Tutorial 2: Computing chemical properties and generating feature datasets

Author: Benjamin P. Brown (benjamin.p.brown17@gmail.com)

Date: 01-2022

In Tutorial 1, we discussed filtering molecules by comparing property values, sorting molecules by property values, and scoring molecular alignments from computed property values. So, what are these property values that we keep mentioning?

Chemical features are those that arise from the chemical identity of a substance. As the true nature of a chemical system is most adequately described with quantum mechanical methods, we tend to use knowledge-based, statistical, or simpler physical models to approximate many chemical features. We use the term "properties" to refer to individual chemical features. We use the term "descriptors" to refer to combinations of properties often used to train quantitative structure activity / property relationship (QSAR/QSPR) models; however, the terms are frequently used interchangeably in the BCL.

The BCL was designed with a modular descriptor interface and extensible property definitions framework. This allows both developers and users alike to write new descriptors for specific applications as needed. The primary applications for working with properties and descriptor datasets in the BCL are `molecule:Properties` and `descriptor:GenerateDataset`, respectively.

To see a list of available predefined molecular properties, perform the following command:

```
bcl.exe molecule:Properties  --help
```

The property interface is organized into two general categories: (1) descriptors of molecules, and (2) descriptors of atoms.

Once again, set the path to your BCL directory as an environment variable. For example, the main BCL directory for these tutorials in the Meiler Lab is in `/sb/apps/bcl/bcl`. In bash, we can set this as an environment variable by doing the following:

```
export BCL=/sb/apps/bcl/bcl
```

In tcsh, this is accomplished with a similar command:

```
setenv BCL "/sb/apps/bcl/bcl"
```

We can also add the BCL executable to our PATH environment variable.

```
export PATH=/sb/apps/bcl/bcl/build/linux64_release/bin:$PATH
```

Note that the "bcl.exe" is interchangeable with "bcl-apps-static.exe" in `${BCL}/build/linux64_release/bin`

Let's get started!

Part 1: Computing properties of small molecules

Let's start by computing a few whole molecule descriptors of the type I tyrosine kinase inhibitor (TKI) dasatinib.

Subsection 2A: Molecular properties

As the names suggest, some descriptors are intrinsic to the whole molecule, while others are specific to atoms. For example, compute some whole molecule descriptors for the EGFR kinase inhibitor osimertinib.

Unless otherwise stated, all commands in this tutorial are run from the `BCL_Workshop_2022/Tutorial_2/inputs/` directory.

```
bcl.exe molecule:Properties \
-add_h -neutralize \
-input_filenames dasatinib.sdf.gz \
-tabulate Weight NRotBond NRings TopologicalPolarSurfaceArea \
-output_table ../output/dasatinib.mol_properties.table.txt
```

The `tabulate` flag will output the properties for each molecule in row-column format in the file specified by `output_table`. There is also a `statistics` flag that will compute basic statistics for each of the specific descriptors across all the molecules in the input SDFs and output to `output_histogram`. In addition, you could also use the `add` flag to add the computed properties as MDL properties on an output SDF.

Compute a few more properties. Count the number of total atoms, the number of heavy atoms, total number of hydrogen bond acceptors, total number of hydrogen bond donors, and the number of rotatable bonds that are in dasatinib. Also compute the polarizability, estimated logP (using any of the metrics you may find), and complexity (synthetic accessibility score).

Subsection 2B: Atomic properties

We can also compute properties of individual atoms on our molecule. For example, one very common atomic property is partial charge. There are several different approximations for atomic partial charge in the BCL. Here, we will compute Atom_TotalCharge, which is the sum of partial charges in sigma and pi orbitals.

```
bcl.exe molecule:Properties \
-input_filenames dasatinib.sdf.gz \
-add_h \
-tabulate Atom_TotalCharge \
-output_table ../output/dasatinib.atom_totalcharge.table.txt
```

Notice that the output table now has multiple columns corresponding to a single property, Atom_TotalCharge. If you count them, you will get 59 – one for each atom in the molecule. Thus, atom-based descriptors return a value for every atom, meaning that the output across an entire dataset of molecules will most likely not be fixed-width.

In addition to the pre-defined properties that the BCL comes packaged with, you can create custom chemical properties directly from the command-line using molecule- or atom-specific operations. For example, some whole molecule properties can be obtained by performing simple operations on the atomic properties. Topo-logicalPolarSurfaceArea (whole molecule property) is the sum of Atom_TopologicalPolarSurfaceArea (atomic property) across the whole molecule.

Compute the sum of the atomic total partial charges on dasatinib and a version of dasatinib with a protonated piperazine ring.

```
bcl.exe molecule:Properties \
-input_filenames dasatinib.sdf.gz dasatinib.p1.sdf.gz \
-add_h \
-tabulate 'MoleculeSum(Atom_TotalCharge)' \
-output_table ../output/dasatinib.sum_atom_totalcharge.table.txt
```

In the neutralized dasatinib molecule we can see that the sum of partial charges is effectively zero. In the formally charged dasatinib molecule we can see that the sum of partial charges is effectively one. Examples of additional operations include other basic statistics (mean, max, min, standard deviation, etc.), property radial distribution function (RDF), Coulomb force, and shape moment. See the help menu for additional options and details.

Subsection 2C: Getting started with property operations

In Tutorial 1, we reviewed how you can use the molecule:Filter application to remove molecules from a dataset that failed specific druglikeness criteria (e.g., TPSA ≥ 140 Å^2). Several familiar druglikeness metrics come prepackaged in the BCL (i.e., Lipinski's Rule of 5 and Veber's Rule), as well as several others inspired by the literature and conventional medicinal chemistry practices. For each molecule in the Platinum Diverse dataset[1], count how many Lipinski and Veber violations there are. In addition, count as drug-like all molecules that have fewer than two Lipinski violations:

```
bcl.exe molecule:Properties \
-add_h -neutralize \
-input_filenames platinum_diverse_dataset_2017_01.sdf.gz \
-output_table platinum_diverse_dataset_2017_01.druglike.txt \
-tabulate LipinskiViolations LipinskiViolationsVeber LipinskiDruglike
```

The property LipinskiViolations counts how many times a molecule violates one of Lipinski's Rules (≤ 5 hydrogen bond donors (HBD; –NH and –OH groups), ≤ 10 hydrogen bond acceptors (HBA; any –N or –O), molecular weight (MW) < 500 Daltons, and water-octanol partition coefficient (logP) < 5). The LipinskiViolationsVeber property computes the number of times a molecule violates Veber's Rule (infraction if TPSA ≥ 140 Å^2 and/or number of rotatable bonds > 10). The LipinskiDruglike property is a Boolean that returns 1 if fewer than two Lipinski violations occur; 0 otherwise. There is no equivalent Boolean operation for Veber druglikeness; however, it is simple to implement one using the aforementioned operations.

```
bcl.exe molecule:Properties \
-add_h -neutralize \
-input_filenames platinum_diverse_dataset_2017_01.sdf.gz \
-output_table platinum_diverse_dataset_2017_01.veber_druglike.txt \
-tabulate 'Define(VeberDruglike=Less(lhs=LipinskiViolationsVeber, rhs=1))'
VeberDruglike
```

This command makes use of the Define and Less operations to return 1 if there are no violations to Veber's Rule and 0 otherwise. New properties created with Define can also be passed to subsequent operations on the same line. For example, one could create a descriptor called VeberAndLipinskiDruglike by doing the following:

```
bcl.exe molecule:Properties \
-add_h -neutralize \
-input_filenames platinum_diverse_dataset_2017_01.sdf.gz \
-output_table platinum_diverse_dataset_2017_01.veber_druglike.txt \
-tabulate \
'Define(VeberDruglike=Less(lhs=LipinskiViolationsVeber, rhs=1))' \
'Define(VeberAndLipinskiDruglike=Multiply(LipinskiDruglike,
VeberDruglike))' \
VeberAndLipinskiDruglike
```

This new descriptor returns 1 if a molecule passes both druglikeness filters, and 0 otherwise.

Many metrics can be created using the BCL descriptor framework without modifying the source code. Not infrequently, we generate machine learning (ML) models (Tutorial 3) to predict a property and we define a new descriptor that acts as a wrapper for the ML model. For example, the XLogP property is a neural network-based prediction of the water-octanol partition coefficient. In Brown et al. 2022[2], we demonstrate how the BCL descriptor framework can be used to train a decision tree that reproduces the canonical QED (quantitative estimate of druglikeness) score originally introduced by Bickerton et al. 2012[3].

I computed halogen count statistics across a version of CHEMBL that I downloaded in 2019(?). Based on these statistics, make a descriptor called HasDruglikeHalogenation (or something else if you like). This descriptor will return true if there are fewer than 6 halogens total, fewer than 5 fluorines, fewer than 3 chlorines, fewer than 2 bromines, and no iodines. These counts represent the mean plus three standard deviation counts (rounded to the nearest integer) across the entire dataset.

The command-line I used to collect the statistics (the dataset is not provided to reduce storage space requirements) is the following:

```
bcl.exe molecule:Properties \
-statistics "MoleculeSum(IsF)" "MoleculeSum(IsCl)" \
"MoleculeSum(IsBr)" "MoleculeSum(IsI)" "MoleculeSum(IsHalogen)" \
-input_filenames <dataset> \
-output_histogram chembl_halogens_statistics.dat \
-scheduler PThread 8
```

On the topic of druglikeness, it is worth noting that additional advanced methods are also available to classify the chemical space of molecules in a dataset. In some cases, it is useful to identify potential drug-like compounds that not only fit the criteria discussed above but are also similar to some known class(es) of drugs. For example, when performing fragment-based combinatorial library design for kinase inhibitors, in addition to filtering out molecules that violate Veber's rules, it may also be desirable to filter molecules that are not sufficiently chemically similar to existing kinase inhibitors. This can be accomplished by building and scoring against an applicability domain (AD) model. For further details on creating and using AD models in the BCL, see section 7.4.3 of Brown et al. 2022[2].

Subsection 2D: Autocorrelations

A very useful operation for QSAR/QSPR modeling is the autocorrelation function. Autocorrelations are regularly used as features in cheminformatics machine learning models [4]. Therefore, in our discussion of autocorrelations, we will primarily think about them as feature representations of molecules that we use to build models with which to perform virtual screening. They are useful for several reasons:

(1) The output from an autocorrelation function is fixed-width independent of the size of the molecule from which it is generated

(2) They are delocalized descriptions of the feature space of a molecule, which means that they do not directly correspond to substructures (i.e., there is not a one-to-one mapping between autocorrelations and the molecules from which they are generated). **This is very useful for scaffold hopping**.

(3) They can be computed from any user-defined atomic property.

So how do they work? Autocorrelations sum pairwise property products into distance bins by calculating the separation between molecule atom pairs in number of bonds (2DA) or Euclidean distance (3DA). Each distance bin is further separated into three sign-pair bins corresponding to property value sign of each atom in the pair (eq. 1) [5].

$$A(r_a, r_b) = \sqrt{\sum_j^N \sum_i^N \delta(r_a \le r_{i,j} < r_b) P_i P_j e^{-\beta(r - r_{ij})^2}}, \quad (1)$$

where r_a and r_b are the boundaries of the current distance interval, N is the total number of atoms in the molecule, $r_{(i,j)}$ is the distance between the two atoms being considered, δ is the Kronecker delta, and P is the property computed for each atom.

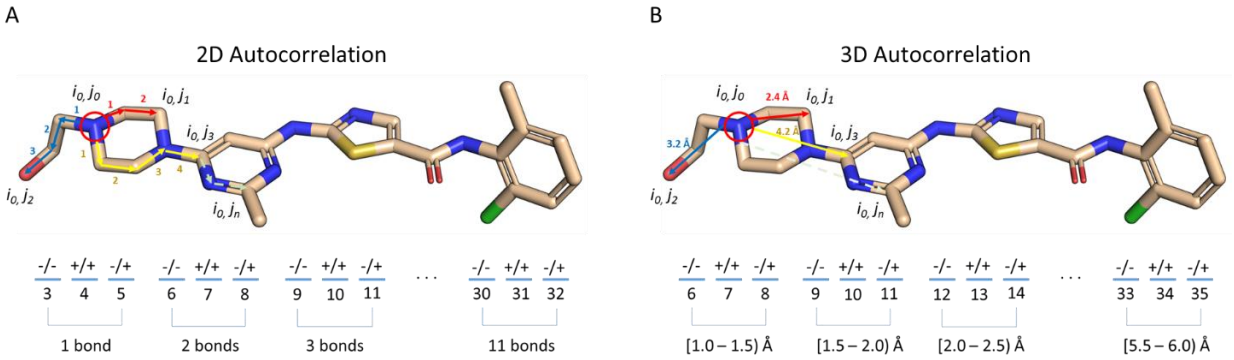

Figure 1. Illustration of signed autocorrelation descriptors. Signed autocorrelations are the sums of products of each atom property pair (e.g., i0,j2) in a distance bin defined by (A) bond separation, or (B) Euclidean distance in 3D space. Within each distance bin, atom property pairs are further separated into bins corresponding to the sign of the property of the first (left hand side of '/') and second (right hand side of '/') atoms in the pair.

2DAs are conformation-independent, while 3DAs are conformation-dependent. Let's look at how this difference affects output.

The "dasatinibs.sdf" file contains the coordinates and connectivity for two dasatinib molecules: one with 2D coordinates, the other with 3D coordinates. Compute the signed 2DA for Atom_SigmaCharge on both dasatinib molecules.

```
bcl.exe molecule:Properties \
-input_filenames dasatinibs.sdf.gz \
-tabulate "Combine(2DASmoothSign(property=Atom_SigmaCharge))" \
-output_table ../output/dasatinibs.2da_atom_sigmacharge.csv
```

There are two molecules in dasatinibs.sdf.gz, so the output will contain two non-header rows indexed 0 and 1 in the first column (where the first column is labeled "Index"). The second column has the label "Combine(2DASmoothSign(property=Atom_SigmaCharge))". Let's separate the two feature rows into different files for easier comparison.

```
tail -n+2 ../output/dasatinibs.2da_atom_sigmacharge.csv | \
awk -F, '{print $2}' | head -n1 > \
../output/dasatinibs.2da_atom_sigmacharge.2d.dat
```

```
tail -n+3 ../output/dasatinibs.2da_atom_sigmacharge.csv | \
awk -F, '{print $2}'  > \
../output/dasatinibs.2da_atom_sigmacharge.3d.dat
```

```
diff ../output/dasatinibs.2da_atom_sigmacharge.2d.dat \
../output/dasatinibs.2da_atom_sigmacharge.3d.dat
```

The diff should return absolutely nothing. This means that the two files have identical contents. In other words, the 2DA returns the same values independent of whether you have a molecule with a 2D conformation or a 3D conformation. Indeed, if the molecules are typed properly and described properly at the connectivity level (i.e., which atoms of what atom type are connected to one another via which type of bond), you can make useful 2DA features from severely distorted or under-optimized molecular structures (not that you should make a habit of this).

Compute the equivalent property using a 3DA.

```
bcl.exe molecule:Properties \
-input_filenames dasatinibs.sdf.gz \
-tabulate "Combine(3daSmoothSign(property=Atom_SigmaCharge))" \
-output_table ../output/dasatinibs.3da_atom_sigmacharge.csv
```

Run the same shell commands, but this time run them on the 3DA output. The diff will return the contents of both of your features because that line is different between the two structures. In other words, the 3DA is sensitive to the conformation of your structures, as expected.

You may then think to yourself "*If 3DAs return different results for different 3D conformers, how can I use them reliably? How can I train a model to learn differences in 3DAs between molecules if the 3DAs change even for different conformations of the same molecule?*"

And you would be thinking in the correct direction – this is an important point. So, let's do an experiment to determine how much noise, represented as property variance, is present in each sign-distance bin of a 3DA in a global conformational ensemble generated by BCL::Conf (see Tutorial 1 for a refresher).

For this experiment, we will use three different molecules with varying degrees of flexibility. We will use dasatinib, a tyrosine kinase inhibitor with 7 rotatable bonds, amprenavir is a HIV protease inhibitor with 12 rotatable bonds, and ethinyl estradiol is a synthetic estradiol with only 1 rotatable bond (most of its potential flexibility comes from different saturated ring conformers). First, generate conformers.

```
bcl.exe molecule:ConformerGenerator \
-ensemble_filenames amprenavir.sdf.gz \
-conformers_single_file ../output/amprenavir.confs.sdf.gz \
-max_iterations 8000 \
-top_models 250 \
-conformation_comparer SymmetryRMSD 0.25 \
-cluster \
-generate_3D
```

Note that for ethinyl estradiol you will have 250 conformers because there are not 250 unique valid conformers. For both dasatinib and amprenavir you will have 250.

We will use `descriptor:GenerateDataset` to compute the 3DAs across the conformers because it is a bit more convenient to manipulate for analysis. This will be a short preview of Part 2 below where we discuss dataset generation in more detail.

```
bcl.exe descriptor:GenerateDataset \
-source "SdfFile(filename=../output/amprenavir.confs.sdf.gz)" \
-feature_labels "Combine(3daSmoothSign(\
property=Atom_SigmaCharge,\
step size=0.25,temperature=100,steps=48,\
gaussian=1,interpolate=1))" \
-result_labels "Constant(999)" \
-output ../output/amprenavir.confs.3da.csv
```

Briefly, the signed 3DA allows you to specify the bin distance discretization such that you will have 3 (-/-, +/+, and -/+) sign pair bins at each distance. In this example, we are discretizing at 0.25 Angstroms. The temperature, gaussian, and interpolation settings are all related to smoothing between bins and are beyond the scope of this tutorial.

Use the provided short Python script to compute the variance of each column across all conformers (rows).

```
python ../scripts/var.py \
-input ../output/amprenavir.confs.3da.csv \
-output ../output/amprenavir.confs.3da.var.dat
```

Follow this procedure for each of the three molecules. At the end, plot their variances on the same figure using the other short Python script.

```
python ../scripts/plot_variance.py \
../output/dasatinib.confs.3da.var.dat \
../output/amprenavir.confs.3da.var.dat \
../output/ethinyl_estradiol.confs.3da.var.dat \
../output/3da_variance.png
```

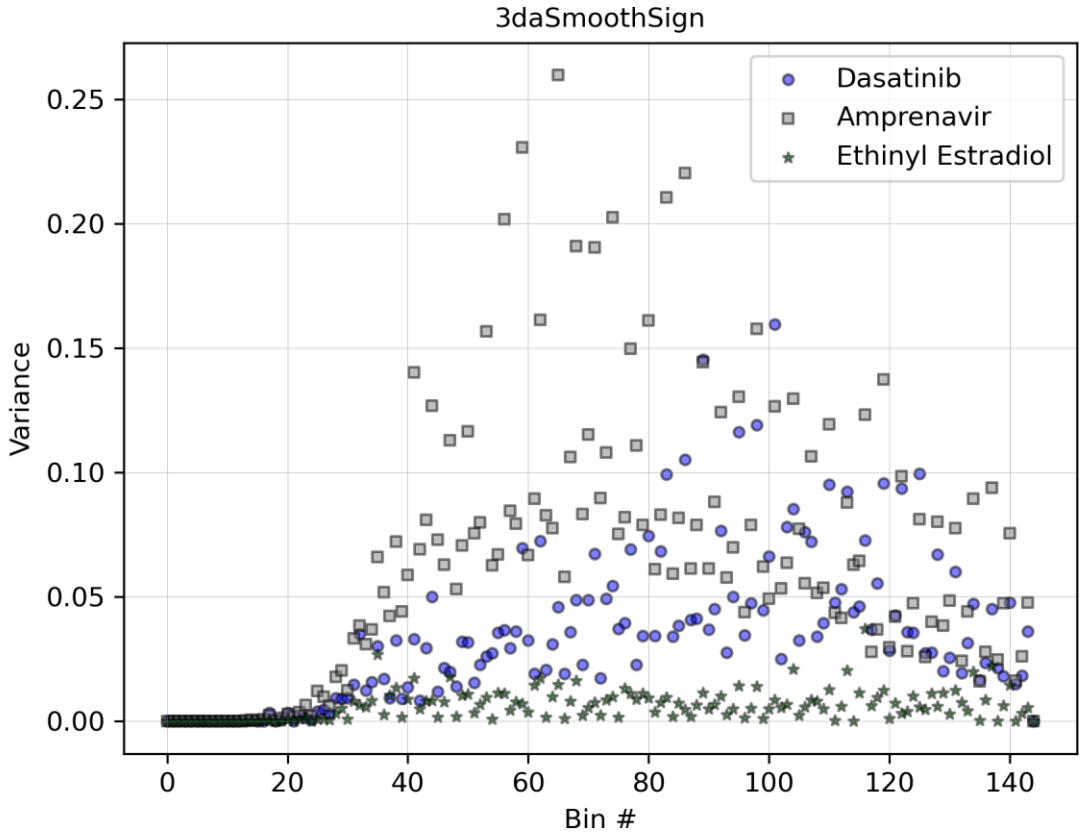

Figure 2. Signed 3DA variance increases with bin distance in flexible molecules. The 3daSmoothSign operation was performed on conformational ensembles of different small molecules using the Atom_SigmaCharge property with 48 steps at 0.25 Angstroms per step for a total distance of 12.0 Angstroms. There are 144 feature bins and one constant result label at the end. Interpolation and Gaussian smoothing were enabled with a temperature factor of 100.

What do you see? Are you surprised? All three of these molecules have approximately zero variance until about bin #30 or so. After bin #30, the variance seems to increase as a function of the flexibility of the molecule. Which distance bin does #30 correspond to? We are taking 48 steps. Each step is 0.25 Angstroms. Therefore, we have a total of 12.0 Angstroms that we are covering; however, each step has 3 bins, not 1, because we are accounting for the different sign pairings. Our total number of bins is therefore 144 (so 144 total bins / 4 bins per angstrom / 3 signs per bin = 12 Angstroms). We can compute the distance bin of bin #30 as 30 / 4 bins per angstrom / 3 signs per bin = 2.5 Angstroms.

It makes sense that there is virtually no variance at 2.5 Angstroms – at such short differences the atomic property pairings are likely conformation independent. Amprenavir is a peptidomimetic with more flexibility than most small molecules in which we would be interested. We see that by 5.0 Angstroms it has variance > 0.2. In contrast, dasatinib (which is not a small drug) never reaches a variance of that extent.

What can we conclude about using 3DAs? 3DAs can provide a signal with low noise in the absence of certainty regarding the biologically relevant conformation if you are careful in how you setup your training set. In the Meiler Lab, we generally have two rules of thumb for using 3DAs successfully:

(1) By default, we set maximum distances to 6.0 Angstroms for most QSAR tasks. You can perform experiments like these on training data to determine if 6.0 Angstroms is an appropriate cutoff. In some cases,

such as amprenavir, you may want to use a shorter cutoff. In other cases, such as ethinyl estradiol, you can use a larger cutoff. Typically, as in the case of dasatinib, 6.0 Angstroms is reasonable.

(2) Minimize stochasticity in conformer generation if the conformer will be used to compute a 3DA. Frequently, we use the external program CORINA[6] to generate conformers for 3DA calculations because it systematically applies the same rules to each molecule to generate the conformer (i.e., you probably will not come very close to the biologically relevant conformer, but you will generate all your conformers the same way). Alternatively, we generate conformational ensembles with BCL::Conf and compute our 3DA properties as an average across the entire ensemble.

We have described multiple uses of the `molecule:Properties` application, placing special emphasis on how it can be utilized to build different types of druglikeness metrics. We also began thinking about autocorrelations of properties as features for QSAR/QSPR models. In the next section, we will perform a deeper investigation of dataset generation in the BCL.

Part 2: Generate feature datasets from chemical properties

The descriptor application group is the workhorse for molecule featurization. Similar to the `molecule:Properties` application, the `descriptor` application group provides command-line access to the internal descriptor framework. Unlike `molecule`, `descriptor` is dataset centric; its primary purpose is to generate, manipulate, and analyze feature datasets for QSAR/QSPR. In this section, we will demonstrate core applications in descriptor and how they can be utilized in QSAR/QSPR modeling.

Four specifications are required to generate feature datasets from small molecules:

1. The molecules for which to generate the features; these can be any valid SDF.

2. The types of features to generate; these are properties such as those described in Part 1. Typically, these are stored in a separate file and passed to the command-line at run-time; however, they can also be specified directly on the command-line. Importantly, combining multiple descriptors for feature generation requires the use of the `Combine` operation.

3. The feature result label: this indicates the output(s) that models will train toward. This can be a constant value (i.e., if featurization is being done for some purpose other than model training), a property (e.g., LogP for a QSPR model), or another label (e.g., bioactivity label from experimental data for a QSAR model). Multiple result labels are allowed (see Tutorial 3 for more details).

4. The output filename: three output types are available. The BCL has a partial binary format with the ".bin" suffix that is used for all model training. Feature datasets can also be output with the ".csv" suffix for a comma-separated values (CSV) file. Moreover, ".csv" files and ".bin" files can be interconverted. In this way, features generated with the BCL can be used by other software, and vice versa. For inter-operability with Weka software, ".arff" format is also supported.

Generate a simple feature dataset consisting of several scalar descriptors for the Kir2.1 inward rectifying potassium channel using the dataset compiled in Butkiewicz et al. [7]. This dataset contains 301,493 small molecules, 172 of which are confirmed active molecules. The SDF corresponding to these compounds is 1843.combined.sdf.gz. These molecules have been labeled with the MDL property "IsActive" such that the confirmed actives have a value of 1 and the negatives have a value of 0.

```
bcl.exe descriptor:GenerateDataset \
-source "SdfFile(filename=1843.combined.sdf.gz)" \
-id_labels "String(KIR2)" \
-result_labels "Combine(IsActive)" \
-feature_labels "Combine(Weight,LogP,HbondDonor,HbondAcceptor)" \
-output 1843.combined.scalars.bin
```

Inspection of the output descriptor binary file shows that the first portion of the file is human readable. The first line has the heading `bcl::model::FeatureLabelSet` under which are all the descriptors passed via the `feature_labels` flag. The next heading is `bcl::storage::Vector<size_t>`. The first row after this header is the total number of descriptor labels, under which are the number of feature values corresponding to each descriptor label. In this case, there are 4 descriptors, and they all have a size of 1 because each descriptor specified with the `feature_labels` flag returns a single value. In contrast, property definitions have a size of 0 because they do not return any values, while autocorrelations often return tens or hundreds of values. The next header corresponds to the number of result labels and values per result. Finally, the last header indicates the number of characters indicated in the `id_labels` flag.

To better understand the binary file encodings, convert 1843.combined.scalars.bin to a CSV file:

```
bcl.exe descriptor:GenerateDataset \
-source "Subset(filename=1843.combined.scalars.bin)" \
-output 1843.combined.scalars.csv
```

The first column of every row contains the ID label "KIR2" as specified when the binary file was generated. Adding row labels is not typically necessary for datasets used in single-task learning; however, if you are performing multitask learning with non-overlapping training samples it can be very useful. The next four columns contain the descriptors specified above: Weight, LogP, HbondDonor, and HbondAcceptor. The very last column is the result value, which contains either 0 or 1 depending on the value in the SDF MDL property "IsActive".

Convert CSV file back to a binary file:

```
bcl.exe descriptor:GenerateDataset \
-source "Csv(filename=1843.combined.scalars.csv, number result
cols=1,number id chars=4)" \
-output 1843.combined.scalars.bin
```

CSV files do not contain all the supplementary information contained within the partial binary file format. Thus, certain information needs to be provided directly. For example, we need to specify the number of characters that are part of the row ID label, otherwise the BCL will try to convert the string (or numerical) ID into feature values. ID labels therefore must be fixed width. In addition, we need to tell the BCL how many of the columns are result values. By default, the BCL will assume that only the last column is the result label. By specifying number result cols=N, we tell the BCL to take the last N columns of the CSV as the result value(s).

Also notice that the feature and result label information is not informative after converting from CSV to binary. The values are transferred to the new file format, but the BCL obviously cannot know where those values came from. These must be manually specified.

```
bcl.exe descriptor:GenerateDataset \
-source "Csv(filename=1843.combined.scalars.csv, \
number result cols=1,number id chars=4)" \
-id_labels "String(KIR2)" \
-result_labels "Combine(IsActive)" \
-feature_labels "Combine(Weight,LogP,HbondDonor,HbondAcceptor)" \
-output 1843.combined.scalars.bin
```

In this case, the feature labels are internal parsable properties of the BCL; however, when relabeling feature labels upon converting from CSV to binary format, the user can specify any labels so long as the total number of labels is consistent with the number of feature columns.

After generating a dataset or importing a CSV file and converting it to binary format, feature datasets can be modified. The most frequent form of modification is randomization. Training a machine learning model, for example a neural network, often requires dataset randomization.

```
bcl.exe descriptor:GenerateDataset \
-source "Randomize(Subset(filename=1843.combined.scalars.bin))" \
-output 1843.combined.scalars.rand.bin
```

The `Randomize` operation is passed through the source flag and provided the dataset retriever option corresponding to the binary file.

Additional dataset operations can be generally classified by how they modify the dataset. For example, the PCA (principal components analysis) and `EncodeByModel` operations perform dimensionality reduction across feature (column) space, while the `KMeans` operation reduces dimensionality across molecule (row) space. Other operations are useful during model training and validation, such as `Balanced`, `Chunks`, and `YScrambled`. Still others can be used to increase the efficiency of generating very large datasets, such as `Rows`. Here, we will look at a few dataset operations.

For full details on all available dataset operations, see the `descriptor:GenerateDataset` help menu. For each molecule, there will be 1315 feature columns and 1 result column.

```
bcl.exe descriptor:GenerateDataset \
-source "SdfFile(filename=1843.combined.sdf.gz)" \
-scheduler PThread 8 \
-feature_labels MendenhallMeiler2015.Minimal.object \
-result_labels "Combine(IsActive)" \
-output 1843.Minimal.bin -logger File 1843.Minimal.log
```

Randomize the dataset:

```
bcl.exe descriptor:GenerateDataset \
-source "Randomize(Subset(filename=1843.combined.bin))" \
-output 1843.combined.rand.bin -logger File 1843.Minimal.rand.log
```

Note that we could have generated a randomized dataset with a single command by wrapping the `SdfFile` dataset retriever with `Randomize`; however, the `Randomize` dataset retriever is unable to support hyperthreading. Consequently, it is faster to generate larger datasets first using multiple threads and randomize them afterward. Next, perform PCA on the dataset using OpenCL to accelerate the calculation with a GPU. The `opencl` is optional and may not be supported on all platforms, but may provide a substantial speedup, depending on the GPU and dataset size:

```
bcl.exe descriptor:GeneratePCAEigenVectors \
-training "Subset(filename=1843.Minimal.rand.bin)" \
-output_filename 1843.Minimal.PCs.dat -opencl \
-logger File 1843.Minimal.PCs.log
```

Finally, generate a new feature dataset accounting for 95% of the variance:

```
bcl.exe descriptor:GenerateDataset \
-source "PCA(dataset=Subset(filename=1843.Minimal.rand.bin), fraction=0.95,
filename=1843.Minimal.PCs.dat)" \
-output 1843.Minimal.rand.pca_095.bin -opencl \
-logger File 1843.Minimal.rand.pca_095.log
```

Performing PCA on the dataset has reduced the number of descriptors from 1315 to 695. Alternatively, one could use EncodeByModel to reduce the number of feature columns using a pre-generated model. The following example utilizes pseudocode and a hypothetical pre-generated ANN with the "Mendenhall-Meiler2015.Minimal.object" features.

```
bcl.exe descriptor:GenerateDataset /
-source "EncodeByModel(storage=File(directory=/path/to/model/directory,\
 prefix=model),retriever=Subset(filename=<my_binary_file.bin>))" \
-output <my_encoded_binary_file.bin>
```

The input binary file would have 1315 descriptors from "MendenhallMeiler2015.Minimal.object", and the output binary file would have the number of descriptors corresponding to the number of neurons in the final hidden layer preceding the output layer of our hypothetical pre-generated ANN.

Suppose you encoded the same original feature set using two different models and now want to combine the new encoded files into one for further training. This can readily be accomplished with the `Combine` operation.

```
bcl.exe descriptor:GenerateDataset \
-source "Combined(Subset(filename=<my_binary_file_1.bin>),\
 Subset(filename=<my_binary_file_2.bin>))" \
-output <my_combined_binary_file.bin>
```

Next, instead of performing dimensionality reduction along the column (features) axis, we will reduce the dimensionality along the row (molecule) axis. Perform K-means clustering of the feature dataset to reduce our row number from 301,493 to 300.

```
bcl.exe descriptor:GenerateDataset \
-source "KMeans(dataset=Subset(filename=1843.combined.rand.bin), \
clusters=300)" \
-output 1843.combined.rand.k300.bin \
-logger File 1843.combined.rand.k300.log
```

This form of dimensionality reduction is unlikely to be as useful for training a deep neural network (DNN); however, it can be useful in similarity analysis in low dimensional feature space.

And that's it! Congratulations. You have completed the tutorial on computing chemical properties in the BCL! In Tutorial 3, we will use chemical property datasets to train QSAR models.

References

(1) Friedrich, N.-O.; Meyder, A.; de Bruyn Kops, C.; Sommer, K.; Flachsenberg, F.; Rarey, M.; Kirchmair, J. High-Quality Dataset of Protein-Bound Ligand Conformations and Its Application to Benchmarking Conformer Ensemble Generators. *J. Chem. Inf. Model.* **2017**, *57* (3), 529–539. https://doi.org/10.1021/acs.jcim.6b00613.

(2) Brown, B. P.; Vu, O.; Geanes, A. R.; Sandeepkumar, K.; Butkiewicz, M.; Lowe Jr, E. W.; Mueller, R.; Pape, R.; Mendenhall, J.; Meiler, J. Introduction to the BioChemical Library (BCL): An Application-Based Open-Source Toolkit for Integrated Cheminformatics and Machine Learning in Computer-Aided Drug Discovery.

(3) Bickerton, G. R.; Paolini, G. V.; Besnard, J.; Muresan, S.; Hopkins, A. L. Quantifying the Chemical Beauty of Drugs. *Nat. Chem.* **2012**, *4* (2), 90–98. https://doi.org/10.1038/nchem.1243.

(4) Sliwoski, G.; Kothiwale, S.; Meiler, J.; Lowe, E. W. Computational Methods in Drug Discovery. *Pharmacol. Rev.* **2014**, *66* (1), 334–395. https://doi.org/10.1124/pr.112.007336.

(5) Sliwoski, G.; Mendenhall, J.; Meiler, J. Autocorrelation Descriptor Improvements for QSAR: 2DA_Sign and 3DA_Sign. *J. Comput. Aided Mol. Des.* **2015**. https://doi.org/10.1007/s10822-015-9893-9.

(6) Gasteiger, J.; Rudolph, C.; Sadowski, J. Automatic Generation of 3D Atomic Coordinates for Organic Molecules. *Tetrahedron Computer Methodology* **1990**, *3* (6c), 537–547.

(7) Butkiewicz, M.; Lowe, E. W.; Mueller, R.; Mendenhall, J. L.; Teixeira, P. L.; Weaver, C. D.; Meiler, J. Benchmarking Ligand-Based Virtual High-Throughput Screening with the PubChem Database. *Molecules* **2013**, *18* (1), 735–756. https://doi.org/10.3390/molecules18010735.