|  |
| --- |
| The University of North Carolina at Chapel Hill ESHELMAN SCHOOL OF PHARMACY |
| C-Chembench |
| Developer Guide V1.0 |
|  |
| **Molecular Modeling Laboratory** |
|  |

|  |
| --- |
|  |

Contents

[Introduction 4](#_Toc224986616)

[Getting started with the C-Chembench code 5](#_Toc224986617)

[Tools: 5](#_Toc224986618)

[References: 5](#_Toc224986619)

[Building the code: 5](#_Toc224986620)

[Running a local server: 5](#_Toc224986621)

[Debugging: 5](#_Toc224986622)

[Glossary 7](#_Toc224986623)

[System Overview 8](#_Toc224986624)

[Frontend 8](#_Toc224986625)

[Stateful Java Classes 8](#_Toc224986626)

[Accessory Classes 8](#_Toc224986627)

[Database 8](#_Toc224986628)

[External Programs 9](#_Toc224986629)

[Workflows 9](#_Toc224986630)

[JSPs 10](#_Toc224986631)

[Actions and struts-config.xml 12](#_Toc224986632)

[Workflows and Running External Programs 13](#_Toc224986633)

[Queue, QueueTask, and WorkflowTask 14](#_Toc224986634)

[Database, Persistence, and Hibernate 15](#_Toc224986635)

[File and Directory Layout 16](#_Toc224986636)

[User Directories: 16](#_Toc224986637)

[Datasets 17](#_Toc224986638)

[Visualization Applet 17](#_Toc224986639)

[Predictions 19](#_Toc224986640)

[SMILES Prediction 19](#_Toc224986641)

[Dataset Prediction 20](#_Toc224986642)

[Modeling 21](#_Toc224986643)

[Users and Administration 22](#_Toc224986644)

[Data File Formats 23](#_Toc224986645)

[.SDF files 23](#_Toc224986646)

[.x files 24](#_Toc224986647)

[.S files 25](#_Toc224986648)

[.act files 25](#_Toc224986649)

# Introduction

What's the purpose of C-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 kNN models
* A way to share our kNN modeling procedure with the world
* A way for us to beat RPI's "RECCR" system
* 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 C-Chembench system.

# Getting started with the C-Chembench code

## Tools:

Get the latest version of Eclipse. 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 "http://source.ibiblio.org/svn/ceccr". Open CECCR-QSAR, right click on "trunk", and click Check Out. You now have the C-Chembench code on your computer. Change the perspective to Java and you should be able to see the code.

In order to commit changes, you will need a username and password for the repository.

## References:

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

This document assumes that the reader is familiar with Tomcat, Java development, web development (HTML/JavaScript/JSPs), version control, Eclipse, databases, and has a basic familiarity with Linux.

## Building the code:

On the server, there is a script "build.sh". When you run it, it will

* stop the Tomcat server,
* check out the latest version of C-Chembench from the Subversion repository,
* compile it using "build.xml",
* move the compiled WAR file to Tomcat's WEB-INF/apps/ directory, and
* restart Tomcat.

## Running a local server:

You can also run the server on your own machine. If you are compiling it locally, the script "build-local.xml" will be used instead. Note that C-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 your own C-Chembench server.

## Debugging:

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 - Utility.writeProgramLogFile() generates that.

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.

# Glossary

Newcomers often get confused at the terms used to describe C-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, Dragon, MACCS, and MOE.

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

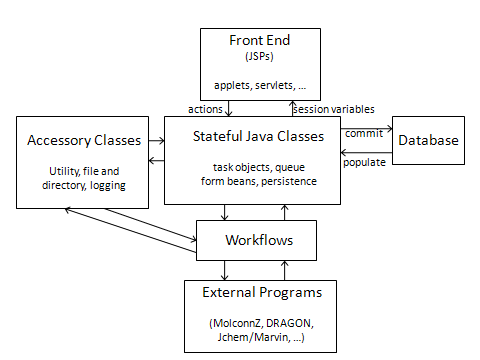
Predictor - A group of models. Sometimes, people will call this a "model". Using many models to make a predictor helps to overcome the weaknesses of each individual model. A predictor is used to predict the likely activity of a set of as yet untested compounds.

Prediction­- The output of applying a predictor to a dataset.

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

# System Overview

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



We briefly describe each of these components.

## Frontend

The users interact with JSPs and embedded Java applets on the frontend; some of these get information via servlets. The data that shows up on the JSP pages is typically fetched via session variables. (You can also embed Java code into the JSP to get data.)

When a user does something in a JSP, an Action is executed. Any information the user entered or selected is passed to the Action. In many cases, a FormBean is used to capture that data.

## Stateful Java Classes

These are the Java classes that hold data that depend on the user's actions. Most of C-Chembench's logic code is in here. As you might imagine, there's a lot of pieces to it; it will be covered in depth later in this document.

## Accessory Classes

These contain functions needed by many parts of the system. Things like writing log files, common file I/O operations, global constants, etc. are implemented here. You'll find these in the Utility package in the code.

## Database

Information about the users, datasets, models, predictors, predictions, tasks, 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 for more details.

## External Programs

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

## Workflows

A collection of static, stateless functions that run the external programs.

# 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 import a header ("header.jsp"), a footer ("footer.jsp"), and the row of links you see at the top of each page ("centralNavigationBar.jsp").

Since it is possible to embed Java code into a page as well, many pages import Java classes.

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

Lastly, JSPs can get data from session variables via JSP code like <logic: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 C-Chembench is running on.)

The error may also be in your JavaScript code. Make sure to check the JavaScript debugger on whatever browser you're running.

The JSPs themselves are pretty straightforward. There are two types of forms in them.

You can use either the html-style forms:

<form name="myform" action="action()">

<input type="text" id="text1" value="3">

<input type="submit" id="button1" name="Joe">

</form>

or the JSP-style forms:

<html:form name="myform" action="action()" />

<html:text styleId="text1" value="3" name="text1" property="num">

<html:submit styleId="button1" name="Joe">

</form>

Use the JSP-style forms if you want to link the form to a FormBean.

Note that "styleId" is used in the JSP-style forms instead of "id". The javascript call document.getElementById("text1").value will return "3" for both of the example forms above.

In the past, we've tried having JSPs also load in JSPs, via AJAX requests. This does the following good things:

* Breaks up the code for an individual JSP into smaller pieces
* Allows for some fancy dynamic loading

But it also does some very bad things:

* The JSPs are complicated already. Spreading the code across many files makes it even more complicated.
* It's way harder to debug when an error in one sub-JSP can cause errors in another sub-JSP.
* It makes it very difficult for future developers to understand what you did and for them to expand on it.

So, if you're in a situation where you're thinking about dynamically loading JSPs into other JSPs, think it over. There may be a better way to do it, that is easier to code and maintain. Also, packing a lot of dynamically-loaded functions onto one page will confuse your users.

A simple design is a good design.

As Dijkstra said: "The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague."

# Actions and struts-config.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 runs "loadModelingPage.do". That gets interpreted by struts-config.xml, which starts the ModelingForward action. ModelingForward.java 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.

Note the "Forward" in that action - page forward actions like that one always have Forward in the name. They will do things like:

* Check a login
* Populate session variables that are needed by the page they forward to (e. g., the Modeling page needs to know what datasets a current user has access to, so a session variable is set with that information.)
* Return a "forward" telling struts-config where to send the user.

Actions may also change the system state or update the database. For example, the CancelJob action will remove a task 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 call functions in Workflows, to access external programs.

# Workflows and Running External Programs

If an external program is run, it should be done through a function call to 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 functions' return values are usually *void*, since the external programs write information to output files. These output files are then read by stateful objects.

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 prediction go in KnnPredictionWorkflow, and so on.

***LOGGING PROGRAM OUTPUT:*** This is important. 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 Utility.writeProgramLogFile() for it, unless you're certain that the program doesn't generate output. There are plenty of examples of how this function is used throughout the Workflows code.

The Workflows functions can also be used for calculations that take the place of external programs. An example is in the MolconnZToDescriptors.java file.

# Queue, QueueTask, and WorkflowTask

The job queue shown on the Jobs page is defined in the Queue class of the persistence package. The queue contains QueueTask objects, which are defined inside of the Queue class. These QueueTask objects are kept in the database via Hibernate (see next section). Each QueueTask is stored inside the database table cbench\_task.

So, if something goes wrong and a QueueTask gets stuck somehow, you can always go into the database and delete it there.

There are many different types of tasks that the queue may be called upon to handle (modeling, prediction...), so an abstraction is needed. This abstraction is the interface WorkflowTask, which is defined in the package "task". (It is NOT related to the Workflows package.)

When a QueueTask comes to the top of the queue, and needs to be executed, the queue calls the QueueTask's WorkflowTask's execute() function, and the WorkflowTask starts running, no matter what type it is.

To summarize:

* The Queue contains a set of QueueTasks, which it executes in order.
* A QueueTask is a Hibernated object that contains a WorkflowTask.
* A WorkflowTask is an abstraction that lets the queue process many different task types.

What happens if the Tomcat server running C-Chembench is restarted partway through executing a job? Well, the WorkflowTask that was running is gone -- that was part of the user's session state, and the session has vanished. But the QueueTask remains: It was stored in the database, and will still exist when the server comes back up. When the server returns, the queue will attempt to run the job, find that it has no WorkflowTask associated with it, and the job will be set to an error state.

# Database, Persistence, and Hibernate

Hibernate is a technology that maps Java objects to database tables. In C-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\_task"), @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, C-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 only place foreign keys come in is for cascading behavior. If a user is deleted, the predictors and predictions and datasets associated with that user must be wiped out as well. This is handled in the database by foreign key constraints.

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 C-Chembench code. Avoiding hardcoded paths is essential; we need to be able to install C-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”.

## 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.

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.

# 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 first uploaded, a QueueTask is started. That QueueTask splits the multi-compound SDF into several single-compound SDFs, and uses molconvert to create 2D images for each compound. This process takes roughly 1 second per compound.

A Dataset object contains: name, description, creation date, sd file name, act file name, knnType.

From DATA MGMT tab you can upload datasets. When you upload a dataset, you will choose what type it should be (CONTINUOUS, CATEGORY, PREDICTION).

CONTINUOUS and CATEGORY datasets will need both **sd** and **act** files, but PREDICTION will ask you only for an **sd** file (the **act** file will be created using compound ids from the **sd** file. The **act** file is needed for the visualization applet). Note that this generated **act** file contains only placeholder values (not actual data), so it is useless for model building.

After the upload is completed you’ll see your dataset in the table on the DATA MGMT tab. To see the dataset info you need to click on the dataset you like. The Visualization applet will start.

From the DATA MGMT tab you’ll be able to delete the dataset (actions.DeleteUserFileAction class is used). Also you can generate additional visualized info for your dataset by clicking on **create** link inside the table. You’ll be redirected to the Generate dataset info page where you’ll choose what you like to see inside the Visualization applet.

# Visualization Applet

The applet code is placed under Applet dir. The best practice in applet development is to copy the code from the project and create new project specific for the applet. This way you’ll build the applet independently from the CECCR project.

Here is the list of the applet classes:

**Main**

**AbstractPanel** – class which handles the mouse events for each panel as AbstractPanel is parent class for each panel in applet. Also it is in charge for hide/show panel content and it controls all other panel sizes.

**CECCRApplet** – main applet class. Here we are reading the input data and creating the tabs for each representation.

**Heat Map**

**ColorAwareTableCellRenderer** - allow us to draw each cell with the different color depending on the value this cell has.

**ColorMap** – Creates the color from the value in cell.

**ColumnHeaderRenderer** –rendersthe header of the heat map table.

**HeatMap** – The third tab which contains heat map for current dataset.

**HeatMapTable** - interactive table for heat map, build heat map automatically from table model.

**HeatMapUI** - user interface for drag’n’drop functionality for rows.

**LabelRenderer** - renderer for drawing row header labels.

**VerticalLabelUI** – UI for render the vertical text inside heat map table.

**Tree**

**TreePanel** – prefuse package class changed to show images when user mouse over the compound node in the tree.

**TreeStructure** – create the xml data for tree from heat map. Clustering is implemented here.

**Trees** – second tab – for showing the tree.

**Other**

**Compound** – Object for storing compound info.

**CompoundTable** – The first tab panel which contains dataset name, description,creation date and the table with compound ids, structure images and act values.

**ImagePanel** – panel for showing images.

**Loader** – shows progress bar with current state on applet loading.

**PCAPlot** – panel for static PCA plots. Will be extended to handle interactive PCA plots.

**Utility** – contains useful functions .

# Predictions

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

kNN prediction is deterministic: given a chemical, a predictor, and a cutoff, you will always get the same result. 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.

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.

## SMILES Prediction

A user enters a SMILES string, either by pasting it in or drawing it using the applet. They choose a cutoff value and submit. Single-molecule prediction is relatively quick, so this does not create a QueueTask. 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 kNN Prediction executable on the normalized descriptors
8. Reads in the result file (cons\_pred)
9. Sets a session 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 a large dataset, so a QueueTask 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 QueueTask 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 kNN prediction executable is run, and the resulting consensus prediction (cons\_pred) is read in.
6. The QueueTask'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 page, and keep or discard the prediction.

# Modeling

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

The kNN modeling process is 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 three sets (training, test, and external validation) using 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.

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 formBean. The formBean passes the information to QsarModelingAction, which does the following:

1. Creates a QsarModelingTask, which is a WorkflowTask.
2. The QsarModelingTask is wrapped in a QueueTask 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 execute() on QsarModelingTask.
5. Descriptors are calculated for the compounds.
6. The descriptor data is split into training, test, and external sets.
7. A y-randomized version of the activity data is generated.
8. kNN model building is performed on the real dataset, and then on the y-randomized dataset.
9. Values for the external set's compounds are calculated.
10. The QueueTask'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.

# Users and Administration

The System Settings page (/jsp/administration/settings/systemSettings.jsp) allows you to manage:

* Users: You can view the list of users, and delete the ones you don't like.
* User Acceptance: This affects how new user accounts are created. It can be set to "Manual" or "Automatic". Normally, it's set to Automatic: new users fill out the form on the front page, and immediately get an email with a generated password in it, so they can log in. When this setting is changed to Manual, the email is sent to everyone on the administrators list (defined in WEB-INF/systemConfig.xml). An administrator then has to log in and approve the user -- only then will the new user get their password. The User Acceptance setting is tored in the database, in cbench\_adminsettings (persistence.AdminSettings)..
* Cutoff values: If a user is not an administrator, we impose limits on the size of the job they can run. This prevents someone from submitting a gigantic task that takes up all the system resources for days. These limits are also stored in the database, in cbench\_adminsettings (persistence.AdminSettings).
* Tasks: If a user submits a task that is bigger than the cutoff values, C-Chembench will email all the administrators. An admin then logs in and approves or denies the user's task. Until the task is approved or denied, it will stay in the queue but not run.
* Software expiration reminders: Software such as MolconnZ depends on license files in order to function properly. If you forget to renew the license, modeling and prediction stop working. So, there's a feature on this page that lists the expiration dates for software licenses. You have to change it manually. The data is stored in the table cbench\_software (persistence.SoftwareExpiration).

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

WEB-INF/systemConfig.xml also holds the keys for the ReCaptcha system, which prevents bots from creating hundreds of user accounts.

There has been some interest in allowing no-login access to C-Chembench, so that new users would not be put off by having to create an account and remember the password. To implement this, we would need a way to create a temporary, anonymous user account. The user could be assigned a username, something like guest\_283495, and allowed some minimal access to the system.

# Data File Formats

The files associated with kNN are described in the kNN documentation. There is a copy on the CECCR website: <http://ceccr.ibiblio.org/c-C-Chembench/theme/kNNQSAR.pdf>. It is also in the code repository, under the “theme” directory. In addition, there are a number of data files that are used and created in the process.

## .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 C-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. |

## .S files

MolconnZ takes in a .sdf file and outputs a .S file. The .S file contains molconnZ descriptors for the compounds from the SDF. The .S 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 .S 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 .S 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.