Property-labelled Fragment Descriptors. *Molecular Informatics* **2010**, *29*, 855–868.
4. Bonachera, F.; Parent, B.; Barbosa, F.; Froloff, N.; Horvath, D. Fuzzy Tricentric Pharmacophore Fingerprints. 1 - Topological Fuzzy Pharmacophore Triplets and adapted Molecular Similarity Scoring Schemes. *J. Chem. Inf. Model.* **2006**, *46*, 2457–2477.
5. Horvath, D.; Marcou, G.; Varnek, A. Predicting the Predictability: A Unified Approach to the Applicability Domain Problem of QSAR Models. *J. Chem. Inf. Model.* **2009**, *49*, 1762–1776.
6. Brown, J.B.; Okuno, Y.; Marcou, G.; Varnek, A.; Horvath, D. Computational chemogenomics: Is it more than inductive transfer? *J. Comput.-Aided Mol. Des.* **2014**, *28*, 597–618.