|  |
| --- |
| The University of North Carolina at Chapel Hill ESHELMAN SCHOOL OF PHARMACY |
| Chembench |
| Developer Guide V1.3 |
| Updated June 30 2011 |
| **Molecular Modeling Laboratory** |
|  |

|  |
| --- |
|  |

Contents

[Introduction 4](#_Toc297208803)

[Getting started with the Chembench code 5](#_Toc297208804)

[First: 5](#_Toc297208805)

[Tools: 5](#_Toc297208806)

[Contacts: 5](#_Toc297208807)

[Technologies: 5](#_Toc297208808)

[References: 6](#_Toc297208809)

[Building the code: 6](#_Toc297208810)

[Running a local server: 7](#_Toc297208811)

[Debugging: 7](#_Toc297208812)

[Glossary 8](#_Toc297208813)

[System Overview 9](#_Toc297208814)

[Web Interface, validation, servlets, navigation 9](#_Toc297208815)

[Job Management 9](#_Toc297208816)

[Accessory Classes 9](#_Toc297208817)

[External Programs Interface 10](#_Toc297208818)

[External Programs 10](#_Toc297208819)

[Persistence 10](#_Toc297208820)

[JSPs 11](#_Toc297208821)

[Actions and struts.xml 13](#_Toc297208822)

[Workflows and Running External Programs 14](#_Toc297208823)

[Queues and WorkflowTasks 15](#_Toc297208824)

[Database, Persistence, and Hibernate 16](#_Toc297208825)

[File and Directory Layout 18](#_Toc297208826)

[User Directories: 18](#_Toc297208827)

[Datasets 19](#_Toc297208828)

[Visualization Flash Application 19](#_Toc297208829)

[Modeling 19](#_Toc297208830)

[Predictions 20](#_Toc297208831)

[SMILES Prediction 21](#_Toc297208832)

[Dataset Prediction 21](#_Toc297208833)

[Users and Administration 22](#_Toc297208834)

[Data File Formats 23](#_Toc297208835)

[.SDF files 23](#_Toc297208836)

[.x files 24](#_Toc297208837)

[.mz files 25](#_Toc297208838)

[.act files 25](#_Toc297208839)

# Introduction

What's the purpose of Chembench? Depending on whom you ask, you'll get different answers.

It could be

* A place where people in the MML build and store their predictors
* A way to share our modeling methodology with the world
* A way for users to assess the toxicity of a chemical
* A way for users to tell if a chemical would make a good drug for the disease they're studying
* A way for users to feed a bunch of random chemicals into models and see what they might be good for
* A warehouse of chemicals and assay data, used for creating and visualizing modeling datasets
* A place to compare prediction results from several modeling methods, ran with different parameters
* A generalized cheminformatics toolkit

And so on. The system already has good fundamental functionality and people inside and outside the lab are using it. There are lots of ideas on how to expand the system and improve on it. But before you go adding on new functions, it's important to understand how the existing ones work. That's what this document is for: to lay out the standards of the current system. If you make your code fit with what already exists, you'll avoid a *lot* of work in debugging. Even better, people will be able to expand on your code for years to come.

This document is kept in the version control system. It's in there for a good reason: You should update it! As the system grows, new features will be added, and architecture changes will happen. This is a simple document, it's easy to update. So, make sure it accurately reflects the Chembench system.

# Getting started with the Chembench code

## First:

Get a feel for chembench by looking around the site. Make a user account. Look through the public datasets from the My Bench page. Try running modeling jobs with different methods from the Modeling page, and look at the results. Run a prediction or two.

## Tools:

Get the latest version of Eclipse (Java EE). Get a Subversion plugin for it -- Subversive and SubClipse both work fine. Open Eclipse, change to the "SVN Repository Exploring" perspective, and add the location "svn://chembench.mml.unc.edu". Open CECCR-QSAR, right click on "trunk", and click Check Out. You now have the Chembench code on your computer. Change the perspective to Java and you should be able to see the code.

To access the repository, you will need to be on campus or on VPN (<https://vpn.unc.edu>).

In order to commit changes, you will need a username and password for the repository. This can be set by accessing the Chembench production machine; look under /usr/local/ceccr/svn/ceccr/conf/passwd.

It is assumed you have some familiarity with Linux tools for editing files on the production and development Linux servers, such as emacs or vi, and some basic shell scripting. An ssh client such as PuTTY and an scp / sftp client such as WinSCP will be essential as well. As with any coding, knowing a good text editor with regular expression support (EditPlus, Notepad++, Komodo Editor, JEdit, etc.) will be important. Excel is also very useful for editing and validating various data files.

## Contacts:

Major system administration tasks, especially those involving hardware or LSF, are handled by Steve Fishback ([fishback@email.unc.edu](mailto:fishback@email.unc.edu)).

There is an Oracle database which will eventually house compound data as well as the rest of our database. The contact for that is Jim Glasson (<glasson@email.unc.edu>).

For descriptor generation, we interact with two companies. MolconnZ descriptors come from eduSoft; the person to contact is Dr. David Haney (<haney@edusoft-lc.com>). Dragon descriptors come from Talete; the contact point is [info@talete.mi.it](mailto:info@talete.mi.it) .

## Technologies:

Chembench uses the following technologies:

* Java
* JSPs
* Struts 2 (MVC). See the *Struts 2 In Action* book.
* Hibernate (Persistence). See *Java Persistence with Hibernate*
* Javascript (UI); Struts 2 elements are built on Dojo
* Python / Perl scripts
* Shell scripts
* Flash (dataset visualization)
* Java applets (3D view of molecules and sketching of molecules for prediction)
* R (Random Forest and dataset visualization)
* C++ (k-Nearest Neighbors and datasplit packages)
* MySQL (database)
* Tomcat (webserver)
* Subversion (version control)

Think of it as a resume builder ☺

## References:

In the repository, under the documentation/ directory, you will find two other documents of use to developers: an install guide and a database design plan.

Other references:

* Random Forest guide (<http://cran.r-project.org/web/packages/randomForest/randomForest.pdf>)
* Compound standardization best practices (Trust but verify, <http://www.ncbi.nlm.nih.gov/pubmed/20572635>)

## Building the code:

On each server, there is a script "build.sh", stored at /usr/local/ceccr/deploy. There are two servers: one for development, and one for production. While they use the same code base, they have separate file systems and database data.

When you run build.sh, it will

* set environment variables for programs Chembench needs in order to operate,
* stop the Tomcat server,
* check out the latest version of Chembench from the Subversion repository,
* read the config file appropriate to the development or production machine (depending)
* compile it using "build.xml",
* move the compiled WAR file to Tomcat's webapps/ directory, and
* restart Tomcat.

Builds on the production server will take longer as they do a clean build every time.

Note that many of the processes involved in the commit / build cycle are lazy; wait a few seconds after committing code before building, and a few seconds after building before testing to ensure best outcomes.

Before you post a build on the production site, check:

1. the strutsdebug.log and usage.log files to see if there are active users (try not to interrupt them)
2. the Jobs / My Bench page to see if there are any jobs that would be affected by the build. (For example, modeling jobs that are in the “Predicting External Set” step should not be interrupted, because the job will have to restart from the beginning in this case.)

After posting a build to production, always load the front page. This will re-establish database connections and automatically resume any jobs that were running.

## Running a local server:

You may find it efficient to run a Chembench server on your own machine. If you are compiling it locally, the script "build-local.xml" will be used instead. Note that Chembench will only run on Linux. So, you can either use a Linux machine, or run Linux through a virtualization product such as VMWare or VirtualBox on a Windows machine. OSX might work too; we haven't tested it.

Consult the Install Guide document for more information on how to set up a Chembench server.

## Debugging:

Whenever a job goes wrong on Chembench, an email is sent to ceccr@email.unc.edu with details and a stack trace. (Forwarding of the ceccr email can be set up via UNC’s Onyen services page.) The email will give a good hint as to what the problem is. For more details, consult the error logs.

In the package "utilities", you'll find the function Utility.writeToDebug(), which provides error logging for the system. It generates the file "javadebug.log", which is very useful for tracking down bugs. Log files are also created by the Tomcat server (ceccr.log), which is good for debugging JSPs. If there's an error in an external program, that program will also write a log file – it will be in the Logs subdirectory of wherever the program ran from. For example, if datasplit threw an error during a modeling job, you would look in the Logs subdirectory of the modeling job path.

The logfile “javadebug.log” is located at the system’s base directory; typically, this is /public/projects/ceccr/workflow-users/.

The Logs directory for external programs will be generated inside the working directory for the program. For instance, during a Modeling job, program logs are generated in [username]/[jobname]/Logs.

Tomcat’s logfile is typically at /usr/local/tomcat/logs.

|  |  |
| --- | --- |
| **File** | **Type of information** |
| javadebug.log | debug output from every job in Chembench |
| strutsdebug.log | Tracks user movements through the JSPs |
| usage.log | Records major user activities (logins, job creation, new user registrations) |
| ceccr.log | Useful when systemic failures occur in Hibernate or Tomcat (rare) |

# Glossary

Newcomers often get confused at the terms used to describe Chembench components, so here's a dictionary.

Activity- A measure of the response a compound produces.

Dataset - A set of chemical structures (typically in SDF file format.) May or may not include data from experiments/assays performed on those chemicals (typically in ACT file format).

Descriptor - A value calculated for a chemical. For example, a descriptor could be a count of the number of carbon atoms a molecule has. There are a number of different types of descriptors used, including MOLCONNZ, CDK, Dragon, MACCS, and MOE.

Model - A statistical function that models the activity for a specific dataset based on a subset of its descriptors.

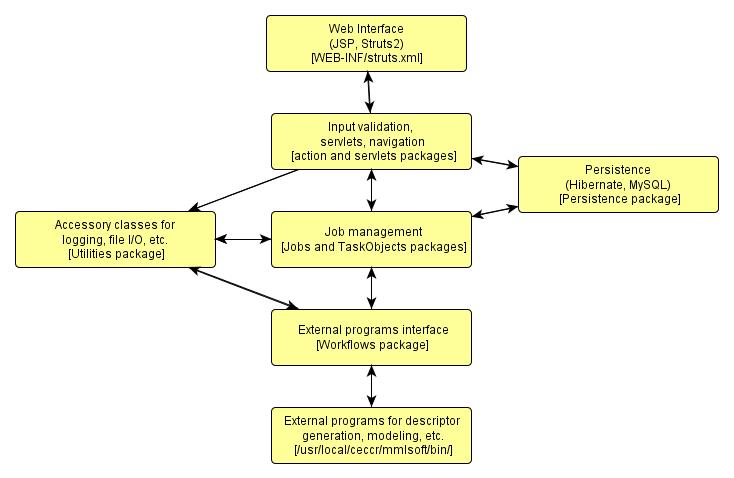
Predictor - A group of models. Sometimes, people will call this a "model". Using many models to make a predictor helps to overcome the weaknesses of each individual model. A predictor is used to predict the likely activity of a set of as yet untested compounds.

Prediction­- The output of applying one or more predictors to a dataset.

Job - A process that's submitted into Chembench's queue. Also called a *Task*, or a *WorkflowTask*.

# System Overview

Chembench has a lot of components. Here's a high-level sketch of how they fit together.



We briefly describe each of these components.

## Web Interface, validation, servlets, navigation

The users interact with JSPs and embedded Java applets on the frontend. The data that shows up on the JSP pages is populated by struts2 tags that read data from Java classes when the page is loaded.

When a user does something in a JSP, a function in an Action class is executed (see the "actions" package in the code). Any information the user entered or selected is passed to the Action.

## Job Management

There are two pieces to the jobs code. The "jobs" package contains the job scheduler and associated logic. At its core is a set of queues; each queue contains a list of WorkflowTask objects to be executed. The "taskObjects" package defines every step in each type of WorkflowTask (for dataset creation, modeling, or prediction).

## Accessory Classes

These contain functions needed by many parts of the system. Things like writing log files, common file I/O operations, running external programs, global constants, etc. are implemented here. You'll find these in the Utility package in the code. Most low-level tasks (writing files, running programs) are implemented here, so you won’t have to rewrite them yourself.

## External Programs Interface

A collection of static functions that are responsible for interfacing with the external programs and reading their output. These are all contained in the "workflows" package. There are sub-packages of “workflows”; these are divided by function. For example, reading and writing of descriptors is handled by “workflows.descriptors”.

## External Programs

Most of the calculations involved in model building, prediction, generating pretty images, and so on are handled by Perl, Python, R, C, C++, and Java programs. Some of these programs are commercial and require licenses, while others were developed by lab members.

## Persistence

Information about the users, datasets, models, predictors, predictions, jobs, and so on is stored here. The database interacts with the Java code through Hibernate, via classes in the Persistence package. Most of the calls to the Persistence classes are done through the PopulateDataObjects class in the Utilities package. See the Database section of this document for more details.

# JSPs

The JSPs are kept in the /jsp/ subdirectory.

They use javascript code from /javascript/. "script.js" contains Javascript that is shared by multiple pages; individual pages have their own ".js" files that correspond to the page's name. (So, for example, modeling.jsp has a corresponding modeling.js).

The JSPs also use CSS styling and images from the /theme/ subdirectory. You'll only need to change things there if you're altering the look and feel of the site.

Most of the JSPs also include a header ("header.jsp"), a footer ("footer.jsp"), and the row of links you see at the top of each page ("centralNavigationBar.jsp").

Many JSPs, such as the Modeling page, are associated with an Action class. The values in the textboxes on that page correspond to values in an Action object; when the page is submitted, the Action class is read.

Lastly, JSPs can get data from session variables via struts2 code like <s:iterate>. There are plenty of examples in the modeling and prediction pages; look at how those work if you want to understand them.

***DEBUGGING:*** If you mess up when you're coding a JSP, when you go to look at that page, you will frequently find that it's either a generic "error" page, or a blank screen, or perhaps only half of the page will load. In order to see why your page isn't working, check Tomcat's server logs. The logfile is currently at */usr/local/tomcat/logs/ceccr.log* . (This may change slightly, depending on the server Chembench is running on.)

The error may also be in your JavaScript code. Make sure to check the JavaScript debugger (Ctrl+Shift+J) on whatever browser you're running.

The JSPs themselves are pretty straightforward.

We use the struts2-style forms:

<s:form name="myform" action="actionName" />

<s:textfield id="text1" name="text1" property="num" />

<s:submit id="button1" name="Joe" />

</s:form>

You can look up the actionName found in the form tag in WEB-INF/struts.xml to tell what Java classes are used in submitting the form.

There is some dynamic loading of JSPs into other JSPs, in order to create the tabbed containers. For example, on the Dataset Upload page, there are three tabs for the choices for how external sets are split. These three tabs are loaded into dataset.jsp; they care called dataset-manualsplit.jsp, dataset-autosplit.jsp, and dataset-nFoldSplit.jsp. <sx:head debug="false" cache="false" compressed="true" /> goes at the top of any page where the struts2 dynamic loading occurs. The <sx:tabbedpanel> tag handles the rest; see the code for examples.

# Actions and struts.xml

Any time the user clicks on a JSP and something happens, there's a class in the Action package that handles it. So, a user might click on the Modeling tab in the navigation bar, which is a link to “modeling”. “modeling” is interpreted by the entry in struts.xml:

<action name="modeling" class="edu.unc.ceccr.action.ModelingFormActions" method="loadPage">

<result>jsp/modeling/modeling.jsp</result>

</action>

which runs the loadPage() function in ModelingFormActions.java. loadPage() checks if the user is logged in; if they are, it directs them to the modeling page. If not, they stay at the Home page.

Generally speaking, the name of the function for populating data to load onto a page in the Actions class will be called loadPage(). The function name for submitting forms is execute().

In addition to loading pages, actions may also change the system state or update the database. For example, the deleteJob() function in DeleteAction.java will remove a job from the queue, and also remove its database entry.

Actions may also create objects. For example, the action QsarPredictionAction creates a WorkflowTask, which is entered into the queue; the queue can later process that task on its own.

Actions may also run external programs by calling functions in Workflows.

# Workflows and Running External Programs

External programs are run using the Workflows package. External programs may be needed by Actions. One example is the SmilesPredictAction, which uses external programs to generate descriptors and predict activity values for a chemical.

The classes in the Workflows package are stateless: none of them have any member variables, nor should they. The external programs write information to output files and log files. These output files can then be read in and returned.

Since they're stateless, the functions can be moved between the different classes freely. So, the functions are organized by what they do: Functions relating to kNN prediction go in workflows.modelingprediction.KnnPrediction, and so on.

***LOGGING PROGRAM OUTPUT:*** In most languages, when you run an external program, that program will run on its own and you don't need to worry about it. In Java, you need to deal with that program's output. If you ignore the output, and the program writes too much output into the buffer, the program will wait forever instead of finishing!

So, any time you run an external program, be sure to call RunExternalProgram.runCommand() or RunExternalProgram.runCommandAndLogOutput(). There are plenty of examples of how this function is used throughout the Workflows code.

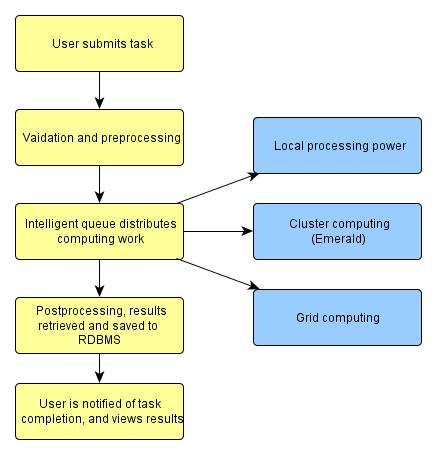
# Queues and WorkflowTasks

There are three queues in Chembench: incoming, LSF, and local. Each queue contains a set of WorkflowTask objects, which represent dataset, prediction, or modeling jobs. The current status of each queue is displayed on the "My Bench" page.

Newly created jobs are put into the "incoming" queue. The incoming queue then immediately sends the job to the LSF or local queue depending on the job type. When a job is completed, it is removed, and some data about its runtime are saved to cbench\_jobstats.

In the LSF and local queues, multiple jobs are run simultaneously. For the local queue, there is a set of threads that will pick up jobs and execute them. When there are no jobs left, the threads will idle until more work is available.

For the LSF queue, there is only one processing thread. The LSF processing thread does two things: it looks for new jobs to submit to the Emerald cluster, and it checks for jobs that have been completed by the Emerald cluster. While a job is running on Emerald, the thread is not doing any processing work. So, several jobs can be running (remotely) by the action of this one thread. There is a limit to the number of jobs Emerald can run at a time, which is around 250. The incoming queue is used as a buffer in cases where the LSF queue cannot hold more jobs.



When a job runs in any queue, its execution follows the pattern above. All jobs have a preprocessing and postprocessing step that must be run locally. The main computation step will be run locally or on LSF depending on which queue the job is in. (Grid computing is yet to be implemented.)

What happens if the Tomcat server running Chembench is restarted partway while jobs are running? It depends on the queue and the type of job. Jobs on the LSF queue that have been submitted to the Emerald cluster will continue running undisturbed, and will still be fetched when they're done. Queued jobs will remain queued. Local jobs will be restarted from the beginning, unless they are prediction jobs; there is a resuming feature built into the prediction job WorkflowTask, so they will be largely unaffected. Error jobs are also restarted when the Tomcat server comes back up, allowing you to fix bugs in existing jobs.

So, it is possible to restart the server to apply updates without significant disruption to user jobs; generally, the only jobs that won't resume are the ones that have a short runtime anyway. The real concern when applying updates is whether there are users currently logged into the site, since they will be disconnected.

# Database, Persistence, and Hibernate

Hibernate is a technology that maps Java objects to database tables. In Chembench, each database table corresponds to one Java object. That is, each member variable in the Java object is a column in the database table for that object.When you make a change to the Java object, you can save those changes to the database with a few lines of code. When you want to get a row from the database, you can get it as a Java object with a few lines of code.

***IMPORTANT:*** Just because you've changed the Java object, that doesn't mean the database changes magically. You have to explicitly save your changes to the database.

The Java objects that correspond to database tables are all in the "persistence" package in the code. The Java member variables and functions use annotations so that the Hibernate libraries recognize them. (Examples: @Id, @Column(name="id"), @Table(name="cbench\_job"), @Transient, and so on.)

There's one class of particular importance in the persistence package: HibernateUtil. It does not correspond to any database table - rather, it holds the information of which Java classes are Hibernate classes. *So if you add a new table and a new Persistence object, you'll need to update HibernateUtil.*

Fetching data from the database is done mostly from the PopulateDataObjects class in the "utilities" package. If you need to pull information from the database, check the functions in that class: the function you need is probably already there. If not, add one.

The functions for saving information back to the database are not organized into any single place.

Since we have a 1-to-1 mapping of database tables to objects, Chembench uses no operations that involve foreign key constraints. If you need to know the Dataset associated with a given Predictor, you get the Predictor object, look up the Predictor's datasetId, and use PopulateDataObjects.getDataSetById() to retrieve the dataset.

The connection string for the database is set in WEB-INF/systemConfig.xml.

The database tables are covered in the Database Design document that’s in the repository under docs/. Look at that, and also at the code in the "persistence" package, to fully understand the database.

# File and Directory Layout

There are no hardcoded paths in the Chembench code. Avoiding hardcoded paths is essential; we need to be able to install Chembench on other servers.

Relative paths are defined in the Constants class in the package “global”. Some of these constants, such as the system’s base path, get their values from WEB-INF/systemConfig.xml.

As an example, the current system has its base path set in systemConfig.xml as “/public/projects/ceccr/”.

Every variable containing a directory name, throughout the system, should end in a “/”. This helps to prevent bugs. When specifying the path of a file, your code should look like:

String fullFilePath = baseDir + fileName;

and *not* like:

String fullFilePath = baseDir + “/” + fileName;

The latter example will produce an error when it tries to read from “/public/projects/ceccr**//**filename”.

Several important file locations and names are listed in global.Constants as well.

## User Directories:

Under the system’s base directory is a directory, typically “workflow-users/”. That contains the files for each user in the system. There is one special directory, “workflow-users/all-users/”, that contains the public datasets and models. The rest look like “workflow-users/userName/”. These directories are created when new users create accounts.

The workflow-users files for production are actually stored at /nas02/depts/chembench/prod/ .

The workflow-users files for development are stored at /nas02/depts/chembench/dev/ .

/nas02/ is an additional storage area we are leasing from ITS. Contact Steve Fishback (<fishback@email.unc.edu>) to renew the lease or in case of issues.

Within each user directory, there are four special subdirectories: “DATASETS/”, “PREDICTORS/”, “PREDICTIONS/”, and “SMILES/”. These store the results of successful operations. For example, when a user submits a new prediction job, the running job’s files are in the directory “jobName/”. If the job reaches an error state, the files will remain there until they are deleted (such as when the user cancels the job). Successfully completed predictions are moved to “PREDICTIONS/jobName/”. The same behavior applies to predictor creation jobs. For SMILES prediction jobs, however, the entire job is performed in the “SMILES/” subdirectory.

Newly uploaded datasets are placed in the directory “DATASETS/datasetName/”.

Under the “DATASETS/datasetName/” directory, there is a “Visualization/Sketches/” subdirectory to store the generated images that represent each molecule. “Visualization/Structures/” stores the same data as the dataset’s SD file, split into many files with one molecule per file. The “Descriptors” subdirectory stores generated descriptors.

# Datasets

A Dataset is an object that contains a set of chemical structures and (optionally) a set of activity values for those compounds. If the Dataset is for creating models, it has activity values.

The set of chemical structures is stored as an SDF (See the Data File Formats section). The activity values are in ACT file format.

When a dataset is uploaded, the files are validated by code in Utility.DatasetFileOperations – formatting and correctness of data are checked. If the dataset passes these checks, a Dataset job is started. If there is an SDF (which is optional if X-file descriptors are uploaded as well), standardization is applied, and descriptors, structure images, and visualizations are generated.

After the upload is completed you’ll see the dataset in the table on the DATASET tab.

# Visualization Flash Application

The Flash code for dataset visualization is compiled independently from the rest of Chembench. The source code is under “visFlash” in the Subversion repository, and it must be compiled inside of Adobe Flash. The compiled swf is stored on the Chembench server under the visFlash directory. The Flash code may someday be remade into a pure ActionScript project so that it can incorporate libraries like prefuse flare, and be compiled without relying on Adobe software. Another option is to recode the pieces into HTML5/Javascript.

# Modeling

On the Modeling page, users choose to model either a category or continuous dataset. The category form of each modeling method uses different parameters and executables from the continuous form. The help pages on the website have good information about what the modeling parameters mean.

The kNN and random forest modeling processes are nondeterministic: given a dataset and a set of parameters, you will get varying results. There are two stochastic processes involved. The compounds are split into training and test sets using random split or sphere exclusion. Descriptors are generated for all compounds, but typically there are too many descriptors for any single model to use. So, random subsets of the descriptors are chosen when each model is built.

For SVM, modeling is deterministic because there is no descriptor selection step: all descriptors are used. However, the results will still be different each time due to randomness in the train/test splitting.

In addition to the main model building process, a y-randomization model building process is run. The Y-randomization shuffles the activity values, and attempts to create models from the resulting noise. Ideally, no models will be built from the noise data. If no y-randomized models pass the modeling criteria, this validates the modeling process.

When the user submits a modeling job, the information is captured by the modeling Action class. The Action class passes the information to QsarModelingAction, which does the following:

1. Creates a QsarModelingTask, which is a type of WorkflowTask.
2. The QsarModelingTask is wrapped in a Job object and entered in the queue.
3. The queue calls setUp() on the QsarModelingTask, creating the directory {username} /{predictorName}/
4. The job waits in the queue. Once it's at the top, the queue calls executeLocal() or executeLSF() on QsarModelingTask.
5. Descriptors are converted to X file format and scaled.
6. The descriptor data is split into training and test sets.
7. A y-randomized version of the activity data is generated.
8. Model building is performed on the real dataset, and then on the y-randomized dataset.
9. Predicted values for the external set's compounds are calculated.
10. The WorkflowTask's state is set to "finished", and the queue calls save() on the QsarModelingTask. This saves the predictor into the database. The user can now view the predictor through the Jobs page, and keep or discard the predictor.

# Predictions

On the Predictions page, users pick a predictor and then use it to predict the activity for some set of chemicals.

Predictions are deterministic: given a chemical, a predictor, and an applicability domain cutoff, you will always get the same result. The cutoff determines the point at which each model in a predictor will stop attempting to predict the activity of a model. A model will choose not to predict compounds if they are "too far outside" the domain of the model. The cutoff determines the "too far" distance. Setting the cutoff higher increases the number of models that will try to predict an activity for the molecule.

A predictor is made up of many models, and each model can predict a value for a given compound. Therefore, the output of the predictor is a *consensus* prediction: an average of the outputs of the individual models.

## SMILES Prediction

A user enters a SMILES string, either by pasting it in or drawing it using the MarvinSketch applet. They choose a cutoff value and submit. Single-molecule prediction is relatively quick, so this does not create a job. Instead, the SmilesPredictAction:

1. Deletes the contents of the directory {username} /SMILES/
2. Copies the selected predictor to {username} /SMILES/ (the slowest step of the process)
3. Writes the SMILES string to a file
4. Uses molconvert to convert the SMILES file to an SDF
5. Generates descriptors for the SDF
6. Normalizes the descriptors to fit the range of the predictor, creating a ".x" file
7. Runs the Prediction executable (kNN, random forest or SVM) on the normalized descriptors
8. Reads in the result file (cons\_pred)
9. Sets a variable with the predicted activity
10. Forwards the user to a JSP with the prediction on it.

All of the external programs in this procedure are executed through functions in the "workflows" package.

## Dataset Prediction

A user picks a predictor and a dataset. Activities are predicted for each compound in the dataset. This can take a long time for large inputs, so a WorkflowTask must be created for it. Predicting a dataset runs the QsarPredictionAction, which starts a process that does the following:

1. Creates a QsarPredictionTask, which is a WorkflowTask.
2. The QsarPredictionTask is wrapped in a WorkflowTask and entered in the queue.
3. The queue calls setUp() on the QsarPredictionTask, creating the directory {username} /{predictionName}/
4. The job waits in the queue. Once it's at the top, the queue calls execute() on QsarPredictionTask.
5. As with SMILES prediction, the predictor is copied over, descriptors are generated and normalized according to the predictor's range (train\_0.x), the prediction executable is run, and the resulting consensus prediction (cons\_pred) is read in.
6. The WorkflowTask's state is set to "finished", and the queue calls save() on the QsarPredictionTask. This saves the prediction result into the database. The user can now view the prediction results through the Jobs display, and keep or discard the prediction.

# Users and Administration

The Admin page (/admin) allows you to manage users:

* You can view the list of users, and delete the ones you don't like.
* You can make other users into administrators.
* You can grant the ability for a user to download descriptors. We can’t let the general public download our descriptors, because that would violate our license agreements. However, users who are members of the lab are allowed to.
* You can send an email to all of the users.

Being defined as an administrator also gives you power over the queue from the Jobs page: you can see other users' tasks there, and cancel them.

In addition to the admin page, some administration tasks are handled in WEB-INF/alternate-configs/dev/systemConfig.xml and WEB-INF/alternate-configs/prod/systemConfig.xml. It holds the keys for the ReCaptcha system, which prevents bots from creating hundreds of user accounts. It also defines important values for the systems such as the name of the webserver and the database connection string.

# Data File Formats

These are also described in the help section.

## .SDF files

(Also referred to as "SD files", since SDF stands for "structure data file".)

SDFs are used to store sets of chemical structures and can be 2D or 3D. They are typically the input files we use. Here's a sample:

4254097

MOE2005 2D

44 47 0 0 1 0 0 0 0 0999 V2000

1.3550 -4.8300 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0

1.0920 -3.9960 0.0000 O 0 0 0 0 0 0 0 0 0 0 0 0

0.4780 -2.3340 0.0000 C 0 0 3 0 0 0 0 0 0 0 0 0

…

1.0240 3.0240 0.0000 O 0 0 0 0 0 0 0 0 0 0 0 0

0.5970 4.3590 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0

2.1340 2.1230 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0

1 2 1 0 0 0 0

2 3 1 0 0 0 0

3 4 1 0 0 0 0

…

41 42 1 0 0 0 0

41 44 1 0 0 0 0

42 43 1 0 0 0 0

M END

$$$$

4239291

MOE2005 2D

19 21 0 0 0 0 0 0 0 0999 V2000

0.9640 -5.2910 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0

1.3970 -3.9610 0.0000 N 0 0 0 0 0 0 0 0 0 0 0 0

2.7660 -3.6690 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0

…

-1.4850 5.1300 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0

-2.3080 3.9980 0.0000 O 0 0 0 0 0 0 0 0 0 0 0 0

0.4600 -2.9200 0.0000 C 0 0 0 0 0 0 0 0 0 0 0 0

1 2 1 0 0 0 0

2 3 1 0 0 0 0

2 19 1 0 0 0 0

…

15 16 1 0 0 0 0

16 17 2 0 0 0 0

17 18 1 0 0 0 0

M END

$$$$

The first line is the ID of the first compound. The atom coordinates and bond information come after that. There can be many optional fields. These come in an XML-like format (e.g. <MolWeight>128</MolWeight>) A compound description ends with $$$$. Officially, an SDF is not allowed to contain lines of over 200 characters. In practice, many do. This can cause programs like MolconnZ to fail ungracefully, so Chembench chops off SDF lines past 200 characters. <http://www.epa.gov/NCCT/dsstox/MoreonSDF.html> has more details on the SDF format.

Chemical structures can also be represented as mol files or SMILES strings. All file formats contain roughly the same information, and there are many programs in existence that translate between them, such as JChem's "molconvert". The descriptor generation executables take SDF inputs. The 3D rotatable molecules you see on the site are from mol files. SMILES strings entered on the Predictions page are converted to SDFs before use.

## .x files

.x is a file format used by the kNN executables. It is similar to the matrix format accepted by other data mining programs such as LibSVM.

The .x file contains a matrix of compounds and descriptor values, as below:

|  |  |
| --- | --- |
| [LINE 1]: 7 315 | The line indicates that a 7 by 315 matrix follows: There are 7 compounds, each with 315 descriptor values. |
| [LINE 2]: narecs nvx nedges nrings ncircuits... | The second line contains the names of the descriptors. |
| [LINE 3]: 1 4254097 0.5 0.609756 0.5625 ... | From the third line on, each line represents one compound. The first value on each line is an index. The second value is an ID for the compound. The remaining numbers are the values of the descriptors for the compound. |
| [LINE 4]: 2 4239291 0 0 0.0208333 0.142857 ... |

At the end of the file, there may be two additional lines. If a .x file has been normalized, the original descriptor values need to be preserved; these lines tell what the range of each descriptor was before normalization.

|  |  |
| --- | --- |
| [SECOND TO LAST LINE]: 2 19 2 ...  [LAST LINE]: 4 60 68 ... | The “2” and “4” that begin these two lines indicates that the first descriptor, "narecs", originally had a minimum value of 2 and a maximum value of 4 over all compounds in the set. The next two values, 19 and 60, indicate the minimum and maximum values for the second descriptor, “nvx.” iIt continues this for all descriptors. |

## .mz files

MolconnZ takes in a .sdf file and outputs a .mz file. The .mz file contains molconnZ descriptors for the compounds from the SDF. The .mz file looks like:

*[descriptor names]:*

moleculenumber narecs nvx nedges nrings ncircuits nclass nelem ntpaths molweight molname formula

nX0 nX1 nX2 nXp3 nXp4 nXp5 nXp6 nXp7 nXp8 nXp9 nXp10

X0 X1 X2 Xp3 Xp4 Xp5 Xp6 Xp7 Xp8 Xp9 Xp10

… *[descriptor values for molecule 1]:*

1 3 44 47 4 11 43 5 7518 635.153 4254097 H(43)C(32)N(2)O(9)Cl(1)

44 47 71 92 115 143 165 188 201 224 249

32.559757 20.627108 19.970501 16.373892 12.664577 9.750673 7.042437 5.126579 3.225958 2.186764 1.456898

…*[descriptor values for molecule 2]:*

2 2 19 21 3 3 17 4 615 262.351 4239291 H(22)C(15)N(2)O(2)

19 21 30 39 49 56 60 62 64 64 62

13.294681 9.185071 8.442608 7.163390 5.869330 4.134155 2.965950 1.912135 1.281447 0.819887 0.555201

The .mz file is ugly, but straightforward to interpret. First, all the descriptors are listed, and then their values are provided for each compound. There are 11 elements on each line of the file. Occasionally, molconnZ will spit out something insane with crazy characters instead of numbers. This will be caught by the current Java code and dealt with - if you write your own code to deal with .mz files from molconnZ, you might need to do that too.

## .act files

The .act files store activities (numerical results associated with each chemical; these are what we're building a model on or trying to predict). They must have a corresponding .sdf file with them to describe the chemical. The system validates that there are the same number of compounds in the .sdf file and its associated .act file.