

---

# **ATOM Data-Driven Modeling Pipeline Documentation**

***Release 1.6.0***

**ATOM DDM Team**

**May 01, 2024**



# CONTENTS

<b>1</b>	<b>Features</b>	<b>3</b>
<b>2</b>	<b>Built with</b>	<b>5</b>
<b>3</b>	<b>User guide</b>	<b>7</b>
<b>4</b>	<b>API documentation</b>	<b>13</b>
<b>5</b>	<b>Useful links</b>	<b>111</b>
	<b>Python Module Index</b>	<b>113</b>
	<b>Index</b>	<b>115</b>



**AMPL** is an open-source, modular, extensible software pipeline for building and sharing models to advance in silico drug discovery.

The ATOM Modeling PipeLine (AMPL) extends the functionality of DeepChem and supports an array of machine learning and molecular featurization tools. AMPL is an end-to-end data-driven modeling pipeline to generate machine learning models that can predict key safety and pharmacokinetic-relevant parameters. AMPL has been benchmarked on a large collection of pharmaceutical datasets covering a wide range of parameters.



## **FEATURES**

AMPL enables tasks for modeling and prediction from data ingestion to data analysis and can be broken down into the following stages:

- Data ingestion and curation
- Featurization
- Model training and tuning
- Prediction generation
- Visualization and analysis
- Details of running specific features are within the parameter (options) documentation.

More detailed documentation is in the library documentation.





## BUILT WITH

- [DeepChem](#): The basis for the graph convolution models
- [RDKit](#): Molecular informatics library
- [Mordred](#): Chemical descriptors
- Other Python package dependencies



A step-by-step guide to getting started with MolVS.

## 3.1 Getting started

### 3.1.1 Prerequisites

**AMPL** is a Python 3 package that has been developed and run in a specific pip environment.

### 3.1.2 Install

Clone the git repository:

```
git clone https://github.com/ATOMScience-org/AMPL.git
```

Please refer to this link, <https://github.com/ATOMScience-org/AMPL#Install>, for details.

Create pip environment:

```
module load python/3.8.2 # use python 3.8.2
python3 -m venv atomsci # create a new pip env
source atomsci/bin/activate # activate the environemt

python3 -m pip install pip --upgrade
cd $AMPL_HOME/pip # cd to AMPL repo's pip directory

pip3 install --force-reinstall --no-use-pep517 -r requirements.txt
```

---

**Note:** Depending on system performance, creating the environment can take some time.

---

## 3.2 Installation

### 3.2.1 Prerequisites

AMPL is a Python 3 package that has been developed and run in a specific pip environment.

### 3.2.2 Install

#### Clone the git repository

```
git clone https://github.com/ATOMScience-org/AMPL.git
```

#### Create pip environment

```
::
    cd $AMPL_HOME/pip # cd to AMPL repo's pip directory
    pip3 install --force-reinstall --no-use-pep517 -r requirements.txt
```

---

**Note:** Depending on system performance, creating the environment can take some time.

---

### 3.2.3 Install AMPL

Go to the AMPL root directory and install the AMPL package:

```
source atomsci/bin/activate # activate the environment
cd ..
./build.sh
pip3 install -e .
```

- The *install.sh* system command installs AMPL directly in the pip environment. If *install.sh* alone is used, then AMPL is installed in the *\$HOME/local* directory.
- After this process, you will have an *atomsci* pip environment with all dependencies installed. The name of the AMPL package is *atomsci-ampl* and is installed in the *install.sh* script to the environment with pip.

### 3.2.4 Install with Docker

- **Download and install Docker Desktop.**
  - [Docker Getting Started](#)
- Create a workspace folder to mount with Docker environment and transfer files.
- Get the Docker image and run it:

```
docker pull paulsonak/atomsci-ampl
docker run -it -p 8888:8888 -v </local_workspace_folder>:</directory_in_docker>
↳atomsci/atomsci-ampl
#inside docker environment
jupyter-notebook --ip=0.0.0.0 --allow-root --port=8888 &
# -OR-
jupyter-lab --ip=0.0.0.0 --allow-root --port=8888 &
```

- Visit the provided URL in your browser, ie
  - <http://d33b0faf6bc9:8888/?token=656b8597498b18db2213b1ec9a00e9d738dfe112bbe7566d>
  - Replace the d33b0faf6bc9 with localhost
  - If this doesn't work, exit the container and change port from 8888 to some other number such as 7777 or 8899 (in all 3 places it's written), then rerun both commands
- Be sure to save any work you want to be permanent in your workspace folder. If the container is shut down, you'll lose anything not in that folder.

### 3.3 Tests

AMPL includes a suite of software tests. This section explains how to run a very simple test that is fast to run. The Python test fits a random forest model using Mordred descriptors on a set of compounds from *Delaney, et al* with solubility data. A molecular scaffold-based split is used to create the training and test sets. In addition, an external holdout set is used to demonstrate how to make predictions on new compounds.

To run the *Delaney* Python script that curates a dataset, fits a model, and makes predictions, run the following commands:

```
source atomsci/bin/activate # activate your atomsci pip environment
cd atomsci/ddm/test/integrative/delaney_RF
pytest
```

---

**Note:** This test generally takes a few minutes on a modern system

---

The important files for this test are listed below:

- *test\_delaney\_RF.py*: This script loads and curates the dataset, generates a model pipeline object, and fits a model. The model is reloaded from the filesystem and then used to predict solubilities for a new dataset.
- *config\_delaney\_fit\_RF.json*: Basic parameter file for fitting
- *config\_delaney\_predict\_RF.json*: Basic parameter file for predicting

### 3.3.1 More example and test information

More details on examples and tests can be found in [Advanced testing](#).

## 3.4 Running AMPL

AMPL can be run from the command line or by importing into *Python* scripts and *Jupyter notebooks*.

### 3.4.1 Python scripts and Jupyter notebooks

AMPL can be used to fit and predict molecular activities and properties by importing the appropriate modules. See the examples for more descriptions on how to fit and make predictions using AMPL.

### 3.4.2 Pipeline parameters (options)

AMPL includes many parameters to run various model fitting and prediction tasks.

- Pipeline options (parameters) can be set within JSON files containing a parameter list.
- The parameter list with detailed explanations of each option can be found at [atom/ddm/docs/PARAMETERS.md](#).
- Example pipeline JSON files can be found in the tests directory and the example directory.

### 3.4.3 Library documentation

AMPL includes detailed docstrings and comments to explain the modules. Full HTML documentation of the Python library is available with the package at [atomsci/ddm/docs/build/html/index.html](#).

### 3.4.4 More information on AMPL usage

More information on AMPL usage can be found in [Advanced AMPL usage](#).

## 3.5 Advanced AMPL Usage

### 3.5.1 Command line

AMPL can *fit* models from the command line with:

```
python model_pipeline.py --config_file test.json
```

### 3.5.2 Hyperparameter optimization

Hyperparameter optimization for AMPL model fitting is available to run on SLURM clusters or with [HyperOpt](#). (Bayesian Optimization). To run Bayesian Optimization, the following steps can be followed.

See [Hyperparameter optimization](#) for more details.

## 3.6 Advanced Installation

### 3.6.1 Deployment

AMPL has been developed and tested on the following Linux systems:

- Red Hat Enterprise Linux 7 with SLURM
- Ubuntu 16.04

### 3.6.2 Uninstallation

To remove AMPL from a pip environment use:

```
deactivate
pip uninstall atomsci-ampl
```

To remove the atomsci pip environment entirely from a system use:

```
deactivate
cd $parent # to the parent of the atomsci `pip env` dir
rm -r atomsci
```

## 3.7 Advanced Testing

### 3.7.1 Running all tests

To run the full set of tests, use Pytest from the test directory:

```
source atomsci/bin/activate # activate your atomsci `pip env`
cd $AMPL_HOME/atomsci/ddm/test # your ampl repo
pytest
```

### 3.7.2 Running SLURM tests

Several of the tests take some time to fit. These tests can be submitted to a SLURM cluster as a batch job. Example general SLURM submit scripts are included as `pytest_slurm.sh`.

```
source atomsci/bin/activate
cd $AMPL_HOME/atomsci/ddm/test/integrative/delaney_NN
sbatch pytest_slurm.sh
cd ../../../../..
```

(continues on next page)

(continued from previous page)

```
cd $AMPL_HOME/atomsci/ddm/test/integrative/wenzel_NN
sbatch pytest_slurm.sh
```

### 3.7.3 Running tests without internet access

AMPL works without internet access. Curation, fitting, and prediction do not require internet access.

However, the public datasets used in tests and examples are not included in the repo due to licensing concerns. These are automatically downloaded when the tests are run.

If a system does not have internet access, the datasets will need to be downloaded before running the tests and examples. From a system with internet access, run the following shell script to download the public datasets. Then, copy the AMPL directory to the offline system.

```
cd atomsci/ddm/test
bash download_datset.sh
cd ../../..
# Copy AMPL directory to offline system
```



## API DOCUMENTATION

### 4.1 atomsci

#### 4.1.1 pipeline package

##### Submodules

##### pipeline.chem\_diversity module

Functions to generate matrices or vectors of distances between compounds

`pipeline.chem_diversity.calc_dist_diskdataset(feat_type, dist_met, dataset1, dataset2=None,  
calc_type='nearest', num_nearest=1, **metric_kwargs)`

Returns an array of distances, either between all compounds in a single dataset or between two datasets, given as DeepChem Dataset objects.

**Args:**

*feat\_type* (str): How the data was featurized. Current options are 'ECFP' or 'descriptors'.

*dist\_met* (str): What distance metric to use. Current options include tanimoto, cosine, cityblock, euclidean, or any other metric supported by `scipy.spatial.distance.pdist()`.

*dataset1* (deepchem.Dataset): Dataset containing features of compounds to be compared.

*dataset2* (deepchem.Dataset, optional): Second dataset, if two datasets are to be compared.

*calc\_type* (str): Type of summarization to perform on rows of distance matrix. See function `calc_summary` for options.

*num\_nearest* (int): Additional parameter for *calc\_types* nearest, *nth\_nearest* and *avg\_n\_nearest*.

*metric\_kwargs*: Additional arguments to be passed to functions that calculate metrics.

**Returns:**

`np.ndarray`: Vector or matrix of distances between feature vectors.

`pipeline.chem_diversity.calc_dist_feat_array(feat_type, dist_met, feat1, feat2=None,  
calc_type='nearest', num_nearest=1, **metric_kwargs)`

Returns a vector or array of distances, either between all compounds in a single dataset or between two datasets, given the feature matrices for the dataset(s).

**Args:**

*feat\_type* (str): How the data was featurized. Current options are 'ECFP' or 'descriptors'.

*dist\_met* (str): What distance metric to use. Current options include tanimoto, cosine, cityblock, euclidean, or any other metric supported by `scipy.spatial.distance.pdist()`.

feat1: feature matrix as a numpy array

feat2: Optional, second feature matrix

calc\_type (str): Type of summarization to perform on rows of distance matrix. See function calc\_summary for options.

num\_nearest (int): Additional parameter for calc\_types nearest, nth\_nearest and avg\_n\_nearest.

metric\_kwargs: Additional arguments to be passed to functions that calculate metrics.

**Returns:**

dists: vector or array of distances

```
pipeline.chem_diversity.calc_dist_smiles(feat_type, dist_met, smiles_arr1, smiles_arr2=None,  
                                         calc_type='nearest', num_nearest=1, **metric_kwargs)
```

Returns an array of distances between compounds given as SMILES strings, either between all pairs of compounds in a single dataset or between two datasets.

**Args:**

feat\_type (str): How the data is to be featurized, if dist\_met is not 'mcs'. The only option supported currently is 'ECFP'.

dist\_met (str): What distance metric to use. Current options include 'tanimoto' and 'mcs'.

smiles\_arr1 (list): First list of SMILES strings.

smiles\_arr2 (list): Optional, second list of SMILES strings. Can have only 1 member if wanting compound to matrix comparison.

calc\_type (str): Type of summarization to perform on rows of distance matrix. See function calc\_summary for options.

num\_nearest (int): Additional parameter for calc\_types nearest, nth\_nearest and avg\_n\_nearest.

metric\_kwargs: Additional arguments to be passed to functions that calculate metrics.

**Returns:**

dists: vector or array of distances

**Todo:**

Fix the function \_get\_descriptors(), which is broken, and re-enable the 'descriptors' option for feat\_type. Will need to add a parameter to indicate what kind of descriptors should be computed.

Allow other metrics for ECFP features, as in calc\_dist\_diskdataset().

```
pipeline.chem_diversity.calc_summary(dist_arr, calc_type, num_nearest=1, within_dset=False)
```

Returns a summary of the distances in dist\_arr, depending on calc\_type.

**Args:**

dist\_arr: (np.array): Either a 2D distance matrix, or a 1D condensed distance matrix (flattened upper triangle).

calc\_type (str): The type of summary values to return:

all: The distance matrix itself

nearest: The distances to the num\_nearest nearest neighbors of each compound (except compound itself)

nth\_nearest: The distance to the num\_nearest'th nearest neighbor

avg\_n\_nearest: The average of the num\_nearest nearest neighbor distances

farthest: The distance to the farthest neighbor

avg: The average of all distances for each compound

num\_nearest (int): Additional parameter for calc\_types nearest, nth\_nearest and avg\_n\_nearest.

within\_dset (bool): True if input distances are between compounds in the same dataset.

**Returns:**

dist (np.array): A numpy array of distances. For calc\_type 'nearest' with num\_nearest > 1, this is a 2D array with a row for each compound; otherwise it is a 1D array.

`pipeline.chem_diversity.upload_distmatrix_to_DS(dist_matrix, feature_type, compound_ids, bucket, title, description, tags, key_values, filepath='./', dataset_key=None)`

Uploads distance matrix in the data store with the appropriate tags

**Args:**

dist\_matrix (np.ndarray): The distance matrix.

feature\_type (str): How the data was featurized.

dist\_met (str): What distance metric was used.

compound\_ids (list): list of compound ids corresponding to the distance matrix (assumes that distance matrix is square and is the distance between all compounds in a dataset)

bucket (str): bucket the file will be put in

title (str): title of the file in (human friendly format)

description (str): long text box to describe file (background/use notes)

tags (list): List of tags to assign to datastore object.

key\_values (dict): Dictionary of key:value pairs to include in the datastore object's metadata.

filepath (str): local path where you want to store the pickled dataframe

**dataset\_key (str): If updating a file already in the datastore enter the corresponding dataset\_key.**

If not, leave as 'none' and the dataset\_key will be automatically generated.

**Returns:**

None

## pipeline.compare\_models module

Functions for comparing and visualizing model performance. Most of these functions rely on ATOM's model tracker and datastore services, which are not part of the standard AMPL installation, but a few functions will work on collections of models saved as local files.

`pipeline.compare_models.copy_best_filesystem_models(result_dir, dest_dir, pred_type, force_update=False)`

Identify the best models for each dataset within a result directory tree (e.g. from a hyperparameter search). Copy the associated model tarballs to a destination directory.

**Args:**

result\_dir (str): Path to model training result directory.

dest\_dir (str): Path of directory where model tarballs will be copied to.

pred\_type (str): Prediction type ('classification' or 'regression') of models to copy

force\_update (bool): If true, overwrite tarball files that already exist in dest\_dir.

**Returns:**

pd.DataFrame: Table of performance metrics for best models.

`pipeline.compare_models.del_ignored_params(dictionary, ignored_params)`

Deletes ignored parameters from the dictionary if they exist

**Args:**

dictionary (dict): A dictionary with parameters

ignored\_parameters (list(str)): A list of keys potentially in the dictionary

**Returns:**

None

`pipeline.compare_models.extract_collection_perf_metrics(collection_name, output_dir, pred_type='regression')`

Obtain list of training datasets with models in the given collection. Get performance metrics for models on each dataset and save them as CSV files in the given output directory.

**Args:**

collection\_name (str): Name of model tracker collection to search for models.

output\_dir (str): Directory where tables of performance metrics will be written.

pred\_type (str): Prediction type ('classification' or 'regression') of models to query.

**Returns:**

None

`pipeline.compare_models.extract_model_and_feature_parameters(metadata_dict)`

Given a config file, extract model and featurizer parameters. Looks for parameter names that end in `*_specific`. e.g. `nn_specific`, `auto_featurizer_specific`

**Args:**

model\_metadict (dict): Dictionary containing NON-FLATTENED metadata for an AMPL model

**Returns:**

dictionary containing featurizer and model parameters. Most contain the following keys. ['max\_epochs', 'best\_epoch', 'learning\_rate', 'layer\_sizes', 'dropouts', 'rf\_estimators', 'rf\_max\_features', 'rf\_max\_depth', 'xgb\_gamma', 'xgb\_learning\_rate', 'xgb\_max\_depth', 'xgb\_colsample\_bytree', 'xgb\_subsample', 'xgb\_n\_estimators', 'xgb\_min\_child\_weight', 'featurizer\_parameters\_dict', 'model\_parameters\_dict']

`pipeline.compare_models.get_best_models_info(col_names=None, bucket='public', pred_type='regression', result_dir=None, PK_pipeline=False, output_dir='/usr/local/data', shortlist_key=None, input_dset_keys=None, save_results=False, subset='valid', metric_type=None, selection_type='max', other_filters={})`

Tabulate parameters and performance metrics for the best models, according to a given metric, trained against each specified dataset.

**Args:**

col\_names (list of str): List of model tracker collections to search.

bucket (str): Datastore bucket for training datasets.

pred\_type (str): Type of models (regression or classification).

result\_dir (list of str): Result directories of the models, if model tracker is not supported.

PK\_pipeline (bool): Are we being called from PK pipeline?

output\_dir (str): Directory to write output table to.

shortlist\_key (str): Datastore key for table of datasets to query models for.

input\_dset\_keys (str or list of str): List of datastore keys for datasets to query models for. Either shortlist\_key or input\_dset\_keys must be specified, but not both.

save\_results (bool): If True, write the table of results to a CSV file.

subset (str): Input dataset subset ('train', 'valid', or 'test') for which metrics are used to select best models.

metric\_type (str): Type of performance metric (r2\_score, roc\_auc\_score, etc.) to use to select best models.

selection\_type (str): Score criterion ('max' or 'min') to use to select best models.

other\_filters (dict): Additional selection criteria to include in model query.

**Returns:**

top\_models\_df (DataFrame): Table of parameters and metrics for best models for each dataset.

`pipeline.compare_models.get_best_perf_table(metric_type, col_name=None, result_dir=None, model_uuid=None, metadata_dict=None, PK_pipe=False)`

Extract parameters and training run performance metrics for a single model. The model may be specified either by a metadata dictionary, a model\_uuid or a result directory; in the model\_uuid case, the function queries the model tracker DB for the model metadata. For models saved in the filesystem, can query the performance data from the original result directory, but not from a saved tarball.

**Args:**

metric\_type (str): Performance metric to include in result dictionary.

col\_name (str): Collection name containing model, if model is specified by model\_uuid.

result\_dir (str): result directory of the model, if Model tracker is not supported and metadata\_dict not provided.

model\_uuid (str): UUID of model to query, if metadata\_dict is not provided.

metadata\_dict (dict): Full metadata dictionary for a model, including training metrics and dataset metadata.

PK\_pipe (bool): If True, include some additional parameters in the result dictionary specific to PK models.

**Returns:**

model\_info (dict): Dictionary of parameter or metric name - value pairs.

**Todo:**

Add support for models saved as local tarball files.

`pipeline.compare_models.get_collection_datasets(collection_name)`

Returns a list of unique training datasets used for all models in a given collection.

**Args:**

collection\_name (str): Name of model tracker collection to search for models.

**Returns:**

list: List of model training (dataset\_key, bucket) tuples.

`pipeline.compare_models.get_dataset_models(collection_names, filter_dict={})`

Query the model tracker for all models saved in the model tracker DB under the given collection names. Returns a dictionary mapping (dataset\_key,bucket) pairs to the list of (collection,model\_uuid) pairs trained on the corresponding datasets.

**Args:**

collection\_names (list): List of names of model tracker collections to search for models.

filter\_dict (dict): Additional filter criteria to use in model query.

**Returns:**

dict: Dictionary mapping training set (dataset\_key, bucket) tuples to (collection, model\_uuid) pairs.

`pipeline.compare_models.get_filesystem_models(result_dir, pred_type)`

Identify all models in result\_dir and create perf\_result table with 'tarball\_path' column containing a path to each tarball.

`pipeline.compare_models.get_filesystem_perf_results(result_dir, pred_type='classification')`

Retrieve metadata and performance metrics for models stored in the filesystem from a hyperparameter search run.

**Args:**

result\_dir (str): Root directory for results from a hyperparameter search training run.

pred\_type (str): Prediction type ('classification' or 'regression') of models to query.

**Returns:**

pd.DataFrame: Table of metadata fields and performance metrics.

`pipeline.compare_models.get_multitask_perf_from_files(result_dir, pred_type='regression')`

Retrieve model metadata and performance metrics stored in the filesystem from a multitask hyperparameter search. Format the per-task performance metrics in a table with a row for each task and columns for each model/subset combination.

**Args:**

result\_dir (str): Path to root result directory containing output from a hyperparameter search run.

pred\_type (str): Prediction type ('classification' or 'regression') of models to query.

**Returns:**

pd.DataFrame: Table of model metadata fields and performance metrics.

`pipeline.compare_models.get_multitask_perf_from_files_new(result_dir, pred_type='regression')`

Retrieve model metadata and performance metrics stored in the filesystem from a multitask hyperparameter search. Format the per-task performance metrics in a table with a row for each task and columns for each model/subset combination.

**Args:**

result\_dir (str): Path to root result directory containing output from a hyperparameter search run.

pred\_type (str): Prediction type ('classification' or 'regression') of models to query.

**Returns:**

pd.DataFrame: Table of model metadata fields and performance metrics.

`pipeline.compare_models.get_multitask_perf_from_tracker(collection_name, response_cols=None, expand_responses=None, expand_subsets='test', exhaustive=False)`

Retrieve full metadata and metrics from model tracker for all models in a collection and format them into a table, including per-task performance metrics for multitask models.

Meant for multitask NN models, but works for single task models as well.

By AKP. Works for model tracker as of 10/2020

**Args:**

collection\_name (str): Name of model tracker collection to search for models.

**response\_cols (list, str or None): Names of tasks (response columns) to query performance results for.**

If None, checks to see if the entire collection has the same response cols. Otherwise, should be list of

strings or a comma-separated string. asks for clarification. Note: make sure response cols are listed in same order as in metadata. Recommended: None first, then clarify.

**expand\_responses (list, str or None): Names of tasks / response columns you want to include results for in**

the final dataframe. Useful if you have a lot of tasks and only want to look at the performance of a few of them. Must also be a list or comma separated string, and must be a subset of response\_cols. If None, will expand all responses.

**expand\_subsets (list, str or None): Dataset subsets ('train', 'valid' and/or 'test') to show metrics for.**

Again, must be list or comma separated string, or None to expand all.

**exhaustive (bool): If True, return large dataframe with all model tracker metadata minus any columns not**

in expand\_responses. If False, return trimmed dataframe with most relevant columns.

**Returns:**

pd.DataFrame: Table of model metadata fields and performance metrics.

`pipeline.compare_models.get_summary_metadata_table(uuids, collections=None)`

Tabulate metadata fields and performance metrics for a set of models identified by specific model\_uuids.

**Args:**

uuids (list): List of model UUIDs to query.

**collections (list or str): Names of collections in model tracker DB to get models from. If collections is**

a string, it must identify one collection to search for all models. If a list, it must be of the same length as uuids. If not provided, all collections will be searched.

**Returns:**

pd.DataFrame: Table of metadata fields and performance metrics for models.

`pipeline.compare_models.get_summary_perf_tables(collection_names=None, filter_dict={},  
result_dir=None, prediction_type='regression',  
verbose=False)`

Load model parameters and performance metrics from model tracker for all models saved in the model tracker DB under the given collection names (or result directory if Model tracker is not available) with the given prediction type. Tabulate the parameters and metrics including:

dataset (assay name, target, parameter, key, bucket) dataset size (train/valid/test/total) number of training folds model type (NN or RF) featurizer transformation type metrics: r2\_score, mae\_score and rms\_score for regression, or ROC AUC for classification

**Args:**

collection\_names (list): Names of model tracker collections to search for models.

filter\_dict (dict): Additional filter criteria to use in model query.

result\_dir (str or list): Directories to search for models; must be provided if the model tracker DB is not available.

prediction\_type (str): Type of models (classification or regression) to query.

verbose (bool): If true, print status messages as collections are processed.

**Returns:**

pd.DataFrame: Table of model metadata fields and performance metrics.

`pipeline.compare_models.get_tarball_perf_table(model_tarball, pred_type='classification')`

Retrieve model metadata and performance metrics for a model saved as a tarball (.tar.gz) file.

**Args:**

model\_tarball (str): Path of model tarball file, named as model.tar.gz.

pred\_type (str): Prediction type ('classification' or 'regression') of model.

**Returns:**

tuple (pd.DataFrame, dict): Table of performance metrics and a dictionary of model metadata.

`pipeline.compare_models.get_training_datasets(collection_names)`

Query the model tracker DB for all the unique dataset keys and buckets used to train models in the given collections.

**Args:**

collection\_names (list): List of names of model tracker collections to search for models.

**Returns:**

dict: Dictionary mapping collection names to lists of (dataset\_key, bucket) tuples for training sets.

`pipeline.compare_models.get_training_perf_table(dataset_key, bucket, collection_name, pred_type='regression', other_filters={})`

Load performance metrics from model tracker for all models saved in the model tracker DB under a given collection that were trained against a particular dataset. Identify training parameters that vary between models, and generate plots of performance vs particular combinations of parameters.

**Args:**

dataset\_key (str): Training dataset key.

bucket (str): Training dataset bucket.

collection\_name (str): Name of model tracker collection to search for models.

pred\_type (str): Prediction type ('classification' or 'regression') of models to query.

other\_filters (dict): Other filter criteria to use in querying models.

**Returns:**

pd.DataFrame: Table of models and performance metrics.

`pipeline.compare_models.num_trainable_parameters_from_file(tar_path)`

Return number of trainable parameters from tarfile

Given a tar file for a DeepChem model this will return the number of trainable parameters

**Args:**

tar\_path (str): Path to a DeepChem model

**Returns:**

int: Number of trainable parameters.

**Raises:**

ValueError: If the model is not a DeepChem neural network model



## pipeline.dist\_metrics module

Distance metrics for compounds: Tanimoto and maximum common substructure (MCS)

`pipeline.dist_metrics.mcs(mols1, mols2=None)`

Computes maximum common substructure (MCS) distances between pairs of molecules.

The MCS distance between molecules *m1* and *m2* is one minus the average of *fMCS*(*m1*,*m2*) and *fMCS*(*m2*,*m1*), where *fMCS*(*m1*,*m2*) is the fraction of *m1*'s atoms that are part of the largest common substructure of *m1* and *m2*.

### Args:

*mols1* (Sequence of *rdkit.Mol*): First list of molecules.

***mols2* (Sequence of *rdkit.Mol*, optional): Second list of molecules.**

If not provided, computes MCS distances between pairs of molecules in *mols1*. Otherwise, computes a matrix of distances between pairs of molecules from *mols1* and *mols2*.

### Returns:

np.ndarray: Matrix of pairwise distances between molecules.

`pipeline.dist_metrics.tanimoto(fps1, fps2=None)`

Compute Tanimoto distances between sets of ECFP fingerprints.

### Args:

*fps1* (Sequence): First list of ECFP fingerprint vectors.

***fps2* (Sequence, optional): Second list of ECFP fingerprint vectors.**

If not provided, computes distances between pairs of fingerprints in *fps1*. Otherwise, computes a matrix of distances between pairs of fingerprints in *fps1* and *fps2*.

### Returns:

np.ndarray: Matrix of pairwise distances between fingerprints.

`pipeline.dist_metrics.tanimoto_single(fp, fps)`

Compute a vector of Tanimoto distances between a single fingerprint and each fingerprint in a list .

### Args:

*fp* : Fingerprint to be compared.

*fps* (Sequence): List of ECFP fingerprint vectors.

### Returns:

np.ndarray: Vector of distances between *fp* and each fingerprint in *fps*.

## pipeline.diversity\_plots module

Plotting routines for visualizing chemical diversity of datasets

`pipeline.diversity_plots.diversity_plots(dset_key, datastore=True, bucket='public', title_prefix=None, ecfp_radius=4, umap_file=None, out_dir=None, id_col='compound_id', smiles_col='rdkit_smiles', is_base_smiles=False, response_col=None, max_for_mcs=300, colorpal=None)`

Plot visualizations of diversity for an arbitrary table of compounds. At minimum, the file should contain columns for a compound ID and a SMILES string. Produces a clustered heatmap display of Tanimoto distances between compounds along with a 2D UMAP projection plot based on ECFP fingerprints, with points colored according to the response variable.

**Args:**

dset\_key (str): Datastore key or filepath for dataset.

datastore (bool): Whether to load dataset from datastore or from filesystem.

bucket (str): Name of datastore bucket containing dataset.

title\_prefix (str): Prefix for plot titles.

ecfp\_radius (int): Radius for ECFP fingerprint calculation.

umap\_file (str, optional): Path to file to write UMAP coordinates to.

**out\_dir (str, optional): Output directory for plots and tables. If provided, plots will be output as PDF files rather**

than in the current notebook, and some additional CSV files will be generated.

id\_col (str): Column in dataset containing compound IDs.

smiles\_col (str): Column in dataset containing SMILES strings.

is\_base\_smiles (bool): True if SMILES strings do not need to be salt-stripped and standardized.

response\_col (str): Column in dataset containing response values.

**max\_for\_mcs (int): Maximum dataset size for plots based on MCS distance. If the number of compounds is less than this**

value, an additional cluster heatmap and UMAP projection plot will be produced based on maximum common substructure distance.

`pipeline.diversity_plots.plot_dataset_dist_distr(dataset, feat_type, dist_metric, task_name, **metric_kwargs)`

Generate a density plot showing the distribution of distances between dataset feature vectors, using the specified feature type and distance metric.

**Args:**

dataset (deepchem.Dataset): A dataset object. At minimum, it should contain a 2D numpy array 'X' of feature vectors.

feat\_type (str): Type of features ('ECFP' or 'descriptors').

dist\_metric (str): Name of metric to be used to compute distances; can be anything supported by `scipy.spatial.distance.pdist`.

task\_name (str): Abbreviated name to describe dataset in plot title.

metric\_kwargs: Additional arguments to pass to metric.

**Returns:**

np.ndarray: Distance matrix.

`pipeline.diversity_plots.plot_tani_dist_distr(df, smiles_col, df_name, radius=2, subset_col='subset', subsets=False, ref_subset='train', plot_width=6, ndist_max=None, **metric_kwargs)`

Generate a density plot showing the distribution of nearest neighbor distances between ecfp feature vectors, using the Tanimoto metric. Optionally split by subset.

**Args:**

df (DataFrame): A data frame containing, at minimum, a column of SMILES strings.

smiles\_col (str): Name of the column containing SMILES strings.

df\_name (str): Name for the dataset, to be used in the plot title.

radius (int): Radius parameter used to calculate ECFP fingerprints. The default is 2, meaning that ECFP4 fingerprints are calculated.

subset\_col (str): Name of the column containing subset names.

subsets (bool): If True, distances are only calculated for compounds not in the reference subset, and the distances computed are to the nearest neighbors in the reference subset.

ref\_subset (str): Reference subset for nearest-neighbor distances, if *subsets* is True.

plot\_width (float): Plot width in inches.

ndist\_max (int): Not used, included only for backward compatibility.

metric\_kwargs: Additional arguments to pass to metric. Not used, included only for backward compatibility.

#### Returns:

dist (DataFrame): Table of individual nearest-neighbor Tanimoto distance values. If subsets is True, the table will include a column indicating the subset each compound belongs to.

## pipeline.feature\_importance module

Functions to assess feature importance in AMPL models

`pipeline.feature_importance.base_feature_importance(model_pipeline=None, params=None)`

Minimal baseline feature importance function. Given an AMPL model (or the parameters to train a model), returns a data frame with a row for each feature. The columns of the data frame depend on the model type and prediction type. If the model is a binary classifier, the columns include t-statistics and p-values for the differences between the means of the active and inactive compounds. If the model is a random forest, the columns will include the mean decrease in impurity (MDI) of each feature, computed by the scikit-learn **feature\_importances\_** function. See the scikit-learn documentation for warnings about interpreting the MDI importance. For all models, the returned data frame will include feature names, means and standard deviations for each feature.

This function has been tested on RFs and NNs with rdkit descriptors. Other models and feature combinations may not be supported.

#### Args:

model\_pipeline (*ModelPipeline*): A pipeline object for a model that was trained in the current Python session or loaded from the model tracker or a tarball file. Either model\_pipeline or params must be provided.

params (*dict*): Parameter dictionary for a model to be trained and analyzed. Either model\_pipeline or a params argument must be passed; if both are passed, params is ignored and the parameters from model\_pipeline are used.

#### Returns:

(imp\_df, model\_pipeline, pparams) (tuple):

imp\_df (*DataFrame*): Table of feature importance metrics. model\_pipeline (*ModelPipeline*): Pipeline object for model that was passed to or trained by function. pparams (*Namespace*): Parsed parameters of model.

`pipeline.feature_importance.cluster_permutation_importance(model_pipeline=None, params=None, score_type=None, clust_height=1, result_file=None, nreps=10, nworkers=1)`

Divide the input features used in a model into correlated clusters, then assess the importance of the features by iterating over clusters, permuting the values of all the features in the cluster, and measuring the effect on the model performance metric given by score\_type for the training, validation and test subsets.

**Args:**

`model_pipeline` (*ModelPipeline*): A pipeline object for a model that was trained in the current Python session or loaded from the model tracker or a tarball file. Either `model_pipeline` or `params` must be provided.

`params` (*dict*): Parameter dictionary for a model to be trained and analyzed. Either `model_pipeline` or a `params` argument must be passed; if both are passed, `params` is ignored and the parameters from `model_pipeline` are used.

`clust_height` (float): Height at which to cut the dendrogram branches to split features into clusters.

`result_file` (str): Path to a CSV file where a table of features and cluster indices will be written.

`nreps` (int): Number of repetitions of the permutation and rescoring procedure to perform for each feature; the importance values returned will be averages over repetitions. More repetitions will yield better importance estimates at the cost of greater computing time.

`nworkers` (int): Number of parallel worker threads to use for permutation and rescoring. Currently ignored; multithreading will be added in a future version.

**Returns:**

`imp_df` (DataFrame): Table of feature clusters and importance values

```
pipeline.feature_importance.display_feature_clusters(model_pipeline=None, params=None,
                                                    clust_height=1, corr_file=None,
                                                    show_matrix=False, show_dendro=True)
```

Cluster the input features used in the model specified by `model_pipeline` or `params`, using Spearman correlation as a similarity metric. Display a dendrogram and/or a correlation matrix heatmap, so the user can decide the height at which to cut the dendrogram in order to split the features into clusters, for input to `cluster_permutation_importance`.

**Args:**

`model_pipeline` (*ModelPipeline*): A pipeline object for a model that was trained in the current Python session or loaded from the model tracker or a tarball file. Either `model_pipeline` or `params` must be provided.

`params` (*dict*): Parameter dictionary for a model to be trained and analyzed. Either `model_pipeline` or a `params` argument must be passed; if both are passed, `params` is ignored and the parameters from `model_pipeline` are used.

`clust_height` (float): Height at which to draw a cut line in the dendrogram, to show how many clusters will be generated.

`corr_file` (str): Path to an optional CSV file to be created containing the feature correlation matrix.

`show_matrix` (bool): If True, plot a correlation matrix heatmap.

`show_dendro` (bool): If True, plot the dendrogram.

**Returns:**

`corr_linkage` (np.ndarray): Linkage matrix from correlation clustering

```
pipeline.feature_importance.permutation_feature_importance(model_pipeline=None, params=None,
                                                           score_type=None, nreps=60,
                                                           nworkers=1, result_file=None)
```

Assess the importance of each feature used by a trained model by permuting the values of each feature in succession in the training, validation and test sets, making predictions, computing performance metrics, and measuring the effect of scrambling each feature on a particular metric.

**Args:**

`model_pipeline` (*ModelPipeline*): A pipeline object for a model that was trained in the current Python session or loaded from the model tracker or a tarball file. Either `model_pipeline` or `params` must be provided.

*params* (*dict*): Parameter dictionary for a model to be trained and analyzed. Either *model\_pipeline* or a *params* argument must be passed; if both are passed, *params* is ignored and the parameters from *model\_pipeline* are used.

*score\_type* (*str*): Name of the scoring metric to use to assess importance. This can be any of the standard values supported by `sklearn.metrics.get_scorer`; the AMPL-specific values 'npv', 'mcc', 'kappa', 'mae', 'rmse', 'ppv', 'cross\_entropy', 'bal\_accuracy' and 'avg\_precision' are also supported. Score types for which smaller values are better, such as 'mae', 'rmse' and 'cross\_entropy' are mapped to their negative counterparts.

*nreps* (*int*): Number of repetitions of the permutation and rescoreing procedure to perform for each feature; the importance values returned will be averages over repetitions. More repetitions will yield better importance estimates at the cost of greater computing time.

*nworkers* (*int*): Number of parallel worker threads to use for permutation and rescoreing.

*result\_file* (*str*): Optional path to a CSV file to which the importance table will be written.

#### Returns:

*imp\_df* (*DataFrame*): Table of features and importance metrics. The table will include the columns returned by *base\_feature\_importance*, along with the permutation importance scores for each feature for the training, validation and test subsets.

`pipeline.feature_importance.plot_feature_importances(imp_df, importance_col='valid_perm_importance_mean', max_feat=20, ascending=False)`

Display a horizontal bar plot showing the relative importances of the most important features or feature clusters, according to the results of *permutation\_feature\_importance*, *cluster\_permutation\_importance* or a similar function.

#### Args:

*imp\_df* (*DataFrame*): Table of results from *permutation\_feature\_importance*, *cluster\_permutation\_importance*, *base\_feature\_importance* or a similar function.

*importance\_col* (*str*): Name of the column in *imp\_df* to plot values from.

*max\_feat* (*int*): The maximum number of features or feature clusters to plot values for.

*ascending* (*bool*): Should the features be ordered by ascending values of *importance\_col*? Defaults to False; can be set True for p-values or something else where small values mean greater importance.

#### Returns:

None

## pipeline.hyper\_perf\_plots module

Functions for visualizing hyperparameter performance. These functions work with a dataframe of model performance metrics and hyperparameter specifications from `compare_models.py`. For models on the tracker, use `get_multitask_perf_from_tracker()`. For models in the file system, use `get_filesystem_perf_results()`. By Amanda P. 7/19/2022

`pipeline.hyper_perf_plots.get_score_types()`

Helper function to show score type choices.

`pipeline.hyper_perf_plots.plot_nn_perf(df, scoretype='r2_score', subset='valid')`

This function plots scatterplots of performance scores based on their NN hyperparameters.

#### Args:

*df* (*pd.DataFrame*): A dataframe containing model performances from a hyperparameter search. Best practice is to use `get_multitask_perf_from_tracker()` or `get_filesystem_perf_results()`.

scoretype (str): the score type you want to use. Valid options can be found in `hpp.classselmets` or `hpp.regselmets`.

subset (str): the subset of scores you'd like to plot from 'train', 'valid' and 'test'.

`pipeline.hyper_perf_plots.plot_rf_nn_xg_perf(df, scoretype='r2_score', subset='valid')`

This function plots boxplots of performance scores based on their hyperparameters including RF, NN and XG-Boost parameters as well as feature types, model types and ECFP radius.

**Args:**

df (pd.DataFrame): A dataframe containing model performances from a hyperparameter search. Best practice is to use `get_multitask_perf_from_tracker()` or `get_filesystem_perf_results()`.

scoretype (str): the score type you want to use. Valid options can be found in `hpp.classselmets` or `hpp.regselmets`.

subset (str): the subset of scores you'd like to plot from 'train', 'valid' and 'test'.

`pipeline.hyper_perf_plots.plot_rf_perf(df, scoretype='r2_score', subset='valid')`

This function plots scatterplots of performance scores based on their RF hyperparameters.

**Args:**

df (pd.DataFrame): A dataframe containing model performances from a hyperparameter search. Best practice is to use `get_multitask_perf_from_tracker()` or `get_filesystem_perf_results()`.

scoretype (str): the score type you want to use. Valid options can be found in `hpp.classselmets` or `hpp.regselmets`.

subset (str): the subset of scores you'd like to plot from 'train', 'valid' and 'test'.

`pipeline.hyper_perf_plots.plot_split_perf(df, scoretype='r2_score', subset='valid')`

This function plots boxplots of performance scores based on the splitter type.

**Args:**

df (pd.DataFrame): A dataframe containing model performances from a hyperparameter search. Best practice is to use `get_multitask_perf_from_tracker()` or `get_filesystem_perf_results()`.

scoretype (str): the score type you want to use. Valid options can be found in `hpp.classselmets` or `hpp.regselmets`.

subset (str): the subset of scores you'd like to plot from 'train', 'valid' and 'test'.

`pipeline.hyper_perf_plots.plot_train_valid_test_scores(df, scoretype='r2_score')`

This function plots kde and line plots of performance scores based on their partitions.

**Args:**

df (pd.DataFrame): A dataframe containing model performances from a hyperparameter search. Best practice is to use `get_multitask_perf_from_tracker()` or `get_filesystem_perf_results()`.

scoretype (str): the score type you want to use. Valid options can be found in `hpp.classselmets` or `hpp.regselmets`.

`pipeline.hyper_perf_plots.plot_xg_perf(df, scoretype='r2_score', subset='valid')`

This function plots scatterplots of performance scores based on their XG hyperparameters.

**Args:**

df (pd.DataFrame): A dataframe containing model performances from a hyperparameter search. Best practice is to use `get_multitask_perf_from_tracker()` or `get_filesystem_perf_results()`.

scoretype (str): the score type you want to use. Valid options can be found in `hpp.classselmets` or `hpp.regselmets`.

subset (str): the subset of scores you'd like to plot from 'train', 'valid' and 'test'.

## pipeline.model\_pipeline module

Contains class ModelPipeline, which loads in a dataset, splits it, trains a model, and generates predictions and output metrics for that model. Works for a variety of featurizers, splitters and other parameters on a generic dataset

**class** pipeline.model\_pipeline.**ModelPipeline**(*params, ds\_client=None, mlmt\_client=None*)

Bases: object

Contains methods to load in a dataset, split and featurize the data, fit a model to the train dataset, generate predictions for an input dataset, and generate performance metrics for these predictions.

### Attributes:

#### Set in `__init__`:

`params` (argparse.Namespace): The argparse.Namespace parameter object

`log` (log): The logger

`run_mode` (str): A flag determine the mode of model pipeline (eg. training or prediction)

`params.dataset_name` (argparse.Namespace): The dataset\_name parameter of the dataset

`ds_client` (ac.DatastoreClient): the datastore api token to interact with the datastore

`perf_dict` (dict): The performance dictionary

`output_dir` (str): The parent path of the model directory

`mlmt_client`: The mlmt service client

`metric_type` (str): Defines the type of metric (e.g. roc\_auc\_score, r2\_score)

#### set in `train_model` or `run_predictions`:

`run_mode` (str): The mode to run the pipeline, set to training

`featurization` (Featurization object): The featurization argument or the featurization created from the input parameters

`model_wrapper` (ModelWrapper object): A model wrapper created from the parameters and featurization object.

#### set in `create_model_metadata`:

`model_metadata` (dict): The model metadata dictionary that stores the model metrics and metadata

#### Set in `load_featurize_data`

`data` (ModelDataset object): A data object that featurizes and splits the dataset

**calc\_train\_dset\_pair\_dis**(*metric='euclidean'*)

Calculate the pairwise distance for training set compound feature vectors, needed for AD calculation.

**create\_model\_metadata**()

Initializes a data structure describing the current model, to be saved in the model zoo. This should include everything necessary to reproduce a model run.

#### Side effects:

Sets `self.model_metadata` (dictionary): A dictionary of the model metadata required to recreate the model. Also contains metadata about the generating dataset.

**create\_prediction\_metadata**(*prediction\_results*)

Initializes a data structure to hold performance metrics from a model run on a new dataset, to be stored in the model tracker DB. Note that this isn't used for the training run metadata; the `training_metrics` section is created by the `train_model()` function.

**Returns:**

prediction\_metadata (dict): A dictionary of the metadata for a model run on a new dataset.

**get\_metrics()**

Retrieve the model performance metrics from any previous training and prediction runs from the model tracker

**load\_featurize\_data(params=None)**

Loads the dataset from the datastore or the file system and featurizes it. If we are training a new model, split the dataset into training, validation and test sets.

The data is also split into training, validation, and test sets and saved to the filesystem or datastore.

Assumes a ModelWrapper object has already been created.

**Args:**

params (Namespace): Optional set of parameters to be used for featurization; by default this function uses the parameters used when the pipeline was created.

**Side effects:**

**Sets the following attributes of the ModelPipeline**

**data (ModelDataset object): A data object that featurizes and splits the dataset**

data.dataset(dc.DiskDataset): The transformed, featurized, and split dataset

**predict\_embedding(dset\_df, dset\_params=None)**

Compute embeddings from a pretrained model on a set of compounds listed in a data frame. The data frame should contain, at minimum, a column of compound IDs and a column of SMILES strings.

**predict\_full\_dataset(dset\_df, is\_featurized=False, contains\_responses=False, dset\_params=None, AD\_method=None, k=5, dist\_metric='euclidean', max\_train\_records\_for\_AD=1000)**

Compute predicted responses from a pretrained model on a set of compounds listed in a data frame. The data frame should contain, at minimum, a column of compound IDs; if SMILES strings are needed to compute features, they should be provided as well. Feature columns may be provided as well. If response columns are included in the input, they will be included in the output as well to facilitate performance metric calculations.

This function is similar to predict\_on\_dataframe, except that it supports multitask models, and includes class probabilities in the output for classifier models.

**Args:**

dset\_df (DataFrame): A data frame containing compound IDs (if the compounds are to be featurized using descriptors) and/or SMILES strings (if the compounds are to be featurized using ECFP fingerprints or graph convolution) and/or precomputed features. The column names for the compound ID and SMILES columns should match id\_col and smiles\_col, respectively, in the model parameters.

is\_featurized (bool): True if dset\_df contains precomputed feature columns. If so, dset\_df must contain *all* of the feature columns defined by the featurizer that was used when the model was trained.

contains\_responses (bool): True if dataframe contains response values

dset\_params (Namespace): Parameters used to interpret dataset, including id\_col, smiles\_col, and optionally, response\_cols. If not provided, id\_col, smiles\_col and response\_cols are assumed to be same as in the pretrained model.

AD\_method (str or None): Method to use to compute applicability domain (AD) index; may be 'z\_score', 'local\_density' or None (the default). With the default value, AD indices will not be calculated.



`k` (int): Number of nearest neighbors of each training data point used to evaluate the AD index.

`dist_metric` (str): Metric used to compute distances between feature vectors for AD index calculation. Valid values are 'cityblock', 'cosine', 'euclidean', 'jaccard', and 'manhattan'. If binary features such as fingerprints are used in model, 'jaccard' (equivalent to Tanimoto distance) may be a better choice than the other metrics which operate on continuous features.

`max_train_records_for_AD` (int): Maximum number of training data rows to use for AD calculation. Note that the AD calculation time scales as the square of the number of training records used. If the training dataset is larger than `max_train_records_for_AD`, a random sample of rows with this size is used instead for the AD calculations.

**Returns:**

`result_df` (DataFrame): Data frame indexed by compound IDs containing a column of SMILES strings, with additional columns containing the predicted values for each response variable. If the model was trained to predict uncertainties, the returned data frame will also include standard deviation columns (named `<response_col>_std`) for each response variable. The result data frame may not include all the compounds in the input dataset, because the featurizer may not be able to featurize all of them.

**`predict_on_dataframe`**(*dset\_df*, *is\_featurized=False*, *contains\_responses=False*, *AD\_method=None*, *k=5*, *dist\_metric='euclidean'*)

DEPRECATED Call `predict_full_dataset` instead.

**`predict_on_smiles`**(*smiles*, *verbose=False*, *AD\_method=None*, *k=5*, *dist\_metric='euclidean'*)

Compute predicted responses from a pretrained model on a set of compounds given as a list of SMILES strings.

**Args:**

`smiles` (list): A list containing valid SMILES strings

`verbose` (boolean): A switch for disabling informational messages

`AD_method` (str or None): Method to use to compute applicability domain (AD) index; may be 'z\_score', 'local\_density' or None (the default). With the default value, AD indices will not be calculated.

`k` (int): Number of nearest neighbors of each training data point used to evaluate the AD index.

`dist_metric` (str): Metric used to compute distances between feature vectors for AD index calculation. Valid values are 'cityblock', 'cosine', 'euclidean', 'jaccard', and 'manhattan'. If binary features such as fingerprints are used in model, 'jaccard' (equivalent to Tanimoto distance) may be a better choice than the other metrics which operate on continuous features.

**Returns:**

`res` (DataFrame): Data frame indexed by compound IDs containing a column of SMILES strings, with additional columns containing the predicted values for each response variable. If the model was trained to predict uncertainties, the returned data frame will also include standard deviation columns (named `<response_col>_std`) for each response variable. The result data frame may not include all the compounds in the input dataset, because the featurizer may not be able to featurize all of them.

**`run_predictions`**(*featurization=None*)

Instantiate a previously trained model, and use it to run predictions on a new dataset.

Generate predictions for a specified dataset, and save the predictions and performance metrics in the model results DB or in a JSON file.

**Args:**

`featurization` (Featurization Object): An optional featurization object for creating the model wrapper

**Side effects:**

**Sets the following attributes of ModelPipeline:**

run\_mode (str): The mode to run the pipeline, set to prediction

featurization (Featurization object): The featurization argument or the featurization created from the input parameters

model\_wrapper (ModelWrapper object): A model wrapper created from the parameters and featurization object.

**save\_metrics**(*model\_metrics*, *prefix=None*, *retries=5*, *sleep\_sec=60*)

Saves the given model\_metrics dictionary to a JSON file on disk, and also to the model tracker database if we're using it.

If writing to disk, outputs to a JSON file <prefix>\_model\_metrics.json in the current output directory.

**Args:**

model\_metrics (dict or list): Either a dictionary containing the model performance metrics, or a list of dictionaries with metrics for each training label and subset.

prefix (str): An optional prefix to include in the JSON filename

retries (int): Number of retries to save to model tracker DB, if save\_results is True.

sleep\_sec (int): Number of seconds to sleep between retries.

**Side effects:**

Saves the model\_metrics dictionary to the model tracker database, or writes out a .json file

**save\_model\_metadata**(*retries=5*, *sleep\_sec=60*)

Saves the data needed to reload the model in the model tracker DB or in a local tarball file.

Inserts the model metadata into the model tracker DB, if self.params.save\_results is True. Otherwise, saves the model metadata to a local .json file. Generates a gzipped tar archive containing the metadata file, the transformer parameters and the model checkpoint files, and saves it in the datastore or the filesystem according to the value of save\_results.

**Args:**

retries (int): Number of times to retry saving to model tracker DB.

sleep\_sec (int): Number of seconds to sleep between retries, if saving to model tracker DB.

**Side effects:**

Saves the model metadata and parameters into the model tracker DB or a local tarball file.

**split\_dataset**(*featurization=None*)

Load, featurize and split the dataset according to the current model parameter settings, but don't actually train a model. Returns the split\_uuid for the dataset split.

**Args:**

featurization (Featurization object): An optional featurization object.

**Returns:**

split\_uuid (str): The unique identifier for the dataset split.

**train\_model**(*featurization=None*)

Build model described by self.params on the training dataset described by self.params.

Generate predictions for the training, validation, and test datasets, and save the predictions and performance metrics in the model results DB or in a JSON file.

**Args:**

featurization (Featurization object): An optional featurization object for creating models on a prefeaturized dataset

**Side effects:**

**Sets the following attributes of the ModelPipeline object**

run\_mode (str): The mode to run the pipeline, set to training

featurization (Featurization object): The featurization argument or the featurization created from the input parameters

model\_wrapper (ModelWrapper obj): A model wrapper created from the parameters and featurization object.

model\_metadata (dict): The model metadata dictionary that stores the model metrics and metadata

`pipeline.model_pipeline.build_dataset_name(dataset_key)`

Return the dataset\_name when given a dataset\_key. Assumes that the dataset\_name is a path and ends with an extension

**Args:**

dataset\_key (str): A dataset\_key

**Returns:**

The dataset\_name which is the base name stripped of extensions

`pipeline.model_pipeline.build_tarball_name(dataset_name, model_uuid, result_dir="")`

**format for building model tarball names**

Creates the file name for a model tarball from dataset key and model\_uuid with optional result\_dir.

**Args:**

dataset\_name (str): The dataset\_name used to train this model model\_uuid (str): The model\_uuid assigned to this model result\_dir (str): Optional directory for this model

**Returns:**

The path or filename of the tarball for this model

`pipeline.model_pipeline.calc_AD_kmean_dist(train_dset, pred_dset, k, train_dset_pair_distance=None, dist_metric='euclidean')`

calculate the probability of the prediction dataset fall in the the domain of traning set. Use Euclidean distance of the K nearest neighbours. train\_dset and pred\_dset should be in 2D numpy array format where each row is a compound.

`pipeline.model_pipeline.calc_AD_kmean_local_density(train_dset, pred_dset, k, train_dset_pair_distance=None, dist_metric='euclidean')`

Evaluate the AD of pred data by comparing the distance between the unseen object and its k nearest neighbors in the training set to the distance between these k nearest neighbors and their k nearest neighbors in the training set. Return the distance ratio. Greater than 1 means the pred data is far from the domain.

`pipeline.model_pipeline.create_prediction_pipeline(params, model_uuid, collection_name=None, featurization=None, alt_bucket='CRADA')`

Create a ModelPipeline object to be used for running blind predictions on datasets where the ground truth is not known, given a pretrained model in the model tracker database.

**Args:**

params (Namespace or dict): A parsed parameters namespace, containing parameters describing how input datasets should be processed. If a dictionary is passed, it will be parsed to fill in default values and convert it to a Namespace object.

model\_uuid (str): The UUID of a trained model.

collection\_name (str): The collection where the model is stored in the model tracker DB.

featurization (Featurization): An optional featurization object to be used for featurizing the input data. If none is provided, one will be created based on the stored model parameters.

alt\_bucket (str): Alternative bucket to search for model tarball and transformer files, if original bucket no longer exists.

**Returns:**

pipeline (ModelPipeline): A pipeline object to be used for making predictions.

```
pipeline.model_pipeline.create_prediction_pipeline_from_file(params, reload_dir,  
                                                           model_path=None,  
                                                           model_type='best_model',  
                                                           featurization=None, verbose=True)
```

Create a ModelPipeline object to be used for running blind predictions on datasets, given a pretrained model stored in the filesystem. The model may be stored either as a gzipped tar archive or as a directory.

**Args:**

params (Namespace): A parsed parameters namespace, containing parameters describing how input datasets should be processed.

**reload\_dir (str): The path to the parent directory containing the various model subdirectories**

(e.g.: '/home/cdsdw/model/delaney-processed/delaney-processed/pxc50\_NN\_graphconv\_scaffold\_regression/').

If reload\_dir is None, then model\_path must be specified. If both are specified, then the tar archive given by model\_path will be unpacked into reload\_dir, possibly overwriting existing files in that directory.

model\_path (str): Path to a gzipped tar archive containing the saved model metadata and parameters. If specified, the tar archive is unpacked into reload\_dir if that directory is given, or to a temporary directory otherwise.

model\_type (str): Name of the subdirectory in reload\_dir or in the tar archive where the trained model state parameters should be loaded from.

featurization (Featurization): An optional featurization object to be used for featurizing the input data. If none is provided, one will be created based on the stored model parameters.

**Returns:**

pipeline (ModelPipeline): A pipeline object to be used for making predictions.

```
pipeline.model_pipeline.ensemble_predict(model_uuids, collections, dset_df, labels=None,  
                                         dset_params=None, splitters=None, mt_client=None,  
                                         aggregate='mean', contains_responses=False)
```

Load a series of pretrained models and predict responses with each model; then aggregate the predicted responses into one prediction per compound.

**Args:**

model\_uuids (iterable of str): Sequence of UUIDs of trained models.

collections (str or iterable of str): The collection(s) where the models are stored in the model tracker DB. If a single string, the same collection is assumed to contain all the models. Otherwise, collections should be of the same length as model\_uuids.

dset\_df (DataFrame): Dataset to perform predictions on. Should contain compound IDs and SMILES strings. May contain features.

labels (iterable of str): Optional suffixes for model-specific prediction column names. If not provided, the columns are labeled 'pred\_<uuid>' where <uuid> is the model UUID.

dset\_params (Namespace): Parameters used to interpret dataset, including id\_col and smiles\_col. If not provided, id\_col and smiles\_col are assumed to be same as in the pretrained model and the same for all models.

mt\_client: Ignored, for backward compatibility only.

aggregate (str): Method to be used to combine predictions.

**Returns:**

pred\_df (DataFrame): Table with predicted responses from each model, plus the ensemble prediction.

`pipeline.model_pipeline.load_from_tracker(model_uuid, collection_name=None, client=None, verbose=False, alt_bucket='CRADA')`

DEPRECATED. Use the function `create_prediction_pipeline()` directly, or use the higher-level function `predict_from_model.predict_from_tracker_model()`.

Create a ModelPipeline object using the metadata in the model tracker.

**Args:**

model\_uuid (str): The UUID of a trained model.

collection\_name (str): The collection where the model is stored in the model tracker DB.

client : Ignored, for backward compatibility only

verbose (bool): A switch for disabling informational messages

alt\_bucket (str): Alternative bucket to search for model tarball and transformer files, if original bucket no longer exists.

**Returns:**

**tuple of:**

pipeline (ModelPipeline): A pipeline object to be used for making predictions.

pparams (Namespace): Parsed parameter namespace from the requested model.

`pipeline.model_pipeline.main()`

Entry point when script is run from a shell

`pipeline.model_pipeline.regenerate_results(result_dir, params=None, metadata_dict=None, shared_featurization=None, system='twintron-blue')`

Query the model tracker for models matching the criteria in `params.model_filter`. Run predictions with each model using the dataset specified by the remaining parameters.

**Args:**

result\_dir (str): Parent of directory where result files will be written

params (Namespace): Parsed parameters

metadata\_dict (dict): Model metadata

shared\_featurization (Featurization): Object to map compounds to features, shared across models. User is responsible for ensuring that `shared_featurization` is compatible with all matching models.

system (str): System name

**Returns:**

result\_dict (dict): Results from predictions

`pipeline.model_pipeline.retrain_model(model_uuid, collection_name=None, result_dir=None, mt_client=None, verbose=True)`

Obtain model parameters from the metadata in the model tracker, given the `model_uuid`, and train a new model using exactly the same parameters (except for `result_dir`). Returns the resulting ModelPipeline object. The pipeline object can then be used as input for performance plots and other analyses that can't be done using just the metrics stored in the model tracker; or to make predictions on new data.

**Args:**

model\_uuid (str): The UUID of a trained model.

collection\_name (str): The collection where the model is stored in the model tracker DB.

result\_dir (str): The directory of model results when the model tracker is not available.

mt\_client : Ignored

verbose (bool): A switch for disabling informational messages

**Returns:**

pipeline (ModelPipeline): A pipeline object containing data from the model training.

`pipeline.model_pipeline.run_models(params, shared_featurization=None, generator=False)`

Query the model tracker for models matching the criteria in `params.model_filter`. Run predictions with each model using the dataset specified by the remaining parameters.

**Args:**

params (Namespace): Parsed parameters

shared\_featurization (Featurization): Object to map compounds to features, shared across models. User is responsible for ensuring that `shared_featurization` is compatible with all matching models.

generator (bool): True if run as a generator

## pipeline.model\_tracker module

Module to interface model pipeline to model tracker service.

**exception** `pipeline.model_tracker.DatastoreInsertionException`

Bases: Exception

**exception** `pipeline.model_tracker.MLMTClientInstantiationException`

Bases: Exception

`pipeline.model_tracker.convert_metadata(old_metadata)`

Convert model metadata from old format (with camel-case parameter group names) to new format.

**Args:**

old\_metadata (dict): Model metadata in old format

**Returns:**

new\_metadata (dict): Model metadata in new format

`pipeline.model_tracker.export_model(model_uuid, collection, model_dir, alt_bucket='CRADA')`

Export the metadata (parameters) and other files needed to recreate a model from the model tracker database to a gzipped tar archive.

**Args:**

model\_uuid (str): Model unique identifier

collection (str): Name of the collection holding the model in the database.

model\_dir (str): Path to directory where the model metadata and parameter files will be written. The directory will be created if it doesn't already exist. Subsequently, the directory contents will be packed into a gzipped tar archive named `model_dir.tar.gz`.

alt\_bucket (str): Alternate datastore bucket to search for model tarball and transformer objects.

**Returns:**

none

`pipeline.model_tracker.extract_datastore_model_tarball(model_uuid, model_bucket, output_dir, model_dir)`

Load a model tarball saved in the datastore and check the format. If it is a new style tarball (containing the model metadata and transformers along with the model state), unpack it into `output_dir`. Otherwise it contains the model state only; unpack it into `model_dir`.

**Args:**

`model_uuid` (str): UUID of model to be retrieved  
`model_bucket` (str): Datastore bucket containing model tarball file  
`output_dir` (str): Output directory to unpack tarball into if it's in the new format  
`model_dir` (str): Output directory to unpack tarball into if it's in the old format

**Returns:**

`extract_dir` (str): The directory (`output_dir` or `model_dir`) the tarball was extracted into.

`pipeline.model_tracker.get_full_metadata(filter_dict, collection_name=None)`

Retrieve relevant full metadata (including training run metrics) of models matching given criteria.

**Args:**

`filter_dict` (dict): dictionary to filter on  
`collection_name` (str): Name of collection to search

**Returns:**

A list of matching full model metadata (including training run metrics) dictionaries. Raises `MongoQueryException` if the query fails.

`pipeline.model_tracker.get_full_metadata_by_uuid(model_uuid, collection_name=None)`

Retrieve model parameter metadata for the given `model_uuid` and collection. The returned metadata dictionary will include training run performance metrics and training dataset metadata.

**Args:**

`model_uuid` (str): model unique identifier  
`collection_name`(str): collection to search (optional, searches all collections if not specified)

**Returns:**

Matching metadata dictionary. Raises `MongoQueryException` if the query fails.

`pipeline.model_tracker.get_metadata_by_uuid(model_uuid, collection_name=None)`

Retrieve model parameter metadata by `model_uuid`. The resulting metadata dictionary can be passed to `parameter_parser.wrapper()`; it does not contain performance metrics or training dataset metadata.

**Args:**

`model_uuid` (str): model unique identifier  
`collection_name`(str): collection to search (optional, searches all collections if not specified)

**Returns:**

Matching metadata dictionary. Raises `MongoQueryException` if the query fails.

`pipeline.model_tracker.get_model_collection_by_uuid(model_uuid, mlmt_client=None)`

Retrieve model collection given a uuid.

**Args:**

`model_uuid` (str): model uuid  
`mlmt_client`: Ignored

**Returns:**

Matching collection name

**Raises:**

ValueError if there is no collection containing a model with the given uuid.

`pipeline.model_tracker.get_model_training_data_by_uuid(uuid)`

Retrieve data used to train, validate, and test a model given the uuid

**Args:**

uuid (str): model uuid

**Returns:**

a tuple of dataframes containing training data, validation data, and test data including the compound ID, RDKit SMILES, and response value

`pipeline.model_tracker.save_model(pipeline, collection_name='model_tracker', log=True)`

Save the model.

Save the model files to the datastore and save the model metadata dict to the Mongo database.

**Args:**

pipeline (ModelPipeline object): the pipeline to use

collection\_name (str): the name of the Mongo DB collection to use

log (bool): True if logs should be printed, default False

use\_personal\_client (bool): True if personal client should be used (i.e. for testing), default False

**Returns:**

None if insertion was successful, raises DatastoreInsertionException, MLMTClientInstantiationException or MongoInsertionException otherwise

`pipeline.model_tracker.save_model_tarball(output_dir, model_tarball_path)`

Save the model parameters, metadata and transformers as a portable gzipped tar archive.

**Args:**

output\_dir (str): Output directory from model training

model\_tarball\_path (str): Path of tarball file to be created

**Returns:**

None

## pipeline.parameter\_parser module

**class** `pipeline.parameter_parser.AutoArgumentAdder(func, prefix)`

Bases: object

Finds, manages, and adds all parameters of an object to a argparse parser

AutoArgumentAdder recursively finds all keyword arguments of a given object. A prefix is added to each keyword argument to prevent collisions and help distinguish automatically added arguments from normal arguments.

**Attributes:**

func (object): The original object e.g. dcm.AttentiveFPModel  
funcs (List[object]): A list of parents. e.g. KerasModel  
prefix (str): A prefix for arguments. e.g. 'AttentiveFPModel'  
types (dict): A mapping between parameter names and types. Prefixes

are not used in the keys.



**used\_by (dict):** A mapping between parameter names (no prefix) and the object or objects that use that parameter.

args (set): A set of all argument names

#### **add\_to\_parser**(*parser*)

Adds expected parameters to an `argparse.ArgumentParser`. Checks to see if the argument has synonyms e.g. `mode` and `prediction_type` and sets `dest` accordingly. All parameters have `default=None`, this is checked later in `self.extract_params`. None parameters are not passed on so we can use default parameters set by `DeepChem`.

##### **Args:**

`parser (argparse.ArgumentParser):` An argument parser

##### **Returns:**

None

#### **all\_prefixed\_names**()

Returns a list of all argument names with prefixes added

##### **Args:**

None

##### **Returns:**

List[str]: A list of all arguments with prefix added

#### **extract\_params**(*params, strip\_prefix=False*)

Extracts non-None parameters from the given Namespace.

##### **Args:**

`params (Namespace):` Parameters. `strip_prefix (bool):` Strips off the prefix of the parameter. e.g. `AttentiveFP_mode` becomes `mode`

##### **Returns:**

**dict:** Dictionary containing a subset of parameters that are expected by this function.

#### **get\_list\_args**()

Returns a list of arguments that accept a List

##### **Args:**

None

##### **Returns:**

List[str]: A list of prefixed argument names that will accept a List

#### **get\_list\_float\_args**()

Returns a list of arguments that accept a List[float]

##### **Args:**

None

##### **Returns:**

List[str]: A list of prefixed argument names that will accept a List[float]

#### **get\_list\_int\_args**()

Returns a list of arguments that accept a List[int]

##### **Args:**

None

**Returns:**

List[str]: A list of prefixed argument names that will accept a List[int]

`pipeline.parameter_parser.all_auto_arguments()`

Returns a set of all arguments that get automatically added

**Args:**

None

**Returns:**

set: A set of all arguments that were automatically added.

`pipeline.parameter_parser.all_auto_float_lists()`

Returns a set of all arguments that are automatically added and accept a list of float.

**Args:**

None

**Returns:**

A set of automatically added arguments that accept a list of floats

`pipeline.parameter_parser.all_auto_int_lists()`

Returns a set of all arguments that are automatically added and accept a list of ints.

**Args:**

None

**Returns:**

**set: A set of automatically added arguments that could accept a list of ints.**

`pipeline.parameter_parser.all_auto_lists()`

Returns a set of all arguments that get automatically added and are lists

**Args:**

None

**Returns:**

set: A set of automatically added arguments that accept a list.

`pipeline.parameter_parser.dict_to_list(inp_dictionary, replace_spaces=False)`

Method to convert dictionary to a modified list of strings for input to argparse. Adds a ‘-’ in front of keys in the dictionary.

**Args:**

`inp_dictionary (dict)`: Flat dictionary of parameters

`replae_spaces (bool)`: A flag for replace spaces with `replace_spaces_str` for handling spaces in command line.

**Returns:**

(list): a list of default parameters + user specified parameters

None if `inp_dictionary` is None

`pipeline.parameter_parser.extract_featurizer_params(params, strip_prefix=True)`

Extracts parameters meant for a specific featurizer. Use only for arguments automatically added by an `AutoArgumentAdder`

**Args:**

`params (Namespace)`: Parameter Namespace `strip_prefix (bool)`: Automatically added parameters come with a prefix.

When True, the prefix is removed. e.g. MolGraphConvFeaturizer\_use\_edges becomes use\_edges

**Returns:**

**dict:** A subset of parameters from params that should be passed on to the featurizer

`pipeline.parameter_parser.extract_model_params(params, strip_prefix=True)`

Extracts parameters meant for a specific model. Use only for arguments automatically added by an AutoArgumentAdder

**Args:**

params (Namespace): Parameter Namespace strip\_prefix (bool): Automatically added parameters come with a prefix.

When True, the prefix is removed. e.g. AttentiveFP\_mode becomes mode

**Returns:**

**dict:** A subset of parameters from params that should be passed on to the model

`pipeline.parameter_parser.flatten_dict(inp_dict, newdict={})`

Method to flatten a hierarchical dictionary. Used in parse\_config\_file(). Throws error if there are duplicated keys in the dictionary. WARNING: immediately throws error upon first detection of duplications.

**Args:**

inp\_dict(dict): hierarchical dictionary

newdict(empty dict): empty dictionary, name of output flattened dictionary

**Returns:**

newdict(dict): Flattened dictionary.

`pipeline.parameter_parser.get_parser()`

Method that performs the actual parsing of pre-processed parameters. Modify this method to add/change/remove parameters

Args: None

**Returns:**

parser (argparse.Namespace): an object containing default parameters + user specific parameters

`pipeline.parameter_parser.is_list(p, type_annotation)`

Given paramter name and annotation, returns true if it accepts a list

Returns False on generic list will only return true for 'typing.List' or <class 'list'>

Performs recursive earch in case of typing.Union

**Args:**

p (str): A parameter name.

**type\_annotation (object):** This is a type annotation returned by the inspect module

**Returns:**

boolean: If this annotation will accept a List

`pipeline.parameter_parser.is_list_float(p, type_annotation)`

Given paramter name and annotation, returns true if it accepts a float list

Returns False on generic list will only return true for 'typing.List[float]'

Performs recursive earch in case of typing.Union

**Args:**

p (str): A parameter name.

**type\_annotation (object):** This is a type annotation returned by the inspect module

**Returns:**

boolean: If this annotation will accept a List[float]

`pipeline.parameter_parser.is_list_int(p, type_annotation)`

Given parameter name and annotation, returns true if this accepts an integer list

Returns False on generic list will only return true for 'typing.List[int]'

Performs recursive earch in case of typing.Union

**Args:**

p (str): A parameter name.

**type\_annotation (object):** This is a type annotation returned by the inspect module

**Returns:**

boolean: If this annotation will accept a List[int]

`pipeline.parameter_parser.is_primitive_type(t)`

Returns true if t is of type int, str, or float

**Args:**

t (type): A type

**Returns:**

bool. True if type is int, str, or float

`pipeline.parameter_parser.list_defaults(hyperparam=False)`

Creates temporary required variables, to generate a Namespace.argparse object of defaults.

**Returns:**

argparse.Namespace: a Namespace.argparse object containing default parameters + user specified parameters

`pipeline.parameter_parser.main(argument)`

Entry point when script is run from a shell

`pipeline.parameter_parser.make_dataset_key_absolute(parsed_args)`

Converts dataset\_key to an absolute path

**Args:**

params (argparse.Namespace): Raw parsed arguments.

`pipeline.parameter_parser.parse_command_line(args=None)`

Parses a command line argument or a specifically formatted list of strings into a Namespace.argparse object.

**String input is in the following format:**

args = ['-arg1', 'val1', '-arg2', 'val2', '-arg3', 'val3']

**Args:**

args(None or list): If args is none, parse\_command\_line parses sys.argv if it . If args is a list, the list is parsed

**Returns:**

parsed\_args (argparse.Namespace): an object containing default parameters + user specific parameters

`pipeline.parameter_parser.parse_config_file(config_file_path)`

Method to convert a .json configuration file to a Namespace object. Does the following conversions: .json -> hierarchical dict -> flat dict -> dict\_to\_list. WARNING: if there are two identical parameters on the same hierarchical level in the config.json, the .json will inherently silence the parameter higher up on the list without flagging a duplication. However, duplicate parameters in two different hierarchies or subdictionaries will be flagged by this parser.

**Args:**

config\_file\_path(str): PATH to configuration .json file

**Returns:**

argparse.Namespace: a Namespace.argparse object containing default parameters + user specified parameters

`pipeline.parameter_parser.parse_namespace(namespace_params=None)`

Method to convert namespace object to dictionary, then pass the value to dict\_to\_list. Will simply pass a dictionary

**Args:**

namespace\_params(dictionary or namespace.argparse object)

**Returns:**

argparse.Namespace: a Namespace.argparse object containing default parameters + user specified parameters

`pipeline.parameter_parser.postprocess_args(parsed_args)`

Postprocessing for the parsed arguments. Replaces any string in null\_options with a NoneType

Replaces any string that matches replace\_with\_space with whitespace.

Parses arguments in convert\_to\_float\_list into a list of floats, if the hyperparams option is True. E.g. parsed\_args.dropouts = "0.001,0.001 0.002,0.002 0.03,003"

-> parsed\_args.dropouts = [[0.001,0.001], [0.002,0.002], [0.03,003]]

Parses arguments in convert\_to\_int\_list into a list of ints, if the hyperparams options is True. E.g. parsed\_args.layer\_sizes = "10,100 20,200 30,300"

-> parsed\_args.layer\_sizes = [[10,100], [20,200], [30,300]]

Parameters in keep\_as\_list are kept as lists, even if there is a single item in the list.

Parameters in convert\_to\_str\_list are converted to a list of strings. E.g. parsed\_args.model\_type = "NN,RF"

-> parsed\_args.model\_type = ['NN','RF'].

If there is a single item in the list (no commas), the response is kept as a StringType, unless it is in response\_cols, which is passed as a list

Setting conditional options for descriptor\_key.

Set uncertainty to False when using XGBoost because GBoost does not support uncertainty

**Args:**

parsed\_args (argparse.Namespace): Raw parsed arguments.

**Returns:**

parsed\_args (argparse.Namespace): a argparse.Namespace object containing properly processed arguments.

**Raises:**

Exception: `layer_sizes`, `dropouts`, `weight_init_stddevs` and `bias_init_consts` arguments must be the same length

Exception: parameters within `not_a_list_outside_of_hyperparams` are not accepted as a list if `hyperparams` is `False`

`pipeline.parameter_parser.primitive_type_only(type_annotation)`

Given annotation, return only primitive types that can be read in from commandline, int, float, and str.

Default return value is str, which is default for type parameter in `add_arguments`

**Args:**

`type_annotation` (type): A type annotation.

**Returns:**

type: One of 3 choices, int, float, str

`pipeline.parameter_parser.prune_defaults(params, keep_params={})`

Removes parameters that are not in `keep_params` or in `get_defaults`

**Args:**

`params` (argparse.Namespace): Raw parsed arguments.

`keep_params` (list): List of parameters to keep

**Returns:**

`new_dict` (dict): Pruned argument dictionary

`pipeline.parameter_parser.remove_unrecognized_arguments(params, hyperparam=False)`

Removes arguments not recognized by argument parser

Can be used to clean inputs to wrapper function or `model_pipeline`. Used heavily in `hyperparam_search_wrapper`

**Args:**

`params` (Namespace or dict): params to filter

**Returns:**

dict of parameters

`pipeline.parameter_parser.strip_optional(type_annotation)`

**In the upgrade to python 3.9 type\_annotaions now use**

`typing.Optional` and we need to strip that off.

**Args:**

`type_annotation` (object): This is a type annotation returned by the inspect module

**Returns:**

`list(type_annotation)` or the `__args__` of `typing.Optional` or `typing.Union`

`pipeline.parameter_parser.to_str(params_obj)`

Converts a namespace.argparse object or a dict into a string for command line input

**Args:**

`params_obj` (argparse.Namespace or dict): an argparse namespace object or dict to be converted into a command line input.

E.g. `params_obj = argparse.Namespace(arg1 = val1, arg2 = val2, arg3 = val3)` OR `params_obj = {'arg1':val1, 'arg2':val2, 'arg3':val3}`

**Returns:**

**str\_params (str): parameters in string format**

E.g. str\_params = '-arg1 val1 -arg2 val2 -arg3 val3'

`pipeline.parameter_parser.wrapper(*any_arg)`

Wrapper to handle the ParseParams class. Calls the correct method depending on the input argument type

**Args:**

\*any\_arg: any single input of a str, dict, argparse.Namespace, or list

**Returns:**

argparse.Namespace: a Namespace object containing default parameters + user specified parameters

**Raises:**

TypeError: Input argument must be a configuration file (str), dict, argparse.Namespace, or list

## pipeline.perf\_data module

Contains class PerfData and its subclasses, which are objects for collecting and computing model performance metrics and predictions

**class pipeline.perf\_data.ClassificationPerfData(model\_dataset, subset)**

Bases: *PerfData*

Class with methods for accumulating classification model prediction data over multiple cross-validation folds and computing performance metrics after all folds have been run. Abstract class with concrete subclasses for different split strategies.

**Attributes:**

**set in \_\_init\_\_**

num\_tasks (int): Set to None, the number of tasks

num\_cmpds (int): Set to None, the number of compounds

num\_classes (int): Set to None, the number of classes

**accumulate\_preds(predicted\_vals, ids, pred\_std=None)**

Raises: NotImplementedError: The method is implemented by subclasses

**get\_pred\_values()**

Raises: NotImplementedError: The method is implemented by subclasses

**get\_prediction\_results()**

Returns a dictionary of performance metrics for a classification model. The dictionary values will contain only primitive Python types, so that it can be easily JSONified.

**Args:**

per\_task (bool): True if calculating per-task metrics, False otherwise.

**Returns:**

pred\_results (dict): dictionary of performance metrics for a classification model.

**model\_choice\_score(score\_type='roc\_auc')**

Computes a score function based on the accumulated predicted values, to be used for selecting the best training epoch and other hyperparameters.

**Args:**

**score\_type (str):** The name of the scoring metric to be used, e.g. ‘roc\_auc’, ‘precision’, ‘recall’, ‘f1’; see [https://scikit-learn.org/stable/modules/model\\_evaluation.html#scoring-parameter](https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter) and `sklearn.metrics.SCORERS.keys()` for a complete list of options. Larger values of the score function indicate better models.

**Returns:**

**score (float):** A score function value. For multitask models, this will be averaged over tasks.

**class** `pipeline.perf_data.EpochManager`(*wrapper*, *subsets*={'test': 'test', 'train': 'train', 'valid': 'valid'}, *production*=False, *\*\*kwargs*)

Bases: object

Manages lists of PerfDatas

This class manages lists of PerfDatas as well as variables related to iteratively training a model over several epochs. This class sets several variables in a given ModelWrapper for the sake of backwards compatibility

**Attributes:**

**Set in `__init__`:**

**`_subsets (dict):`** Must contain the keys ‘train’, ‘valid’, ‘test’. The values are used as subsets when calling `create_perf_data`.

**`_model_choice_score_type (str):`** Passed into `PerfData.model_choice_score`

**`_log (logger):`** This is the from `wrapper.log`

**`_should_stop (bool):`** True when training as satisfied stopping conditions. Either it has reached the max number of epochs or has exceeded `early_stopping_patience`

**`wrapper (ModelWrapper):`** The model wrapper where this object is being used.

**`_new_best_valid_score (function):`** This function takes no arguments and is called whenever a new best validation score is achieved.

**`accumulate`**(*ei*, *subset*, *dset*)

Accumulate predictions

Makes predictions, accumulate predictions and calculate the performance metric. Calls `PerfData.accumulate_preds` belonging to the epoch, subset, and given dataset.

**Args:**

*ei* (int): Epoch index

*subset* (str): Which subset, should be train, valid, or test.

*dset* (dc.data.Dataset): Calculates the performance for the given dset

**Returns:**

float: Performance metric for the given dset.

**`compute`**(*ei*, *subset*)

Computes performance metrics

This calls `PerfData.compute_perf_metrics` and saves the result in `f'{subset}_epoch_perfs'`



**Args:**

ei (int): Epoch index

subset (str): Which subset to compute\_perf\_metrics. Should be train, valid, or test

**Returns:**

None

**on\_new\_best\_valid**(*functional*)

Sets the function called when a new best validation score is achieved

Saves the function called when there's a new best validation score.

**Args:**

**functional (function):** This function takes no arguments and returns nothing. This function is called when there's a new best validation score. This can be used to tell the Model-Wrapper to save the model.

**Returns:**

None

**Side effects:**

Saves the \_new\_best\_valid\_score function.

**set\_make\_pred**(*functional*)

Sets the function used to make predictions

Sets the function used to make predictions. This must be called before invoking self.update and self.accumulate

**Args:**

**functional (function):** This function takes one argument, a `dc.data.Dataset`, and returns an array of predictions for that `dset`. This function is called when updating the training state after a given epoch.

**Returns:**

None

**Side effects:**

Saves the functional as self.\_make\_pred

**should\_stop**()

Returns True when the training loop should stop

**Returns:**

bool: True when the training loop should stop

**update**(*ei, subset, dset=None*)

Update training state

Updates the training state for a given subset and epoch index with the given dataset.

**Args:**

ei (int): Epoch index.

subset (str): Should be train, valid, test

dset (`dc.data.Dataset`): Updates using this `dset`

**Returns:**

perf (float): the performance of the given dset.

**update\_epoch**(*ei*, *train\_dset=None*, *valid\_dset=None*, *test\_dset=None*)

Update training state after an epoch

This function updates train/valid/test\_perf\_data. Call this function once per epoch. Call self.should\_stop() after calling this function to see if you should exit the training loop.

Subsets with None arguments will be ignored

**Args:**

ei (int): The epoch index

train\_dset (dc.data.Dataset): The train dataset

**valid\_dset (dc.data.Dataset): The valid dataset. Providing this argument updates best\_valid\_score and \_should\_stop**

test\_dset (dc.data.Dataset): The test dataset

**Returns:**

list: A list of performance values for the provided datasets.

**Side effects:**

This function updates self.\_should\_stop

**update\_valid**(*ei*)

Checks validation score

Checks validation performance of the given epoch index. Updates self.\_should\_stop, checks on early stopping conditions, calls self.\_new\_best\_valid\_score() when necessary.

**Args:**

ei (int): Epoch index

**Returns:**

None

**Side effects:**

Updates self.\_should\_stop when it's time to exit the training loop.

**class** pipeline.perf\_data.**EpochManagerKFold**(*wrapper*, *subsets*={'test': 'test', 'train': 'train', 'valid': 'valid'}, *production=False*, *\*\*kwargs*)

Bases: [EpochManager](#)

This class manages the training state when using KFold cross validation. This is necessary because this manager uses f'{subset}\_epoch\_perf\_stds' unlike EpochManager

**compute**(*ei*, *subset*)

Calls PerfData.compute\_perf\_metrics()

This differs from EpochManager.compute in that it saves the results into f'{subset}\_epoch\_perf\_stds'

**Args:**

ei (int): Epoch index

subset (str): Should be train, valid, test.

**Returns:**

None

**class** pipeline.perf\_data.HybridPerfData(*model\_dataset*, *subset*)

Bases: [PerfData](#)

Class with methods for accumulating regression model prediction data over multiple cross-validation folds and computing performance metrics after all folds have been run. Abstract class with concrete subclasses for different split strategies.

**Attributes:**

**set in `__init__`**

`num_tasks` (int): Set to None, the number of tasks

`num_cmpds` (int): Set to None, the number of compounds

**accumulate\_preds**(*predicted\_vals*, *ids*, *pred\_stds=None*)

Raises: `NotImplementedError`: The method is implemented by subclasses

**compute\_perf\_metrics**(*per\_task=False*)

Raises: `NotImplementedError`: The method is implemented by subclasses

**get\_pred\_values**()

Raises: `NotImplementedError`: The method is implemented by subclasses

**get\_prediction\_results**()

Returns a dictionary of performance metrics for a regression model. The dictionary values should contain only primitive Python types, so that it can be easily JSONified.

**Args:**

`per_task` (bool): True if calculating per-task metrics, False otherwise.

**Returns:**

`pred_results` (dict): dictionary of performance metrics for a regression model.

**model\_choice\_score**(*score\_type='r2'*)

Computes a score function based on the accumulated predicted values, to be used for selecting the best training epoch and other hyperparameters.

**Args:**

`score_type` (str): The name of the scoring metric to be used, e.g. 'r2', 'mae', 'rmse'

**Returns:**

**score (float): A score function value. For multitask models, this will be averaged over tasks.**

**class** pipeline.perf\_data.KFoldClassificationPerfData(*model\_dataset*, *transformers*, *subset*,  
*predict\_probs=True*, *transformed=True*)

Bases: [ClassificationPerfData](#)

Class with methods for accumulating classification model performance data over multiple cross-validation folds and computing performance metrics after all folds have been run.

**Attributes:**

**Set in `__init__`:**

`subset` (str): Label of the type of subset of dataset for tracking predictions  
`num_cmps` (int): The number of compounds in the dataset  
`num_tasks` (int): The number of tasks in the dataset  
`pred_vals` (dict): The dictionary of prediction results  
`folds` (int): Initialized at zero, flag for determining which k-fold is being assessed  
`transformers` (list of Transformer objects): from input arguments  
`real_vals` (dict): The dictionary containing the origin response column values  
`class_names` (np.array): Assumes the classes are of deepchem index type (e.g. 0,1,2,...)  
`num_classes` (int): The number of classes to predict on

**accumulate\_preds**(*predicted\_vals*, *ids*, *pred\_stds*=None)

Add training, validation or test set predictions from the current fold to the data structure where we keep track of them.

**Args:**

*predicted\_vals* (np.array): Array of the predicted values for the current dataset

*ids* (np.array): An np.array of compound ids for the current dataset

*pred\_stds* (np.array): An array of the standard deviation in the predictions, not used in this method

**Returns:**

None

**Side effects:**

Overwrites the attribute *pred\_vals*

Increments folds by 1

**compute\_perf\_metrics**(*per\_task*=False)

Computes the ROC AUC metrics for each task based on the accumulated values, averaged over training folds, along with standard deviations of the scores. If *per\_task* is False, the scores are averaged over tasks and the overall standard deviation is reported instead.

**Args:**

*per\_task* (bool): True if calculating per-task metrics, False otherwise.

**Returns:**

A tuple (*roc\_auc\_mean*, *roc\_auc\_std*):

*roc\_auc\_mean*: A numpy array of mean ROC AUC scores for each task, averaged over folds, if *per\_task* is True.

Otherwise, a float giving the ROC AUC score averaged over both folds and tasks.

***roc\_auc\_std*: A numpy array of standard deviations over folds of ROC AUC values, if *per\_task* is True.**

Otherwise, a float giving the overall standard deviation.

**get\_pred\_values**()

Returns the predicted values accumulated over training, with any transformations undone. If *self.subset* is 'train', 'train\_valid' or 'test', the function will return the means and standard deviations of the class probabilities over the training folds for each compound, for each task. Otherwise, returns a single set of predicted probabilities for each validation set compound. For all subsets, returns the compound IDs and the most probable classes for each task.

**Returns:**

*ids* (list): list of compound IDs.

*pred\_classes* (np.array): an (ncmpds, ntasks) array of predicted classes.

*class\_probs* (np.array): a (ncmpds, ntasks, nclasses) array of predicted probabilities for the classes, and

*prob\_stds* (np.array): a (ncmpds, ntasks, nclasses) array of standard errors over folds for the class probability estimates (only available for the 'train' and 'test' subsets; None otherwise).

**get\_real\_values**(*ids*=None)

Returns the real dataset response values as an (ncmpds, ntasks, nclasses) array of indicator bits (if nclasses > 2) or an (ncmpds, ntasks) array of binary classes (if nclasses == 2), with compound IDs in the same order as in the return from *get\_pred\_values*() (unless *ids* is specified).

**Args:**

ids (list of str): Optional list of compound IDs to return values for.

**Returns:**

np.array of shape (ncmpds, tasks, nclasses): of either indicator bits or a 2D array of binary classes

**get\_weights**(ids=None)

Returns the dataset response weights, as an (ncmpds, ntasks) array in the same ID order as get\_pred\_values() (unless ids is specified).

**Args:**

ids (list of str): Optional list of compound IDs to return values for.

**Returns:**

np.array (ncmpds, ntasks) of the real dataset response weights, in the same ID order as get\_pred\_values().

**class** pipeline.perf\_data.**KFoldRegressionPerfData**(model\_dataset, transformers, subset, transformed=True)

Bases: [RegressionPerfData](#)

Class with methods for accumulating regression model prediction data over multiple cross-validation folds and computing performance metrics after all folds have been run.

**Arguments:**

**Set in \_\_init\_\_:**

subset (str): Label of the type of subset of dataset for tracking predictions

num\_cmps (int): The number of compounds in the dataset

num\_tasks (int): The number of tasks in the dataset

pred\_vals (dict): The dictionary of prediction results

folds (int): Initialized at zero, flag for determining which k-fold is being assessed

transformers (list of Transformer objects): from input arguments

real\_vals (dict): The dictionary containing the origin response column values

**accumulate\_preds**(predicted\_vals, ids, pred\_stds=None)

Add training, validation or test set predictions from the current fold to the data structure where we keep track of them.

**Args:**

predicted\_vals (np.array): Array of the predicted values for the current dataset

ids (np.array): An np.array of compound ids for the current dataset

pred\_stds (np.array): An array of the standard deviation in the predictions, not used in this method

**Returns:**

None

**Raises:**

ValueError: If Predicted value dimensions don't match num\_tasks for RegressionPerfData

**Side effects:**

Overwrites the attribute pred\_vals

Increments folds by 1

**compute\_perf\_metrics**(*per\_task=False*)

Computes the R-squared metrics for each task based on the accumulated values, averaged over training folds, along with standard deviations of the scores. If *per\_task* is False, the scores are averaged over tasks and the overall standard deviation is reported instead.

**Args:**

*per\_task* (bool): True if calculating per-task metrics, False otherwise.

**Returns:**

A tuple (*r2\_mean*, *r2\_std*):

***r2\_mean*: A numpy array of mean R<sup>2</sup> scores for each task, averaged over folds, if *per\_task* is True.**

Otherwise, a float giving the R<sup>2</sup> score averaged over both folds and tasks.

***r2\_std*: A numpy array of standard deviations over folds of R<sup>2</sup> values, if *per\_task* is True.**

Otherwise, a float giving the overall standard deviation.

**get\_pred\_values**()

Returns the predicted values accumulated over training, with any transformations undone. If *self.subset* is 'train' or 'test', the function will return averages over the training folds for each compound along with standard deviations when there are predictions from multiple folds. Otherwise, returns a single predicted value for each compound.

**Returns:**

*ids* (np.array): list of compound IDs

*vals* (np.array): (ncmpds, ntasks) array of mean predicted values

*fold\_stds* (np.array): (ncmpds, ntasks) array of standard deviations over folds if applicable, and None otherwise.

**get\_real\_values**(*ids=None*)

Returns the real dataset response values, with any transformations undone, as an (ncmpds, ntasks) array in the same ID order as *get\_pred\_values*() (unless *ids* is specified).

**Args:**

*ids* (list of str): Optional list of compound IDs to return values for.

**Returns:**

np.array (ncmpds, ntasks) of the real dataset response values, with any transformations undone, in the same ID order as *get\_pred\_values*()

**get\_weights**(*ids=None*)

Returns the dataset response weights, as an (ncmpds, ntasks) array in the same ID order as *get\_pred\_values*() (unless *ids* is specified).

**Args:**

*ids* (list of str): Optional list of compound IDs to return values for.

**Returns:**

np.array (ncmpds, ntasks) of the real dataset response weights, in the same ID order as *get\_pred\_values*()

**class pipeline.perf\_data.PerfData**(*model\_dataset, subset*)

Bases: object

Class with methods for accumulating prediction data over multiple cross-validation folds and computing performance metrics after all folds have been run. Abstract class with concrete subclasses for classification and regression models.

**accumulate\_preds**(*predicted\_vals*, *ids*, *pred\_stds=None*)

Raises: NotImplementedError: The method is implemented by subclasses

**compute\_perf\_metrics**(*per\_task=False*)

Raises: NotImplementedError: The method is implemented by subclasses

**get\_pred\_values**()

Raises: NotImplementedError: The method is implemented by subclasses

**get\_prediction\_results**()

Raises: NotImplementedError: The method is implemented by subclasses

**get\_real\_values**(*ids=None*)

Raises: NotImplementedError: The method is implemented by subclasses

**get\_weights**(*ids=None*)

Returns the dataset response weights as an (ncmpds, ntasks) array

**Raises:**

NotImplementedError: The method is implemented by subclasses

**class** pipeline.perf\_data.**RegressionPerfData**(*model\_dataset*, *subset*)

Bases: [PerfData](#)

Class with methods for accumulating regression model prediction data over multiple cross-validation folds and computing performance metrics after all folds have been run. Abstract class with concrete subclasses for different split strategies.

**Attributes:**

**set in `__init__`**

num\_tasks (int): Set to None, the number of tasks

num\_cmpds (int): Set to None, the number of compounds

**accumulate\_preds**(*predicted\_vals*, *ids*, *pred\_stds=None*)

Raises: NotImplementedError: The method is implemented by subclasses

**compute\_perf\_metrics**(*per\_task=False*)

Raises: NotImplementedError: The method is implemented by subclasses

**get\_pred\_values**()

Raises: NotImplementedError: The method is implemented by subclasses

**get\_prediction\_results**()

Returns a dictionary of performance metrics for a regression model. The dictionary values should contain only primitive Python types, so that it can be easily JSONified.

**Args:**

per\_task (bool): True if calculating per-task metrics, False otherwise.

**Returns:**

pred\_results (dict): dictionary of performance metrics for a regression model.

**model\_choice\_score**(*score\_type='r2'*)

Computes a score function based on the accumulated predicted values, to be used for selecting the best training epoch and other hyperparameters.

**Args:**

**score\_type (str):** The name of the scoring metric to be used, e.g. 'r2', 'neg\_mean\_squared\_error', 'neg\_mean\_absolute\_error', etc.; see [https://scikit-learn.org/stable/modules/model\\_evaluation.html#scoring-parameter](https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter) and `sklearn.metrics.SCORERS.keys()` for a complete list of options. Larger values of the score function indicate better models.

**Returns:**

**score (float):** A score function value. For multitask models, this will be averaged over tasks.

**class** `pipeline.perf_data.SimpleClassificationPerfData`(*model\_dataset, transformers, subset, predict\_probs=True, transformed=True*)

Bases: `ClassificationPerfData`

Class with methods for collecting classification model prediction and performance data from single-fold training and prediction runs.

**Attributes:**

**Set in `__init__`:**

`subset` (str): Label of the type of subset of dataset for tracking predictions

`num_cmps` (int): The number of compounds in the dataset

`num_tasks` (int): The number of tasks in the dataset

`pred_vals` (dict): The dictionary of prediction results

`folds` (int): Initialized at zero, flag for determining which k-fold is being assessed

`transformers` (list of Transformer objects): from input arguments

`real_vals` (dict): The dictionary containing the origin response column values

`class_names` (np.array): Assumes the classes are of deepchem index type (e.g. 0,1,2,...)

`num_classes` (int): The number of classes to predict on

**accumulate\_preds**(*predicted\_vals, ids, pred\_stds=None*)

Add training, validation or test set predictions from the current dataset to the data structure where we keep track of them.

**Arguments:**

`predicted_vals` (np.array): Array of predicted values (class probabilities)

`ids` (list): List of the compound ids of the dataset

`pred_stds` (np.array): Optional np.array of the prediction standard deviations

**Side effects:**

Updates `self.pred_vals` and `self.perf_metrics`

**compute\_perf\_metrics**(*per\_task=False*)

Returns the ROC\_AUC metrics for each task based on the accumulated predictions. If `per_task` is False, returns the average ROC AUC over tasks.

**Args:**

`per_task` (bool): Whether to return individual ROC AUC scores for each task

**Returns:**

**A tuple (roc\_auc, std):**

**roc\_auc:** A numpy array of ROC AUC scores, if `per_task` is True. Otherwise, a float giving the mean ROC AUC score over tasks.



std: Placeholder for an array of standard deviations. Always None for this class.

#### **get\_pred\_values()**

Returns the predicted values accumulated over training, with any transformations undone. If self.subset is 'train', the function will average class probabilities over the k-1 folds in which each compound was part of the training set, and return the most probable class. Otherwise, there should be a single set of predicted probabilities for each validation or test set compound. Returns a tuple (ids, pred\_classes, class\_probs, prob\_stds), where ids is the list of compound IDs, pred\_classes is an (ncmpds, ntasks) array of predicted classes, class\_probs is a (ncmpds, ntasks, nclasses) array of predicted probabilities for the classes, and prob\_stds is a (ncmpds, ntasks, nclasses) array of standard errors for the class probability estimates.

#### **Returns:**

**Tuple (ids, pred\_classes, class\_probs, prob\_stds)**

ids (list): Contains the dataset compound ids

pred\_classes (np.array): Contains (ncmpds, ntasks) array of prediction classes

class\_probs (np.array): Contains (ncmpds, ntasks, nclasses) array of predict class probabilities

prob\_stds (np.array): Contains (ncmpds, ntasks, nclasses) array of standard errors for the class probability estimates

#### **get\_real\_values(ids=None)**

Returns the real dataset response values as an (ncmpds, ntasks, nclasses) array of indicator bits. If nclasses == 2, the returned array has dimension (ncmpds, ntasks).

#### **Args:**

ids: Ignored for this class

#### **Returns:**

np.array of the response values of the real dataset as indicator bits

#### **get\_weights(ids=None)**

Returns the dataset response weights

#### **Args:**

ids: Ignored for this class

#### **Returns:**

np.array: Containing the dataset response weights

**class** pipeline.perf\_data.**SimpleHybridPerfData**(model\_dataset, transformers, subset, is\_ki, ki\_convert\_ratio=None, transformed=True)

Bases: [HybridPerfData](#)

Class with methods for accumulating hybrid model prediction data from training, validation or test sets and computing performance metrics.

#### **Attributes:**

##### **Set in \_\_init\_\_:**

subset (str): Label of the type of subset of dataset for tracking predictions

num\_cmps (int): The number of compounds in the dataset

num\_tasks (int): The number of tasks in the dataset

pred\_vals (dict): The dictionary of prediction results

folds (int): Initialized at zero, flag for determining which k-fold is being assessed

transformers (list of Transformer objects): from input arguments

`real_vals` (dict): The dictionary containing the origin response column values

**`accumulate_preds`**(*predicted\_vals*, *ids*, *pred\_stds=None*)

Add training, validation or test set predictions to the data structure where we keep track of them.

**Args:**

`predicted_vals` (np.array): Array of predicted values

`ids` (list): List of the compound ids of the dataset

`pred_stds` (np.array): Optional np.array of the prediction standard deviations

**Side effects:**

Reshapes the predicted values and the standard deviations (if they are given)

**`compute_perf_metrics`**(*per\_task=False*)

Returns the R-squared metrics for each task or averaged over tasks based on the accumulated values

**Args:**

`per_task` (bool): True if calculating per-task metrics, False otherwise.

**Returns:**

**A tuple (`r2_score`, `std`):**

`r2_score` (np.array): An array of scores for each task, if `per_task` is True. Otherwise, it is a float containing the average  $R^2$  score over tasks.

`std`: Always None for this class.

**`get_pred_values`**()

Returns the predicted values accumulated over training, with any transformations undone. Returns a tuple (`ids`, `values`, `stds`), where `ids` is the list of compound IDs, `values` is a (ncmpds, ntasks) array of predictions, and `stds` is always None for this class.

**Returns:**

**Tuple (`ids`, `vals`, `stds`)**

`ids` (list): Contains the dataset compound ids

`vals` (np.array): Contains (ncmpds, ntasks) array of prediction

`stds` (np.array): Contains (ncmpds, ntasks) array of prediction standard deviations

**`get_real_values`**(*ids=None*)

Returns the real dataset response values, with any transformations undone, as an (ncmpds, ntasks) array with compounds in the same ID order as in the return from `get_pred_values`().

**Args:**

`ids`: Ignored for this class

**Returns:**

np.array: Containing the real dataset response values with transformations undone.

**`get_weights`**(*ids=None*)

Returns the dataset response weights as an (ncmpds, ntasks) array

**Args:**

`ids`: Ignored for this class

**Returns:**

np.array: Containing the dataset response weights

```
class pipeline.perf_data.SimpleRegressionPerfData(model_dataset, transformers, subset,  
                                                transformed=True)
```

Bases: [\*RegressionPerfData\*](#)

Class with methods for accumulating regression model prediction data from training, validation or test sets and computing performance metrics.

**Attributes:**

**Set in `__init__`:**

`subset` (str): Label of the type of subset of dataset for tracking predictions

`num_cmps` (int): The number of compounds in the dataset

`num_tasks` (int): The number of tasks in the dataset

`pred_vals` (dict): The dictionary of prediction results

`folds` (int): Initialized at zero, flag for determining which k-fold is being assessed

`transformers` (list of Transformer objects): from input arguments

`real_vals` (dict): The dictionary containing the origin response column values

**`accumulate_preds`**(*predicted\_vals, ids, pred\_stds=None*)

Add training, validation or test set predictions to the data structure where we keep track of them.

**Args:**

`predicted_vals` (np.array): Array of predicted values

`ids` (list): List of the compound ids of the dataset

`pred_stds` (np.array): Optional np.array of the prediction standard deviations

**Side effects:**

Reshapes the predicted values and the standard deviations (if they are given)

**`compute_perf_metrics`**(*per\_task=False*)

Returns the R-squared metrics for each task or averaged over tasks based on the accumulated values

**Args:**

`per_task` (bool): True if calculating per-task metrics, False otherwise.

**Returns:**

**A tuple (r2\_score, std):**

`r2_score` (np.array): An array of scores for each task, if `per_task` is True. Otherwise, it is a float containing the average  $R^2$  score over tasks.

`std`: Always None for this class.

**`get_pred_values`**()

Returns the predicted values accumulated over training, with any transformations undone. Returns a tuple (`ids`, `values`, `stds`), where `ids` is the list of compound IDs, `values` is a (ncmpds, ntasks) array of predictions, and `stds` is always None for this class.

**Returns:**

**Tuple (`ids`, `vals`, `stds`)**

`ids` (list): Contains the dataset compound ids

`vals` (np.array): Contains (ncmpds, ntasks) array of prediction

`stds` (np.array): Contains (ncmpds, ntasks) array of prediction standard deviations

**get\_real\_values**(*ids=None*)

Returns the real dataset response values, with any transformations undone, as an (ncmpds, ntasks) array with compounds in the same ID order as in the return from `get_pred_values()`.

**Args:**

*ids*: Ignored for this class

**Returns:**

np.array: Containing the real dataset response values with transformations undone.

**get\_weights**(*ids=None*)

Returns the dataset response weights as an (ncmpds, ntasks) array

**Args:**

*ids*: Ignored for this class

**Returns:**

np.array: Containing the dataset response weights

**pipeline.perf\_data.create\_perf\_data**(*prediction\_type, model\_dataset, transformers, subset, \*\*kwargs*)

Factory function that creates the right kind of PerfData object for the given subset, prediction\_type (classification or regression) and split strategy (k-fold or train/valid/test).

**Args:**

*prediction\_type* (str): classification or regression.

*model\_dataset* (ModelDataset): Object representing the full dataset.

*transformers* (list): A list of transformer objects.

*subset* (str): Label in ['train', 'valid', 'test', 'full'], indicating the type of subset of dataset for tracking predictions

**\*\*kwargs**: Additional PerfData subclass arguments

**Returns:**

PerfData object

**Raises:**

ValueError: if *split\_strategy* not in ['train\_valid\_test', 'k\_fold\_cv'] ValueError: *prediction\_type* not in ['regression', 'classification']

**pipeline.perf\_data.negative\_predictive\_value**(*y\_real, y\_pred*)

Computes negative predictive value of a binary classification model:  $NPV = TN/(TN+FN)$ .

**Args:**

*y\_real* (np.array): Array of ground truth values

*y\_pred* (np.array): Array of predicted values

**Returns:**

(float): The negative predictive value

**pipeline.perf\_data.rms\_error**(*y\_real, y\_pred*)

Calculates the root mean squared error. Score function used for model selection.

**Args:**

*y\_real* (np.array): Array of ground truth values

*y\_pred* (np.array): Array of predicted values

**Returns:**

(np.array): root mean squared error of the input

## pipeline.perf\_plots module

Plotting routines for visualizing performance of regression and classification models

`pipeline.perf_plots.plot_ROC_curve(MP, epoch_label='best', pdf_dir=None)`

Plot ROC curves for a classification model.

### Args:

*MP (ModelPipeline)*: Pipeline object for a model that was trained in the current Python session.

*epoch\_label (str)*: Label for training epoch to draw predicted values from. Currently 'best' is the only allowed value.

*pdf\_dir (str)*: If given, output the plots to a PDF file in the given directory.

### Returns:

None

`pipeline.perf_plots.plot_perf_vs_epoch(MP, pdf_dir=None)`

Plot the current NN model's standard performance metric (*r2\_score* or *roc\_auc\_score*) vs epoch number for the training, validation and test subsets. If the model was trained with k-fold CV, plot shading for the validation set out to  $\pm 1$  SD from the mean score metric values, and plot the training and test set metrics from the final model retraining rather than the cross-validation phase. Make a second plot showing the validation set model choice score used for ranking training epochs and other hyperparameters against epoch number.

### Args:

*MP (ModelPipeline)*: Pipeline object for a model that was trained in the current Python session.

*pdf\_dir (str)*: If given, output the plots to a PDF file in the given directory.

### Returns:

None

`pipeline.perf_plots.plot_prec_recall_curve(MP, epoch_label='best', pdf_dir=None)`

Plot precision-recall curves for a classification model.

### Args:

*MP (ModelPipeline)*: Pipeline object for a model that was trained in the current Python session.

*epoch\_label (str)*: Label for training epoch to draw predicted values from. Currently 'best' is the only allowed value.

*pdf\_dir (str)*: If given, output the plots to a PDF file in the given directory.

### Returns:

None

`pipeline.perf_plots.plot_pred_vs_actual(MP, epoch_label='best', threshold=None, errorBars=False, pdf_dir=None)`

Plot predicted vs actual values from a trained regression model for each split subset (train, valid, and test).

### Args:

*MP (ModelPipeline)*: Pipeline object for a model that was trained in the current Python session.

*epoch\_label (str)*: Label for training epoch to draw predicted values from. Currently 'best' is the only allowed value.

*threshold (float)*: Threshold activity value to mark on plot with dashed lines.

**errorBars (bool): If true and if uncertainty estimates are included in the model predictions, draw error bars**  
at  $\pm 1$  SD from the predicted y values.

pdf\_dir (str): If given, output the plots to a PDF file in the given directory.

**Returns:**

None

```
pipeline.perf_plots.plot_pred_vs_actual_from_df(pred_df, actual_col='avg_pIC50_actual',
                                                pred_col='avg_pIC50_pred', label='Prediction of
                                                Test Set', ax=None)
```

Plot predicted vs actual values from a trained regression model for a given dataframe.

**Args:**

pred\_df (Pandas.DataFrame): A dataframe containing predicted and actual values for each compound.

actual\_col (str): Column with actual values.

pred\_col (str): Column with predicted values.

label (str): Descriptive label for the plot.

ax (matplotlib.axes.Axes): Optional, an axes object to plot onto. If None, one is created.

**Returns:**

g (matplotlib.axes.Axes): The axes object with data.

```
pipeline.perf_plots.plot_pred_vs_actual_from_file(model_path)
```

Plot predicted vs actual values from a trained regression model from a model tarball.

**Args:**

model\_path (str): Path to an AMPL model tar.gz file.

**Returns:**

None

**Effects:**

A matplotlib figure is displayed with subplots for each response column and train/valid/test subsets.

```
pipeline.perf_plots.plot_umap_feature_projections(MP, ndim=2, num_neighbors=20, min_dist=0.1,
                                                  fit_to_train=True, dist_metric='euclidean',
                                                  dist_metric_kwds={}, target_weight=0,
                                                  random_seed=17, pdf_dir=None)
```

Projects features of a model's input dataset using UMAP to 2D or 3D coordinates and draws a scatterplot. Shape-codes plot markers to indicate whether the associated compound was in the training, validation or test set. For classification models, also uses the marker shape to indicate whether the compound's class was correctly predicted, and uses color to indicate whether the true class was active or inactive. For regression models, uses the marker color to indicate the discrepancy between the predicted and actual values.

**Args:**

MP (*ModelPipeline*): Pipeline object for a model that was trained in the current Python session.

ndim (int): Number of dimensions (2 or 3) to project features into.

**num\_neighbors (int): Number of nearest neighbors used by UMAP for manifold approximation.**

Larger values give a more global view of the data, while smaller values preserve more local detail.

min\_dist (float): Parameter used by UMAP to set minimum distance between projected points.

**fit\_to\_train (bool): If true (the default), fit the UMAP projection to the training set feature vectors only.**

Otherwise, fit it to the entire dataset.

**dist\_metric (str):** Name of metric to use for initial distance matrix computation. Check UMAP documentation

for supported values. The metric should be appropriate for the type of features used in the model (fingerprints or descriptors); note that *jaccard* is equivalent to Tanimoto distance for ECFP fingerprints.

**dist\_metric\_kwds (dict):** Additional key-value pairs used to parameterize `dist_metric`; see the UMAP documentation.

In particular, `dist_metric_kwds['p']` specifies the power/exponent for the Minkowski metric.

**target\_weight (float):** Weighting factor determining balance between activities and feature values in determining topology

of projected points. A weight of zero prioritizes the feature vectors; weight = 1 prioritizes the activity values, so that compounds with the same activity tend to be clustered together.

`random_seed (int):` Seed for random number generator.

`pdf_dir (str):` If given, output the plot to a PDF file in the given directory.

#### Returns:

None

```
pipeline.perf_plots.plot_umap_train_set_neighbors(MP, num_neighbors=20, min_dist=0.1,
                                                  dist_metric='euclidean', dist_metric_kwds={},
                                                  random_seed=17, pdf_dir=None)
```

Project features of whole dataset to 2 dimensions, without regard to response values. Plot training & validation set or training and test set compounds, color- and symbol-coded according to actual classification and split set. The plot does not take predicted values into account at all. Does not work with regression data.

#### Args:

`MP (ModelPipeline):` Pipeline object for a model that was trained in the current Python session.

**num\_neighbors (int):** Number of nearest neighbors used by UMAP for manifold approximation.

Larger values give a more global view of the data, while smaller values preserve more local detail.

`min_dist (float):` Parameter used by UMAP to set minimum distance between projected points.

**dist\_metric (str):** Name of metric to use for initial distance matrix computation. Check UMAP documentation

for supported values. The metric should be appropriate for the type of features used in the model (fingerprints or descriptors); note that *jaccard* is equivalent to Tanimoto distance for ECFP fingerprints.

**dist\_metric\_kwds (dict):** Additional key-value pairs used to parameterize `dist_metric`; see the UMAP documentation.

In particular, `dist_metric_kwds['p']` specifies the power/exponent for the Minkowski metric.

`random_seed (int):` Seed for random number generator.

`pdf_dir (str):` If given, output the plot to a PDF file in the given directory.

### pipeline.predict\_from\_model module

Functions to run predictions from a pre-trained model against user-provided data.

```
pipeline.predict_from_model.predict_from_model_file(model_path, input_df, id_col='compound_id',
                                                    smiles_col='rdkit_smiles', response_col=None,
                                                    conc_col=None, is_featurized=False,
                                                    dont_standardize=False, AD_method=None,
                                                    k=5, dist_metric='euclidean',
                                                    external_training_data=None,
                                                    max_train_records_for_AD=1000)
```

Loads a pretrained model from a model tarball file and runs predictions on compounds in an input data frame.

**Args:**

`model_path` (str): File path of the model tarball file.

`input_df` (DataFrame): Input data to run predictions on; must at minimum contain SMILES strings.

`id_col` (str): Name of the column containing compound IDs. If none is provided, sequential IDs will be generated.

`smiles_col` (str): Name of the column containing SMILES strings; required.

`response_col` (str): Name of an optional column containing actual response values; if it is provided, the actual values will be included in the returned data frame to make it easier for you to assess performance.

`conc_col` (str): Name of an optional column containing the concentration for single concentration activity (% binding) prediction in hybrid models.

`is_featurized` (bool): True if `input_df` contains precomputed feature columns. If so, `input_df` must contain *all* of the feature columns defined by the featurizer that was used when the model was trained. Default is False which tells AMPL to compute the necessary descriptors.

`dont_standardize` (bool): By default, SMILES strings are salt-stripped and standardized using RDKit; if you have already done this, or don't want them to be standardized, set `dont_standardize` to True.

`AD_method` (str or None): Method to use to compute applicability domain (AD) index; may be 'z\_score', 'local\_density' or None (the default). With the default value, AD indices will not be calculated.

`k` (int): Number of nearest neighbors of each training data point used to evaluate the AD index.

`dist_metric` (str): Metric used to compute distances between feature vectors for AD index calculation. Valid values are 'cityblock', 'cosine', 'euclidean', 'jaccard', and 'manhattan'. If binary features such as fingerprints are used in model, 'jaccard' (equivalent to Tanimoto distance) may be a better choice than the other metrics which operate on continuous features.

`external_training_data` (str): Path to a copy of the model training dataset. Used for AD index computation in the case where the model was trained on a different computing system, or more generally when the training data is not accessible at the path saved in the model metadata.

`max_train_records_for_AD` (int): Maximum number of training data rows to use for AD calculation. Note that the AD calculation time scales as the square of the number of training records used. If the training dataset is larger than `max_train_records_for_AD`, a random sample of rows with this size is used instead for the AD calculations.

**Returns:**

A data frame with compound IDs, SMILES strings, predicted response values, and (optionally) uncertainties and/or AD indices. In addition, actual response values will be included if `response_col` is specified. Standard prediction error estimates will be included if the model was trained with `uncertainty=True`. Note that the predicted and actual response columns and standard errors will be labeled according to the `response_col` setting in the original training data, not the `response_col` passed to this function. For example, if the original model `response_col` was 'pIC50', the returned data frame will contain columns 'pIC50\_actual', 'pIC50\_pred' and 'pIC50\_std'.

For proper AD index calculation, the original data column names must be the same for the new data.

```
pipeline.predict_from_model.predict_from_tracker_model(model_uuid, collection, input_df,
                                                         id_col='compound_id',
                                                         smiles_col='rdkit_smiles',
                                                         response_col=None, conc_col=None,
                                                         is_featurized=False,
                                                         dont_standardize=False, AD_method=None,
                                                         k=5, dist_metric='euclidean',
                                                         max_train_records_for_AD=1000)
```



Loads a pretrained model from the model tracker database and runs predictions on compounds in an input data frame.

**Args:**

`model_uuid` (str): The unique identifier of the model

`collection` (str): Name of the collection in the model tracker DB containing the model.

`input_df` (DataFrame): Input data to run predictions on; must at minimum contain SMILES strings.

`id_col` (str): Name of the column containing compound IDs. If none is provided, sequential IDs will be generated.

`smiles_col` (str): Name of the column containing SMILES strings; required.

`response_col` (str): Name of an optional column containing actual response values; if it is provided, the actual values will be included in the returned data frame to make it easier for you to assess performance.

`conc_col` (str): Name of an optional column containing the concentration for single concentration activity (% binding) prediction in hybrid models.

`is_featurized` (bool): True if `input_df` contains precomputed feature columns. If so, `input_df` must contain *all* of the feature columns defined by the featurizer that was used when the model was trained. Default is False which tells AMPL to compute the necessary descriptors.

`dont_standardize` (bool): By default, SMILES strings are salt-stripped and standardized using RDKit; if you have already done this, or don't want them to be standardized, set `dont_standardize` to True.

`AD_method` (str or None): Method to use to compute applicability domain (AD) index; may be 'z\_score', 'local\_density' or None (the default). With the default value, AD indices will not be calculated.

`k` (int): Number of nearest neighbors of each training data point used to evaluate the AD index.

`dist_metric` (str): Metric used to compute distances between feature vectors for AD index calculation. Valid values are 'cityblock', 'cosine', 'euclidean', 'jaccard', and 'manhattan'. If binary features such as fingerprints are used in model, 'jaccard' (equivalent to Tanimoto distance) may be a better choice than the other metrics which operate on continuous features.

`max_train_records_for_AD` (int): Maximum number of training data rows to use for AD calculation. Note that the AD calculation time scales as the square of the number of training records used. If the training dataset is larger than `max_train_records_for_AD`, a random sample of rows with this size is used instead for the AD calculations.

**Returns:**

A data frame with compound IDs, SMILES strings, predicted response values, and (optionally) uncertainties and/or AD indices. In addition, actual response values will be included if `response_col` is specified. Standard prediction error estimates will be included if the model was trained with `uncertainty=True`. Note that the predicted and actual response columns and standard errors will be labeled according to the `response_col` setting in the original training data, not the `response_col` passed to this function. For example, if the original model `response_col` was 'pIC50', the returned data frame will contain columns 'pIC50\_actual', 'pIC50\_pred' and 'pIC50\_std'.

For proper AD index calculation, the original data column names must be the same for the new data.

## Module contents

### 4.1.2 utils package

#### Submodules

##### utils.compare\_splits\_plots module

**class** `utils.compare_splits_plots.SplitStats(total_df, split_df, smiles_col, id_col, response_cols)`

Bases: `object`

This object manages a dataset and a given split dataframe.

**dist\_hist\_plot**(*dists, title, dist\_path=""*)

Creates a histogram of pairwise Tanimoto distances between training and test sets

**Args:**

**dist\_path (str):** Optional Where to save the plot. The string ‘\_dist\_hist’ will be appended to this input

**dist\_hist\_train\_v\_test\_plot**(*ax=None*)

Plots Tanimoto differences between training and valid subsets

**Returns:**

g (Seaborn FacetGrid): FacetGrid object from seaborn

**dist\_hist\_train\_v\_valid\_plot**(*ax=None*)

Plots Tanimoto differences between training and valid subsets

**Returns:**

g (Seaborn FacetGrid): FacetGrid object from seaborn

**make\_all\_plots**(*dist\_path=""*)

Makes a series of diagnostic plots

**Args:**

**dist\_path (str):** Optional Where to save the plot. The string ‘\_frac\_box’ will be appended to this input

**print\_stats**()

Prints useful statistics to stdout

**subset\_frac\_plot**(*dist\_path=""*)

Makes a box plot of the subset fractions

**Args:**

**dist\_path (str):** Optional Where to save the plot. The string ‘\_frac\_box’ will be appended to this input

**umap\_plot**(*dist\_path=""*)

Plots the first 10000 samples in Umap space using Morgan Fingerprints

**Args:**

**dist\_path (str):** Optional Where to save the plot. The string ‘\_umap\_scatter’ will be appended to this input

`utils.compare_splits_plots.parse_args()`

`utils.compare_splits_plots.save_figure(filename)`

Saves a figure to disk. Saves both png and svg formats.

**Args:**

filename (str): The name of the figure.

`utils.compare_splits_plots.split(total_df, split_df, id_col)`

Splits a dataset into training, test and validation sets using a given split.

**Args:**

total\_df (DataFrame): A pandas dataframe. split\_df (DataFrame): A split dataframe containing 'cmpd\_id' and 'subset' columns. id\_col (str): The ID column in total\_df

**Returns:**

(DataFrame, DataFrame, DataFrame): Three dataframes for train, test, and valid respectively.

## utils.curate\_data module

Utility functions used for AMPL dataset curation and creation.

`utils.curate_data.add_classification_column(thresholds, value_column, label_column, data, right_inclusive=True)`

Add a classification column to a DataFrame.

Add a classification column 'label\_column' to DataFrame 'data' based on values in 'value\_column', according to a sequence of thresholds. The number of classes is one plus the number of thresholds.

**Args:**

**thresholds (float or sequence of floats):** Thresholds to use to assign class labels. Label *i* will be assigned to values such that  $\text{thresholds}[i-1] < \text{value} \leq \text{thresholds}[i]$  (if `right_inclusive` is `True`) or  $\text{thresholds}[i-1] \leq \text{value} < \text{thresholds}[i]$  (otherwise).

value\_column (str): Name of the column from which class labels are derived.

label\_column (str): Name of the new column to be created for class labels.

data (DataFrame): DataFrame holding all data.

**right\_inclusive (bool):** Whether the thresholding intervals are closed on the right or on the left.

Set this `False` to get the same behavior as `add_binary_tertiary_classification`. The default behavior is preferred for the common case where the classification is based on a left-censoring threshold.

**Returns:**

DataFrame: DataFrame updated to include class label column.

`utils.curate_data.aggregate_assay_data(assay_df, value_col='VALUE_NUM', output_value_col=None, label_actives=True, active_thresh=None, id_col='CMPD_NUMBER', smiles_col='rdkit_smiles', relation_col='VALUE_FLAG', date_col=None, verbose=False)`

Aggregates replicated values in assay data

Map RDKit SMILES strings in assay\_df to base structures, then compute an MLE estimate of the mean value over replicate measurements for the same SMILES strings, taking censoring into account. Generate an aggregated result table with one value for each unique base SMILES string, to be used in an ML-ready dataset.

**Args:**

assay\_df (DataFrame): The input DataFrame to be processed.

value\_col (str): The column in the DataFrame containing assay values to be averaged.

output\_value\_col (str): Optional; the column name to use in the output DataFrame for the averaged data.

label\_actives (bool): If True, generate an additional column 'active' indicating whether the mean value is above a threshold specified by active\_thresh.

**active\_thresh (float): The threshold to be used for labeling compounds as active or inactive.**

If active\_thresh is None (the default), the threshold used is the minimum reported value across all records with left-censored values (i.e., those with '<' in the relation column).

id\_col (str): The input DataFrame column containing compound IDs.

smiles\_col (str): The input DataFrame column containing SMILES strings.

relation\_col (str): The input DataFrame column containing relational operators (<, >, etc.).

**date\_col (str): The input DataFrame column containing dates when the assay data was uploaded. If not None, the code will assign the earliest**

date among replicates to the aggregate data record.

**Returns:**

A DataFrame containing averaged assay values, with one value per compound.

```
utils.curate_data.average_and_remove_duplicates(column, tolerance, list_bad_duplicates, data,
                                              max_stdev=100000,
                                              compound_id='CMPD_NUMBER',
                                              rm_duplicate_only=False,
                                              smiles_col='rdkit_smiles_parent')
```

This while loop loops through until no 'bad duplicates' are left.

This function removes duplicates based on max\_stdev and tolerance. If the value in data[column] falls too far from the mean based on tolerance and max\_stdev then that entry is removed. This is repeated until all bad entries are removed

**Args:**

column (str): column with the value of interest

**tolerance (float): acceptable % difference between value and average**

ie.: if "[ (value - mean)/mean\*100 ] > tolerance" then remove data row

list\_bad\_duplicates (str): 'Yes' to list the bad duplicates

data (DataFrame): input DataFrame

max\_stdev (float): maximum standard deviation threshold

compound\_id (str): column containing compound ids

**rm\_duplicate\_only (bool): only remove bad duplicates, don't average good ones, the resulting table can be fed into aggregate assay data to further process.**

note: The mean is recalculated on each loop through to make sure it isn't skewed by the 'bad duplicate' values

smiles\_col (str): column containing base rdkit smiles strings

**Returns:**

DataFrame: Returns remaining rows after all bad duplicates have been removed.

`utils.curate_data.create_new_rows_for_extra_results(extra_result_col, value_col, data)`

Moves results from an extra column to an existing column

Returns a new DataFrame with values from 'extra\_result\_col' appended to the end of 'value\_col'. NaN values in 'extra\_result\_col' are dropped. 'Extra\_result\_col' is dropped from the resulting DataFrame

**Args:**

extra\_result\_col (str): A column in 'data'.

value\_col (str): A column in 'data'.

data (DataFrame):

**Returns:**

DataFrame

`utils.curate_data.filter_in_by_column_values(column, values, data)`

Include rows only for given values in specified column.

Filters in all rows in data if row[column] in values.

**Args:**

column (str): Name of a column in data.

**values (iterable): An iterable, Series, DataFrame, or dict of values**  
contained in data[column].

data (DataFrame): A DataFrame.

**Returns:**

DataFrame: DataFrame containing filtered rows.

`utils.curate_data.filter_in_out_by_column_values(column, values, data, in_out)`

Include rows only for given values in specified column.

Given a DataFrame, column, and an iterable, Series, DataFrame, or dict, of values, return a DataFrame with rows containing value in values or all rows that do not contain a value in values.

**Args:**

column (str): Name of a column in data.

**values (iterable): An iterable, Series, DataFrame, or dict of values**  
contained in data[column].

data (DataFrame): A DataFrame.

**in\_out (str): If set to 'in', will filter in rows that contain a value**

in values. If set to anything else, this function will filter out rows that contain a value in values.

**Returns:**

DataFrame: DataFrame containing filtered rows.

`utils.curate_data.filter_out_by_column_values(column, values, data)`

Exclude rows only for given values in specified column.

Filters out all rows in data if row[column] in values.

**Args:**

column (str): Name of a column in data.

**values (iterable): An iterable, Series, DataFrame, or dict of values**  
contained in data[column].

data (DataFrame): A DataFrame.

**Returns:**

DataFrame: DataFrame containing filtered rows.

`utils.curate_data.filter_out_comments(values, values_cs, data)`

Remove rows that contain the text listed

Removes any rows where data['COMMENTS'] contains the words in values or values\_cs. Used for removing results that indicate bad data in the comments.

**Args:**

values (str): list of values that are not case sensitive

values\_cs (str): list of values that are case sensitive

data (DataFrame): DataFrame containing a column named 'COMMENTS'

**Returns:**

DataFrame: Returns a DataFrame with the remaining rows

`utils.curate_data.freq_table(dset_df, column, min_freq=1)`

Generate a DataFrame tabulating the repeat frequencies of unique values.

Generate a DataFrame tabulating the repeat frequencies of each unique value in 'column'. Restrict it to values occurring at least min\_freq times.

**Args:**

dset\_df (DataFrame): An input DataFrame

column (str): The name of one column in DataFrame

min\_freq (int): Restrict unique count to at least min\_freq times.

**Returns:**

**DataFrame: Dataframe containing two columns: the column passed in as the 'column' argument and the column 'Count'.** The 'Count' column contains the number of occurrences for each value in the 'column' argument.

`utils.curate_data.get_rdkit_smiles_parent(data)`

Strip the salts off the rdkit SMILES strings

First, loops through data and determines the base/parent smiles string for each row. Appends the base smiles string to a new row in a list. Then adds the list as a new column, 'rdkit\_smiles\_parent', in 'data'. Basically calls base\_smiles\_from\_smiles for each smile in the column 'rdkit\_smiles'

**Args:**

data (DataFrame): A DataFrame with a column named 'rdkit\_smiles'.

**Returns:**

DataFrame with column 'rdkit\_smiles\_parent' with salts stripped

`utils.curate_data.labeled_freq_table(dset_df, columns, min_freq=1)`

Generate a frequency table in which additional columns are included.

Generate a frequency table in which additional columns are included. The first column in 'columns' is assumed to be a unique ID; there should be a many-to-1 mapping from the ID to each of the additional columns.

**Args:**

dset\_df (DataFrame): The input DataFrame.

**columns (list(str)):** A list of columns to include in the output frequency table.

The first column in 'columns' is assumed to be a unique ID; there should be a many-to-1 mapping from the ID to each of the additional columns.

`min_freq` (int): Restrict unique count to at least `min_freq` times.

**Returns:**

DataFrame: A DataFrame containing a frequency table.

**Raises:**

**Exception: If the DataFrame violates the rule: there should be a many-to-1 mapping from the ID to each of the additional columns.**

`utils.curate_data.mle_censored_mean(cmpd_df, std_est, value_col='PIC50', relation_col='relation')`

Computes maximum likelihood estimate of the true mean value for a single replicated compound.

Compute a maximum likelihood estimate of the true mean value underlying the distribution of replicate assay measurements for a single compound. The data may be a mix of censored and uncensored measurements, as indicated by the 'relation' column in the input DataFrame `cmpd_df`. `std_est` is an estimate for the standard deviation of the distribution, which is assumed to be Gaussian; we typically compute a common estimate for the whole dataset using `replicate_rmsd()`.

**Args:**

`cmpd_df` (DataFrame): DataFrame containing measurements and SMILES strings.

`std_est` (float): An estimate for the standard deviation of the distribution.

`smiles_col` (str): Name of the column that contains SMILES strings.

`value_col` (str): Name of the column that contains target values.

`relation_col` (str): The input DataFrame column containing relational operators (<, >, etc.).

**Returns:**

float: maximum likelihood estimate of the true mean for a replicated compound str: Relation, ' ' not censored, '>' right censored, '<' left censored

`utils.curate_data.remove_outlier_replicates(df, response_col='pIC50', id_col='compound_id', max_diff_from_median=1.0)`

Examine groups of replicate measurements for compounds identified by compound ID and compute median response for each group. Eliminate measurements that differ by more than a given value from the median; note that in some groups this will result in all replicates being deleted. This function should be used together with `aggregate_assay_data` instead of `average_and_remove_duplicates` to reduce data to a single value per compound.

**Args:**

`df` (DataFrame): Table of compounds and response data

`response_col` (str): Column containing response values

`id_col` (str): Column that uniquely identifies compounds, and therefore measurements to be treated as replicates.

`max_diff_from_median` (float): Maximum absolute difference from median value allowed for retained replicates.

**Returns:**

`result_df` (DataFrame): Filtered data frame with outlier replicates removed.

`utils.curate_data.replicate_rmsd(dset_df, smiles_col='base_rdkit_smiles', value_col='PIC50', relation_col='relation', default_val=1.0)`

Compute RMS deviation of all replicate uncensored measurements from means

Compute RMS deviation of all replicate uncensored measurements in `dset_df` from their means. Measurements are treated as replicates if they correspond to the same SMILES string, and are considered censored if the relation column contains > or <. The resulting value is meant to be used as an estimate of measurement error for all compounds in the dataset.

**Args:**

dset\_df (DataFrame): DataFrame containing uncensored measurements and SMILES strings.  
 smiles\_col (str): Name of the column that contains SMILES strings.  
 value\_col (str): Name of the column that contains target values.  
 relation\_col (str): The input DataFrame column containing relational operators (<, >, etc.).  
 default\_val (float): The value to return if there are no compounds with replicate measurements.

**Returns:**

float: returns root mean squared deviation of all replicate uncensored measurements

`utils.curate_data.set_group_permissions(path, system='AD', owner='GSK')`

Sets file and group permissions to standard values for a dataset containing proprietary data owned by 'owner'. Later we may add a 'public' option, or groups for data from other pharma companies.

**Args:**

path (string): File path  
 system (string): Computing environment from which group ownerships will be derived; currently, either 'LC' for LC filesystems or 'AD' for LLNL systems where owners and groups are managed by Active Directory.  
 owner (string): Who the data belongs to, either 'public' or the name of a company (e.g. 'GSK') associated with a restricted access group.

**Returns:**

None

`utils.curate_data.summarize_data(column, num_bins, title, units, filepath, data, log_column='No')`

Summarizes the in data[column]

Summarizes the data by printing mean, stdev, max, and min of the data. Creates plots of the binned values in data[column]. If log\_column != 'No' this also creates plots that compares normal and log distributions of the data.

**Args:**

column (str): Column of interest.  
 num\_bins (int): Number of bins in the histogram.  
 title (str): Title of the histogram.  
 units (str): Units for values in 'column'.  
 filepath (str): This file path gets printed to the console.  
 data (DataFrame): Input DataFrame.  
**log\_column (str): Defaults to 'No'. Any other value will generate a plot comparing normal and log distributions.**

**Returns:**

None

`utils.curate_data.xc50topxc50_for_nm(x)`

Convert XC50 values measured in nanomolars to -log10 (PX50)

**Args :**

x (float): input XC50 value measured in nanomolars

**Returns :**

float: -log10 value of x



## utils.data\_curation\_functions module

data\_curation\_functions.py

Extract Kevin's functions for curation of public datasets Modify them to match Jonathan's curation methods in notebook 01/30/2020

`utils.data_curation_functions.atom_curation(targ_lst, smiles_lst, shared_inchi_keys)`

Apply ATOM standard 'curation' step to "shared\_df": Average replicate assays, remove duplicates and drop cases with large variance between replicates. mleqonly

### Args:

`targ_lst` (list): A list of targets.

**`smiles_lst` (list): A list of DataFrames.**

These DataFrames must contain the columns `gene_names`, `standard_type`, `standard_relation`, `standard_inchi_key`, `PIC50`, and `rdkit_smiles`

`shared_inchi_keys` (list): A list of inchi keys used in this dataset.

### Returns:

**list, list:**A list of curated DataFrames and a list of the number of compounds dropped during the curation process for each target.

`utils.data_curation_functions.atom_curation_excape(targ_lst, smiles_lst, shared_inchi_keys)`

Apply ATOM standard 'curation' step: Average replicate assays, remove duplicates and drop cases with large variance between replicates. Rows with NaN values in `rdkit_smiles`, `VALUE_NUM_mean`, and `pXC50` are dropped

### Args:

`targ_lst` (list): A list of targets.

**`smiles_lst` (list): A of DataFrames.**

These DataFrames must contain the columns `gene_names`, `standard_type`, `standard_relation`, `standard_inchi_key`, `pXC50`, and `rdkit_smiles`

`shared_inchi_keys` (list): A list of inchi keys used in this dataset.

### Returns:

list:A list of curated DataFrames

`utils.data_curation_functions.compute_negative_log_responses(df, unit_col='unit', value_col='value', new_value_col='average_col', relation_col=None, new_relation_col=None, unit_conv={'nM': <function <lambda>>, 'uM': <function <lambda>>}, inplace=False)`

Given the response values in `value_col` (IC50, Ki, Kd, etc.), compute their negative base 10 logarithms (pIC50, pKi, pKd, etc.) after converting them to molar units and store them in `new_value_col`. If `relation_col` is provided, replace any '<' or '>' relations with their opposites and store the result in `new_relation_col` (if provided), or in `relation_col` if note. Rows where the original value is 0 or negative will be dropped from the dataset.

### Args:

`df` (DataFrame): A DataFrame that contains `value_col`, `unit_col` and `relation_col`.

`unit_conv` (dict): A dictionary mapping concentration units found in `unit_col` to functions that convert the corresponding concentrations to molar. The default handles micromolar and nanomolar units, represented as 'uM' and 'nM' respectively.

`unit_col (str)`: Column containing units.

`value_col (str)`: Column containing input values.

`new_value_col (str)`: Column to receive converted values.

`relation_col (str)`: Column containing relational operators for censored data.

`new_relation_col (str)`: Column to receive inverted relations applicable to the negative log transformed values.

`inplace (bool)`: If True, the input DataFrame is modified in place when possible. The default is to return a copy

**Returns:**

DataFrame: A table containing the transformed values and relations.

```
utils.data_curation_functions.convert_IC50_to_pIC50(df, unit_col='unit', value_col='value',
                                                    new_value_col='average_col',
                                                    relation_col=None, new_relation_col=None,
                                                    unit_conv={'nM': <function <lambda>>, 'uM':
                                                                <function <lambda>>}, inplace=False)
```

For backward compatibilty only: equivalent to calling `compute_negative_log_responses` with the same arguments.

```
utils.data_curation_functions.down_select(df, kv_lst)
```

Filters rows given a set of values

Given a DataFrame and a list of tuples columns (k) to values (v), this function filters out all rows where `df[k] == v`.

**Args:**

`df (DataFrame)`: An input DataFrame.

`kv_list (list)`: A list of tuples of (column, value)

**Returns:**

DataFrame: Rows where all `df[k] == v`

```
utils.data_curation_functions.exclude_organometallics(df, smiles_col='rdkit_smiles')
```

Filters data frame df based on column smiles\_col to exclude organometallic compounds

```
utils.data_curation_functions.filter_dtc_data(orig_df, geneNames)
```

Extracts and post processes JAK1, 2, and 3 datasets from DTC

This is specific to the DTC database. Extract JAK1, 2 and 3 datasets from Drug Target Commons database, filtered for data usability. filter criteria:

```
gene_names == JAK1 | JAK2 | JAK3
InChi key not missing
standard_type IC50 units NM
standard_relation mappable to =, < or >
wildtype_or_mutant != 'mutated'
valid SMILES maps to valid RDKit base SMILES
standard_value not missing
pIC50 > 3
```

**Args:**

**orig\_df (DataFrame): Input DataFrame. Must contain the following columns:** `gene_names`, `standard_inchi_key`, `standard_type`, `standard_units`, `standard_value`, `compound_id`, `wildtype_or_mutant`.

`geneNames (list)`: A list of gene names to filter out of `orig_df` e.g. ['JAK1', 'JAK2'].

**Returns:**

DataFrame: The filtered rows of the `orig_df`

`utils.data_curation_functions.get_smiles_4dtt_data(nm_df, targ_lst, save_smiles_df)`

Returns SMILES strings from DTC data

`nm_df` must be a DataFrame from DTC with the following columns: `gene_names`, `standard_type`, `standard_value`, `'standard_inchi_key'`, and `standard_relation`.

This function selects all rows where `nm_df['gene_names']` is in `targ_lst`, `nm_df['standard_type']=='IC50'`, `nm_df['standard_relation']=='='`, and `'standard_value' > 0`.

Then pIC50 values are calculated and added to the 'PIC50' column, and smiles strings are merged in from `save_smiles_df`

**Args:**

`nm_df` (DataFrame): Input DataFrame.

`targ_lst` (list): A list of targets.

`save_smiles_df` (DataFrame): A DataFrame with the column 'standard\_inchi\_key'

**Returns:**

**list, list, str: A list of smiles. A list of inchi keys shared between targets.**

And a description of the targets

`utils.data_curation_functions.get_smiles_dtt_data(nm_df, targ_lst, save_smiles_df)`

Returns SMILES strings from DTC data

`nm_df` must be a DataFrame from DTC with the following columns: `gene_names`, `standard_type`, `standard_value`, `'standard_inchi_key'`, and `standard_relation`.

This function selects all rows where `nm_df['gene_names']` is in `targ_lst`, `nm_df['standard_type']=='IC50'`, `nm_df['standard_relation']=='='`, and `'standard_value' > 0`.

Then pIC50 values are calculated and added to the 'PIC50' column, and smiles strings are merged in from `save_smiles_df`

**Args:**

`nm_df` (DataFrame): Input DataFrame.

`targ_lst` (list): A list of targets.

`save_smiles_df` (DataFrame): A DataFrame with the column 'standard\_inchi\_key'

**Returns:**

list, list: A list of smiles and a list of inchi keys shared between targets.

`utils.data_curation_functions.get_smiles_escape_data(nm_df, targ_lst)`

Calculate base rdkit smiles

Divides up `nm_df` based on target and makes one DataFrame for each target.

Rows with NaN pXC50 values are dropped. Base rdkit SMILES are calculated from the SMILES column using `atomscli.ddm.utils.struct_utils.base_rdkit_smiles_from_smiles`. A new column, 'rdkit\_smiles', is added to each output DataFrame.

**Args:**

**`nm_df` (DataFrame): DataFrame for Escape database. Should contain the columns, pXC50, SMILES, and Ambit\_InchiKey**

`targ_lst` (list): A list of targets to filter out of `nm_df`

**Returns:**

**list, list:** A list of DataFrames, one for each target, and a list of all inchi keys used in the dataset.

`utils.data_curation_functions.ic50topi50(x)`

Calculates pIC50 from IC50

**Args:**

x (float): An IC50 in nanomolar (nM) units.

**Returns:**

float: The pIC50.

`utils.data_curation_functions.is_organometallic(mol)`

Returns True if the molecule is organometallic

`utils.data_curation_functions.set_data_root(dir)`

Set global variables for data directories

Creates paths for DTC and Escape given a root data directory. Global variables 'data\_root' and 'data\_dirs'. 'data\_root' is the root data directory. 'data\_dirs' is a dictionary that maps 'DTC' and 'Escape' to directories calculated from 'data\_root'

**Args:**

dir (str): root data directory containing folds 'dte' and 'escape'

**Returns:**

None

`utils.data_curation_functions.standardize_relations(dset_df, db=None, rel_col=None, output_rel_col=None, invert=False)`

Standardizes censoring operators

Standardize the censoring operators to =, < or >, and remove any rows whose operators don't map to a standard one. There is a special case for db='ChEMBL' that strips the extra ""'s around relationship symbols. Assumes relationship columns are 'Standard Relation', 'standard\_relation' and 'activity\_prefix' for ChEMBL, DTC and GoStar respectively.

This function makes the following mappings: ">" to ">", ">=" to ">", "<" to "<", "<=" to "<", and "=" to "=". All other relations are removed from the DataFrame.

**Args:**

**dset\_df (DataFrame):** Input DataFrame. Must contain either 'Standard Relation' or 'standard\_relation'

db (str): Source database. Must be either 'GoStar', 'DTC' or 'ChEMBL'. Required if rel\_col is not specified.

**rel\_col (str):** Column containing relational operators. If specified, overrides the default relation column for db.

**output\_rel\_col (str):** If specified, put the standardized operators in a new column with this name and leave the original operator column unchanged.

**invert (bool):** If true, replace the inequality operators with their inverses. This is useful when a reported value such as IC50 is converted to its negative log such as pIC50.

**Returns:**

DataFrame: Dataframe with the standardized relationship symbols

```
utils.data_curation_functions.upload_df_dtc_base_smiles_all(dset_name, title, description, tags,  
functional_area, target, target_type,  
activity, assay_category, data_df,  
dtc_mleqonly_fileID,  
data_origin='journal',  
species='human',  
force_update=False)
```

Uploads DTC base smiles data to the datastore

Uploads base SMILES string for the DTC dataset.

Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://doi.org/10.1016/j.chembiol.2017.11.009> as the doi. This also assumes that the id\_col is 'compound\_id', the response column is set to PIC50, and the SMILES are assumed to be in 'base\_rdkit\_smiles'.

**Args:**

dset\_name (str): Name of the dataset. Should not include a file extension.

title (str): title of the file in (human friendly format)

description (str): long text box to describe file (background/use notes)

tags (list): Must be a list of strings.

functional\_area (str): The functional area.

target (str): The target.

target\_type (str): The target type of the dataset.

activity (str): The activity of the dataset.

assay\_category (str): The assay category of the dataset.

data\_df (DataFrame): DataFrame to be uploaded.

dtc\_mleqonly\_fileID (str): Source file id used to generate data\_df.

data\_origin (str): The origin of the dataset e.g. journal.

species (str): The species of the dataset e.g. human, rat, dog.

force\_update (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

```
utils.data_curation_functions.upload_df_dtc_mleqonly(dset_name, title, description, tags,  
functional_area, target, target_type, activity,  
assay_category, data_df, dtc_smiles_fileID,  
data_origin='journal', species='human',  
force_update=False)
```

Uploads DTC mleqonly data to the datastore

Upload mleqonly data to the datastore from the given DataFrame. The DataFrame must contain the column 'rdkit\_smiles' and 'VALUE\_NUM\_mean'. This function is meant to upload data that has been aggregated using `atomscli.ddm.utils.curate_data.average_and_remove_duplicates`. Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://doi.org/10.1016/j.chembiol.2017.11.009> as the doi. This also assumes that the id\_col is 'compound\_id'.

**Args:**

dset\_name (str): Name of the dataset. Should not include a file extension.

title (str): title of the file in (human friendly format)

description (str): long text box to describe file (background/use notes)

tags (list): Must be a list of strings.

functional\_area (str): The functional area.

target (str): The target.

target\_type (str): The target type of the dataset.

activity (str): The activity of the dataset.

assay\_category (str): The assay category of the dataset.

data\_df (DataFrame): DataFrame to be uploaded.

dtc\_smiles\_fileID (str): Source file id used to generate data\_df.

data\_origin (str): The origin of the dataset e.g. journal.

species (str): The species of the dataset e.g. human, rat, dog.

force\_update (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

```
utils.data_curation_functions.upload_df_dtc_mleqonly_class(dset_name, title, description, tags,
                                                         functional_area, target, target_type,
                                                         activity, assay_category, data_df,
                                                         dtc_mleqonly_fileID,
                                                         data_origin='journal',
                                                         species='human', force_update=False)
```

Uploads DTC mleqonly classification data to the datastore

Upload mleqonly classification data to the datastore from the given DataFrame. The DataFrame must contain the column 'rdkit\_smiles' and 'binary\_class'. This function is meant to upload data that has been aggregated using `atomsci.ddm.utils.curate_data.average_and_remove_duplicates` and then thresholded to make a binary classification dataset. Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://doi.org/10.1016/j.chembiol.2017.11.009> as the doi. This also assumes that the `id_col` is 'compound\_id'.

**Args:**

dset\_name (str): Name of the dataset. Should not include a file extension.

title (str): title of the file in (human friendly format)

description (str): long text box to describe file (background/use notes)

tags (list): Must be a list of strings.

functional\_area (str): The functional area.

target (str): The target.

target\_type (str): The target type of the dataset.

activity (str): The activity of the dataset.

assay\_category (str): The assay category of the dataset.

data\_df (DataFrame): DataFrame to be uploaded.

dtc\_mleqonly\_fileID (str): Source file id used to generate data\_df.

data\_origin (str): The origin of the dataset e.g. journal.

species (str): The species of the dataset e.g. human, rat, dog.

force\_update (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

```
utils.data_curation_functions.upload_df_dtc_smiles(dset_name, title, description, tags,
                                                    functional_area, target, target_type, activity,
                                                    assay_category, smiles_df, orig_fileID,
                                                    data_origin='journal', species='human',
                                                    force_update=False)
```

Uploads DTC smiles data to the datastore

Upload a raw dataset to the datastore from the given DataFrame. Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://doi.org/10.1016/j.chembiol.2017.11.009> as the doi. This also assumes that the id\_col is 'compound\_id'

**Args:**

dset\_name (str): Name of the dataset. Should not include a file extension.

title (str): title of the file in (human friendly format)

description (str): long text box to describe file (background/use notes)

tags (list): Must be a list of strings.

functional\_area (str): The functional area.

target (str): The target.

target\_type (str): The target type of the dataset.

activity (str): The activity of the dataset.

assay\_category (str): The assay category of the dataset.

smiles\_df (DataFrame): DataFrame containing SMILES to be uploaded.

orig\_fileID (str): Source file id used to generate smiles\_df.

data\_origin (str): The origin of the dataset e.g. journal.

species (str): The species of the dataset e.g. human, rat, dog.

force\_update (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

```
utils.data_curation_functions.upload_df_dtc_smiles_regr_all_class(dset_name, title, description,
                                                                    tags, functional_area, target,
                                                                    target_type, activity,
                                                                    assay_category, data_df,
                                                                    dtc_smiles_regr_all_fileID,
                                                                    smiles_column,
                                                                    data_origin='journal',
                                                                    species='human',
                                                                    force_update=False)
```

Uploads DTC classification data to the datastore

Uploads binary classification data for the DTC dataset. Classnames are assumed to be 'active' and 'inactive'

Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://doi.org/10.1016/j.chembiol.2017.11.009> as the doi. This also assumes that the `id_col` is 'compound\_id', the response column is set to PIC50.

**Args:**

`dset_name` (str): Name of the dataset. Should not include a file extension.

`title` (str): title of the file in (human friendly format)

`description` (str): long text box to describe file (background/use notes)

`tags` (list): Must be a list of strings.

`functional_area` (str): The functional area.

`target` (str): The target.

`target_type` (str): The target type of the dataset.

`activity` (str): The activity of the dataset.

`assay_category` (str): The assay category of the dataset.

`data_df` (DataFrame): DataFrame to be uploaded.

`dte_smiles_regr_all_fileID`(str): Source file id used to generate `data_df`.

`smiles_column` (str): Column containing SMILES.

`data_origin` (str): The origin of the dataset e.g. journal.

`species` (str): The species of the dataset e.g. human, rat, dog.

`force_update` (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

```
utils.data_curation_functions.upload_df_excape_mleqonly(dset_name, title, description, tags,  
                                                         functional_area, target, target_type,  
                                                         activity, assay_category, data_df,  
                                                         smiles_fileID, data_origin='journal',  
                                                         species='human', force_update=False)
```

Uploads Excape mleqonly data to the datastore

Upload mleqonly to the datastore from the given DataFrame. Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://dx.doi.org/10.1186%2Fs13321-017-0203-5> as the doi. This also assumes that the `id_col` is 'Original\_Entry\_ID', `smiles_col` is 'rdkit\_smiles' and `response_col` is 'VALUE\_NUM\_mean'.

**Args:**

`dset_name` (str): Name of the dataset. Should not include a file extension.

`title` (str): title of the file in (human friendly format)

`description` (str): long text box to describe file (background/use notes)

`tags` (list): Must be a list of strings.

`functional_area` (str): The functional area.

`target` (str): The target.

`target_type` (str): The target type of the dataset.

`activity` (str): The activity of the dataset.



assay\_category (str): The assay category of the dataset.  
 data\_df (DataFrame): DataFrame containing SMILES to be uploaded.  
 smiles\_fileID (str): Source file id used to generate data\_df.  
 data\_origin (str): The origin of the dataset e.g. journal.  
 species (str): The species of the dataset e.g. human, rat, dog.  
 force\_update (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

`utils.data_curation_functions.upload_df_excape_mleqonly_class`(*dset\_name, title, description, tags, functional\_area, target, target\_type, activity, assay\_category, data\_df, mleqonly\_fileID, data\_origin='journal', species='human', force\_update=False*)

Uploads Excape mleqonly classification data to the datastore

data\_df contains a binary classification dataset with 'active' and 'inactive' classes.

Upload mleqonly classification to the datastore from the given DataFrame. Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://dx.doi.org/10.1186%2F13321-017-0203-5> as the doi. This also assumes that the id\_col is 'Original\_Entry\_ID', smiles\_col is 'rd-kit\_smiles' and response\_col is 'binary\_class'.

**Args:**

dset\_name (str): Name of the dataset. Should not include a file extension.  
 title (str): title of the file in (human friendly format)  
 description (str): long text box to describe file (background/use notes)  
 tags (list): Must be a list of strings.  
 functional\_area (str): The functional area.  
 target (str): The target.  
 target\_type (str): The target type of the dataset.  
 activity (str): The activity of the dataset.  
 assay\_category (str): The assay category of the dataset.  
 data\_df (DataFrame): DataFrame containing SMILES to be uploaded.  
 mleqonly\_fileID (str): Source file id used to generate data\_df.  
 data\_origin (str): The origin of the dataset e.g. journal.  
 species (str): The species of the dataset e.g. human, rat, dog.  
 force\_update (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

```
utils.data_curation_functions.upload_df_excape_smiles(dset_name, title, description, tags,  
                                                    functional_area, target, target_type, activity,  
                                                    assay_category, smiles_df, orig_fileID,  
                                                    data_origin='journal', species='human',  
                                                    force_update=False)
```

Uploads Excape SMILES data to the datastore

Upload SMILES to the datastore from the given DataFrame. Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://dx.doi.org/10.1186%2Fs13321-017-0203-5> as the doi. This also assumes that the id\_col is 'Original\_Entry\_ID'

**Args:**

dset\_name (str): Name of the dataset. Should not include a file extension.

title (str): title of the file in (human friendly format)

description (str): long text box to describe file (background/use notes)

tags (list): Must be a list of strings.

functional\_area (str): The functional area.

target (str): The target.

target\_type (str): The target type of the dataset.

activity (str): The activity of the dataset.

assay\_category (str): The assay category of the dataset.

smiles\_df (DataFrame): DataFrame containing SMILES to be uploaded.

orig\_fileID (str): Source file id used to generate smiles\_df.

data\_origin (str): The origin of the dataset e.g. journal.

species (str): The species of the dataset e.g. human, rat, dog.

force\_update (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

```
utils.data_curation_functions.upload_file_dtc_raw_data(dset_name, title, description, tags,  
                                                    functional_area, target, target_type, activity,  
                                                    assay_category, file_path,  
                                                    data_origin='journal', species='human',  
                                                    force_update=False)
```

Uploads raw DTC data to the datastore

Upload a raw dataset to the datastore from the given DataFrame. Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://doi.org/10.1016/j.chembiol.2017.11.009> as the doi. This also assumes that the id\_col is 'compound\_id'

**Args:**

dset\_name (str): Name of the dataset. Should not include a file extension.

title (str): title of the file in (human friendly format)

description (str): long text box to describe file (background/use notes)

tags (list): Must be a list of strings.

functional\_area (str): The functional area.

target (str): The target.

target\_type (str): The target type of the dataset.

activity (str): The activity of the dataset.

assay\_category (str): The assay category of the dataset.

file\_path (str): The filepath of the dataset.

data\_origin (str): The origin of the dataset e.g. journal.

species (str): The species of the dataset e.g. human, rat, dog.

force\_update (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

```
utils.data_curation_functions.upload_file_dtc_smiles_regr_all(dset_name, title, description, tags,  
functional_area, target, target_type,  
activity, assay_category, file_path,  
dtc_smiles_fileID, smiles_column,  
data_origin='journal',  
species='human',  
force_update=False)
```

Uploads regression DTC data to the datastore

Uploads regression dataset for DTC dataset.

Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://doi.org/10.1016/j.chembiol.2017.11.009> as the doi. This also assumes that the id\_col is 'compound\_id', the response column is set to PIC50.

**Args:**

dset\_name (str): Name of the dataset. Should not include a file extension.

title (str): title of the file in (human friendly format)

description (str): long text box to describe file (background/use notes)

tags (list): Must be a list of strings.

functional\_area (str): The functional area.

target (str): The target.

target\_type (str): The target type of the dataset.

activity (str): The activity of the dataset.

assay\_category (str): The assay category of the dataset.

data\_df (DataFrame): DataFrame to be uploaded.

dtc\_smiles\_fileID(str): Source file id used to generate data\_df.

smiles\_column (str): Column containing SMILES.

data\_origin (str): The origin of the dataset e.g. journal.

species (str): The species of the dataset e.g. human, rat, dog.

force\_update (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

```
utils.data_curation_functions.upload_file_escape_raw_data(dset_name, title, description, tags,  
                                                         functional_area, target, target_type,  
                                                         activity, assay_category, file_path,  
                                                         data_origin='journal', species='human',  
                                                         force_update=False)
```

Uploads raw Escape data to the datastore

Upload a raw dataset to the datastore from the given DataFrame. Returns the datastore OID of the uploaded dataset. The dataset is uploaded to the public bucket and lists <https://dx.doi.org/10.1186%2Fs13321-017-0203-5> as the doi. This also assumes that the id\_col is 'Original\_Entry\_ID'

**Args:**

dset\_name (str): Name of the dataset. Should not include a file extension.

title (str): title of the file in (human friendly format)

description (str): long text box to describe file (background/use notes)

tags (list): Must be a list of strings.

functional\_area (str): The functional area.

target (str): The target.

target\_type (str): The target type of the dataset.

activity (str): The activity of the dataset.

assay\_category (str): The assay category of the dataset.

file\_path (str): The filepath of the dataset.

data\_origin (str): The origin of the dataset e.g. journal.

species (str): The species of the dataset e.g. human, rat, dog.

force\_update (bool): Overwrite existing datasets in the datastore.

**Returns:**

str: datastore OID of the uploaded dataset.

## utils.datastore\_functions module

This file contains functions to make it easier to browse and retrieve data from the datastore. Intended for general use. Add/modify functions as needed. Created 23Jul18 CHW

```
utils.datastore_functions.bulk_export_kv_for_files(files, save_as, client=None)
```

exports a csv file with 3 columns: bucket, dataset\_key, key/value pairs to make reviewing metadata easier

**Args:**

files (list of tuples): format [(bucket1, dataset\_key1), (bucket2, dataset\_key2)]

save\_as (str): filename to use for new file

**Returns:**

None

```
utils.datastore_functions.bulk_update_kv(file, client=None, i=0)
```

this function allows you to upload a properly formatted csv file with 4 columns (order and spelling of headings must match): bucket, dataset\_key, kv\_add, kv\_del the metadata for the files listed will then be updated in the Datastore

`utils.datastore_functions.check_key_val(key_values, client=None, df=None, enforced=True)`

Checks to ensure the keys and values specified are ‘approved’ and that (optionally) all required keys are filled out.

**Args:**

key\_values (dict): keys and values specified by user for a file  
 client (optional): set client if not using the default  
 df (DataFrame): dataframe to be uploaded  
 enforced (bool, optional): If True (default) checks that all required keys are filled out

**Returns:**

(bool): returns True if all keys and values are ‘approved’ AND enforcement criteria are met

`utils.datastore_functions.config_client(token=None, url='https://twintron-blue.llnl.gov/atom/datastore/api/v1.0/swagger.json', new_instance=False)`

Configures client to access datastore service.

**Args:**

token (str): Path to file containing token for accessing datastore. Defaults to /usr/local/data/ds\_token.txt on non-LC systems, or to \$HOME/data/ds\_token.txt on LC systems.  
 url (str): URL for datastore REST service.  
 new\_instance (bool): True to force creation of a new client object. By default, a shared singleton object is returned.

**Returns:**

returns configured client

`utils.datastore_functions.copy_datasets_to_bucket(dataset_keys, from_bucket, to_bucket, ds_client=None)`

Copy each named dataset from one bucket to another.

**Args:**

dataset\_keys (str or list of str): List of dataset\_keys for datasets to move.  
 from\_bucket (str): Bucket where datasets are now.  
 to\_bucket (str): Bucket to move datasets to.

**Returns:**

None

`utils.datastore_functions.dataset_key_exists(dataset_key, bucket, client=None)`

Returns a boolean indicating whether the given dataset\_key is already present in the bucket specified.

**Args:**

dataset\_key (str): the dataset\_key for the dataset you want (unique in each bucket)  
 bucket (str): the bucket the dataset you want resides in  
 client (optional): set client if not using the default

**Returns:**

(bool): returns ‘True’ if dataset\_key is present in bucket specified

```
utils.datastore_functions.filter_datasets_interactive(bucket='all', client=None,
                                                    save_search=False, restrict_key=True,
                                                    restrict_value=False,
                                                    dataset_oid_only=False,
                                                    display_all_columns=False, max_rows=10)
```

This is an old way of searching for files. Not based on the current format. Only use

**Args:**

bucket (str or list, optional): buckets to search (defaults to searching all buckets you have access to in the datastore)

client (optional): set client if not using the default

restrict\_key (bool, optional): if set to True, restricts the search to keys that are on the approved list (see file in bucket with dataset\_key: accepted\_key\_values)

restrict\_value (bool, optional): if set to True, restricts the search to values that are on the approved list (see file in bucket with dataset\_key: accepted\_key\_values)

dataset\_oid\_only (bool, optional): if True, return a list of dataset\_oids meeting the criteria; if False, returns a dataframe of all the metadata for the files meeting search criteria

display\_all\_columns (bool, optional): If 'False' (default), then show only a selected subset of the columns

max\_rows (int, optional): maximum rows to display during interactive search

**Returns:**

None

```
utils.datastore_functions.get_key_val(metadata, key=None)
```

Simple utility to search through list of key value pairs and return values for query key

**Args:**

**metadata list of key,value pairs (list): a list with position 0 = string/list of buckets, and remaining positions dictionaries of search criteria**

**example:**

```
[{'key': 'species', 'value': ['rat'] }, {'key': 'assay_category','value': ['solubility', 'volume_of_distribution']}]
```

key (str): key to search for

**Returns:**

returns When key is provide, returns value for matching key if found, None otherwise returns dictionary for the list of key,value pairs when a query key is not provided.

```
utils.datastore_functions.get_keyval(dataset_oid=None, dataset_key=None, bucket=None, client=None)
```

Requires either dataset\_oid or dataset\_key+bucket. Function extracts the key:value pairs and converts from the 'datastore format' (list of dictionaries) into 'model tracker format' (a single dictionary).

```
utils.datastore_functions.initialize_model_tracker(new_instance=False)
```

Create or obtain a client object for the model tracker service..

**Returns:**

mlmt\_client (MLMTClientSingleton): The client object for the model tracker service.

```
utils.datastore_functions.key_exists(key, bucket='all', client=None)
```

Check if key exists in bucket(s) specified.

**Args:**

key (str): the key of interest

bucket (str or list, optional): 'all' by default. Specify bucket (as a str or list) to limit search

client (optional): set client if not using the default

**Returns:**

(bool): Returns True if key exists in bucket(s) specified

`utils.datastore_functions.list_key_values(bucket, input_key, category='experimental', client=None)`

List the values for input key. Requires that the input key be in the 'approved' list

**Args:**

bucket (str or list, optional): buckets to search (defaults to searching all buckets you have access to in the datastore)

input\_key: user specified key to query

category: 'experimental' or 'pdb\_bind'

client (optional): set client if not using the default

**Returns:**

None

`utils.datastore_functions.repeat_defined_search(defined_search, client=None, to_return='df', display_all_columns=False)`

Retrieves a DataFrame of files (and associated metadata) meeting the search criteria. This is designed to work well with the output from the filter\_datasets\_interactive function with defined\_search=True

**Args:**

**defined\_search (list): a list with position 0 = string/list of buckets, and remaining positions dictionaries of search criteria**

**example: defined\_search = ['gsk\_ml',  
{ 'key': 'species', 'value': ['rat'], 'operator': 'in' }, { 'key': 'assay\_category', 'value': ['solubility', 'volume\_of\_distribution'], 'operator': 'in' }]**

client (optional): set client if not using the default to\_return (str, optional): (default=df)

'df' (df\_results) = return a pandas dataframe summarizing metadata of files meeting criteria oid'  
(dataset\_oid) = return a list of dataset\_oids meeting criteria ds\_key' (dataset\_key) = return a list of dataset\_key + bucket tuples

display\_all\_column (bool, optional): default False. If True, displays all associated metadata instead of just a selected subset

**Returns:**

One of the following will be returned (based on selection for 'to\_return') (DataFrame): dataframe of meta-data for the files matching the criteria specified in the search (list): list of dataset\_oids meeting the criteria specified in the search (list): list of bucket and dataset\_key meeting the criteria specified in the search

`utils.datastore_functions.retrieve_bucket_names(client=None)`

Retrieve a list of the bucket names in datastore

**Args:**

client (optional): set client if not using the default

**Returns:**

(list): list of bucket names that exist in the datastore which user has access to

`utils.datastore_functions.retrieve_columns_from_dataset(bucket, dataset_key, client=None, max_rows=0, column_names="", return_names=False)`

**Retrieve column(s) from csv file (may be bz2 compressed) in datastore.**

‘NA’ returned if column not in file (as well as warning message).

**Args:**

return\_names (bool): If true, just return column headers from file

max\_rows (int): default=0 which will return all rows

client (optional): set client if not using the default

**Returns:**

(dict): dictionary corresponding to selected columns

```
utils.datastore_functions.retrieve_dataset_by_dataset_oid(dataset_oid, client=None,
                                                         return_metadata=False, nrows=None,
                                                         print_metadata=False, sep=False,
                                                         index_col=None, tarpath='.')
```

retrieves the dataset and returns as a pandas dataframe (or other format as needed depending on file type).

**Args:**

dataset\_oid (str): unique identifier for the dataset you want

client (optional): set client if not using the default

return\_metadata (bool, optional): if set to True, return a dictionary of the metadata INSTEAD of a dataframe of the data

nrows (num, optional): used to limit the number of rows returned

print\_metadata (bool, optional): if set to True, displays the document metadata/properties

sep (str, optional): separator used for csv file

tarpath (str, optional): path to use for tarball files

index\_col (int, optional): For csv files, column to use as the row labels of the DataFrame

**Returns:**

(DataFrame, OrderedDict, str, dict): filetype determines what type of object is returned. xls and xlsx files returns an OrderedDict. tarball (gz and tgz) files returns the location of the files as a string csv returns a DataFrame

optionally, return a dictionary of the metadata only if ‘return\_metadata’ is set to TRUE.

```
utils.datastore_functions.retrieve_dataset_by_datasetkey(dataset_key, bucket, client=None,
                                                         return_metadata=False, nrows=None,
                                                         print_metadata=False, sep=False,
                                                         index_col=None, tarpath='.', **kwargs)
```

Retrieves the dataset and returns as a pandas dataframe (or other format as needed depending on file type).

**Args:**

dataset\_key (str): the dataset\_key for the dataset you want (unique in each bucket)

bucket (str): the bucket the dataset you want resides in

client (optional): set client if not using the default

return\_metadata (bool, optional): if set to True, return a dictionary of the metadata INSTEAD of a dataframe of the data

nrows (num, optional): used to limit the number of rows returned

print\_metadata (bool, optional): if set to True, displays the document metadata/properties

sep (str, optional): separator used for csv file



tarpath (str, optional): path to use for tarball files

index\_col (int, optional): For csv files, column to use as the row labels of the DataFrame

**Returns:**

(DataFrame, OrderedDict, str, dict): filetype determines what type of object is returned. xls and xlsx files returns an OrderedDict. tarball (gz and tgz) files returns the location of the files as a string csv returns a DataFrame

optionally, return a dictionary of the metadata only if 'return\_metadata' is set to TRUE.

`utils.datastore_functions.retrieve_keys(bucket='all', client=None, sort=True)`

Get a list of keys in bucket(s) specified.

**Args:**

bucket (str, optional): 'all' by default. Specify bucket (as a str or list) to limit search

client (optional): set client if not using the default

sort (bool, optional): if 'True' (default), sort the keys alphabetically

**Returns:**

(list): returns a list of keys in bucket(s) specified

`utils.datastore_functions.retrieve_values_for_key(key, bucket='all', client=None)`

Get a list of values associated with a specified key.

**Args:**

key (str): the key of interest

bucket (str or list, optional): 'all' by default. Specify bucket (as a str or list) to limit search

client (optional): set client if not using the default

**Returns:**

(list): Returns a list of values (str) associated with a specified key

`utils.datastore_functions.search_datasets_by_key_value(key, value, client=None, operator='in', bucket='all', display_all_columns=False)`

Find datasets by key:value pairs and returns a DataFrame of datasets and associated properties.

**Args:**

key (str): the key of interest

value (str): the value of interest

client (optional): set client if not using the default

**operator (str, optional): 'in' by default, but can be changed to any of the following:**

=, !=, <, <=, >, >=, all, in, not in

bucket (str or list, optional): 'all' by default. Specify bucket (as a str or list) to limit search

display\_all\_columns (bool, optional): If 'False' (default), then show only a selected subset of the columns

**Returns:**

(DataFrame): summary table of the files and relevant metadata matching the criteria specified

`utils.datastore_functions.search_files_interactive(bucket='all', client=None, to_return='df', display_all_columns=False, max_rows=10)`

This tool helps you find the files you need via an interactive/guided interface.

**Args:**

bucket (str or list, optional): buckets to search (defaults to searching all buckets you have access to in the datastore)

client (optional): set client if not using the default

**to\_return (str): 'df' (df\_results) = return a pandas dataframe summarizing metadata of files meeting criteria**

**'search' (search\_criteria) = return a list containing search criteria where position 0 = string/list of buckets, and remaining positions are dictionaries of search criteria.**

Designed to work with 'repeat\_defined\_search' function.

'oid' (dataset\_oid) = return a list of dataset\_oids meeting criteria 'ds\_key' (dataset\_key) = return a list of dataset\_key + bucket tuples

display\_all\_columns (bool, optional): If 'False' (default), then show only a selected subset of the columns

max\_rows (int, optional): maximum rows to display during interactive search

**Returns:**

None

`utils.datastore_functions.string_to_dict(dict_string)`

`utils.datastore_functions.string_to_list(list_string)`

`utils.datastore_functions.summarize_datasets(dataset_keys, bucket, client=None, column=None, save_as=None, plot_ht=10, labels=None, last=False)`

Generate summary statistics such as min/max/median/mean on files specified (all files must be in same bucket).

**Args:**

dataset\_keys (list): dataset\_keys corresponding to the files to summarize

bucket (str): bucket the files reside in

client (optional): set client if not using the default

column (str, optional): column to summarize (will be prompted to specify if not pre-specified or if column does not exist in file)

save\_as (str, optional): filename to save image of box plot(s) to

plot\_ht (int, optional): height of box plots (default = 10)

labels ('str', optional):

last (bool optional): If True (default=False), then summarize values from last column instead of specifying column heading

**Returns**

(DataFrame): returns table summarizing the stats for the file(s) specified

`utils.datastore_functions.update_distribution_kv(bucket, dataset_key, client=None, kv_add=None, kv_del=None, return_metadata=False)`

update the key:values for specified file. No change to file.

**Args:**

bucket (str): Specify bucket where the file exists

dataset\_key (str): dataset\_key for the file to update metadata for

client (optional): set client if not using the default

kv\_add (dict, optional): key-value pairs to add to the metadata for the file specified

`kv_del` (str or list, optional): keys to delete from the metadata for the file specified

**Returns:**

None

`utils.datastore_functions.update_kv(bucket, dataset_key, client=None, kv_add=None, kv_del=None, return_metadata=False)`

update the key:values for specified file. No change to file.

**Args:**

`bucket` (str): Specify bucket where the file exists

`dataset_key` (str): `dataset_key` for the file to update metadata for

`client` (optional): set client if not using the default

`kv_add` (dict, optional): key-value pairs to add to the metadata for the file specified

`kv_del` (str or list, optional): keys to delete from the metadata for the file specified

**Returns:**

None

`utils.datastore_functions.upload_df_to_DS(df, bucket, filename, title, description, tags, key_values, client=None, dataset_key=None, override_check=True, return_metadata=False, index=False, data_type=None)`

This function will upload a file to the Datastore along with the associated metadata

**Args:**

`df` (DataFrame): dataframe to be uploaded

`bucket` (str): bucket the file will be put in

`filename` (str): the filename to save the dataframe as in the datastore. Include the extension

`title` (str): title of the file in (human friendly format)

`description` (str): long text box to describe file (background/use notes)

`tags` (list): must be a list.

`key_values` (dict): key-value pairs to enable future users to find the file. Must be a dictionary.

`client` (optional): set client if not using the default

`dataset_key` (str): If updating a file already in the datastore enter the corresponding `dataset_key`. If not, leave as 'none' and the `dataset_key` will be automatically generated.

`data_type` (str,optional): Specify `data_type` (e.g. csv,bz, etc) if not specified attempt to use file extension

**Returns:**

(dict): if `return_metadata=True`, then function returns a dictionary of the metadata for the uploaded dataset.

`utils.datastore_functions.upload_file_to_DS(bucket, title, description, tags, key_values, filepath, filename, client=None, dataset_key=None, override_check=True, return_metadata=False, file_ref=False, data_type=None)`

This function will upload a file to the Datastore along with the associated metadata

**Args:**

`bucket` (str): bucket the file will be put in

`title` (str): title of the file in (human friendly format)

`description` (str): long text box to describe file (background/use notes)

tags (list): must be a list.

key\_values (dict): key:value pairs to enable future users to find the file. Must be a dictionary.

filepath (str): current location of the file

filename (str): current filename of the file

client (optional): set client if not using the default

dataset\_key (str, optional): If updating a file already in the datastore enter the corresponding dataset\_key. If not, leave as 'none' and the dataset\_key will be automatically generated.

override\_check (bool, optional): If 'True' then do NOT perform a check of the keys/values against approved list and enforcement criteria

return\_metadata (bool, optional): If 'True' (default=False), then return the metadata from the uploaded file

file\_ref (bool, optional): If 'True' (default=False), links file to the datastore instead of creating a copy to managed by the datastore.

data\_type (str, optional): Specify dataType (e.g. csv,bz, etc) if not specified attempt to use file extension

**Returns:**

(dict): optionally returns the metadata from the uploaded file (if return\_metadata=True)

```
utils.datastore_functions.upload_pickle_to_DS(data, bucket, filename, title, description, tags,
                                             key_values, client=None, dataset_key=None,
                                             override_check=True, return_metadata=False)
```

This function will upload a file to the Datastore along with the associated metadata.

**Args:**

data (DataFrame, str, list, tuple, pickle): data to be pickled and uploaded

bucket (str): bucket the file will be put in

filename (str): the filename to save the dataframe as in the datastore. Include the extension

title (str): title of the file in (human friendly format)

description (str): long text box to describe file (background/use notes)

tags (list): must be a list.

key\_values (dict): key:value pairs to enable future users to find the file. Must be a dictionary.

client (optional): set client if not using the default

**dataset\_key (str, optional): If updating a file already in the datastore enter the corresponding dataset\_key.**

If not, leave as 'none' and the dataset\_key will be automatically generated.

override\_check (bool, optional): If True, overrides checking the metadata for the file when uploaded.

return\_metadata (bool, optional): If True, returns metadata for the file after it is uploaded.

**Returns:**

None

## utils.hyperparam\_search\_wrapper module

Script to generate hyperparameter combinations based on input params and send off jobs to a slurm system. Author: Amanda Minnich

**class** `utils.hyperparam_search_wrapper.GeometricSearch(params)`

Bases: [\*HyperparameterSearch\*](#)

Generates parameter values in logistic steps, rather than linear like GridSearch does

**generate\_assay\_list()**

Generates the list of datasets to build models for, with their key, bucket, split, and split uuid

**Returns:**

None

**generate\_combo(params\_dict)**

Method to generate all combinations from a given set of key-value pairs

**Args:**

params\_dict: Set of key-value pairs with the key being the param name and the value being the list of values you want to try for that param

**Returns:**

new\_dict: The list of all combinations of parameters

**generate\_param\_combos()**

Performs additional parsing of parameters and generates all combinations

**Returns:**

None

**split\_and\_save\_dataset(assay\_params)**

Splits a given dataset, saves it, and sets the split\_uuid in the metadata

**Args:**

assay\_params: Dataset metadata

**Returns:**

None

**class** `utils.hyperparam_search_wrapper.GridSearch(params)`

Bases: [\*HyperparameterSearch\*](#)

Generates fixed steps on a grid for a given hyperparameter range

**generate\_assay\_list()**

Generates the list of datasets to build models for, with their key, bucket, split, and split uuid

**Returns:**

None

**generate\_combo(params\_dict)**

Method to generate all combinations from a given set of key-value pairs

**Args:**

params\_dict: Set of key-value pairs with the key being the param name and the value being the list of values you want to try for that param

**Returns:**

new\_dict: The list of all combinations of parameters

### **generate\_param\_combos()**

Performs additional parsing of parameters and generates all combinations

#### **Returns:**

None

### **split\_and\_save\_dataset(*assay\_params*)**

Splits a given dataset, saves it, and sets the split\_uuid in the metadata

#### **Args:**

assay\_params: Dataset metadata

#### **Returns:**

None

### **class** `utils.hyperparam_search_wrapper.HyperOptSearch(params)`

Bases: object

Perform hyperparameter search with Bayesian Optimization (Tree Parzen Estimator)

To use HyperOptSearch, modify the config json file as follows:

serach\_type: use “hyperopt”

result\_dir: use two directories (recommended), separated by comma, 1st one will be used to save the best model tarball, 2nd one will be used to store all models during the process. e.g. “result\_dir”: “/path/of/the/final/dir,/path/of/the/temp/dir”

model\_type: RF or NN, also add max number of HyperOptSearch evaluations, e.g. “model\_type”: “RF|100”. If no max number provide, the default 100 will be used. #For NN models only

**lr: specify learning rate searching method and related parameters as the following scheme.**

method|parameter1,parameter2...

method: supported searching schemes in HyperOpt include: choice, uniform, loguniform, and uniformint, see <https://github.com/hyperopt/hyperopt/wiki/FMin> for details.

#### **parameters:**

choice: all values to search from, separated by comma, e.g. choice|0.0001,0.0005,0.0002,0.001

uniform: low and high bound of the interval to serach, e.g. uniform|0.00001,0.001

loguniform: low and high bound (in natural log) of the interval to serach, e.g. loguniform|-13.8,-6.9

uniformint: low and high bound of the interval to serach, e.g. uniformint|8,256

ls: similar as learning\_rate, specify number of layers and size of each one.

method|num\_layers|parameter1,parameter2... e.g. choice|2|8,16,32,64,128,256,512 #this will generate a two-layer config, each layer takes size from the list “8,16,32,64,128,256,512” e.g. uniformint|3|8,512 #this will generate a three-layer config, each layer takes size from the uniform interval [8,512]

dp: similar as layer\_sizes, just make sure dropouts and layer\_sizes should have the same number of layers.

e.g. uniform|3|0,0.4 #this will generate a three-layer config, each layer takes size from the uniform interval [0,0.4]

#For RF models only rfe: rf\_estimator, same structure as the learning rate above, e.g. uniformint|64,512 #take integer values from a uniform interval [64,512]

rfd: rf\_max\_depth, e.g. uniformint|8,256

rff: rf\_max\_feature, e.g. uniformint|8,128

**run\_search()**

**class** `utils.hyperparam_search_wrapper.HyperparameterSearch(params)`

Bases: `object`

The class for generating and running all hyperparameter combinations based on the input params given

**already\_run(*assay\_params*, *retry\_time=10*)**

Checks to see if a model with a given metadata combination has already been built

**Args:**

`assay_params`: model metadata information

**Returns:**

Boolean specifying if model has been previously built

**assemble\_layers()**

Reformats layer parameters

**Returns:**

None

**build\_jobs()**

Builds jobs. Reformats parameters as necessary

**Returns:**

None

**filter\_jobs(*job\_list*)**

Removes jobs that should not be run

**Returns:**

None

**generate\_assay\_list()**

Generates the list of datasets to build models for, with their key, bucket, split, and split uuid

**Returns:**

None

**generate\_combo(*params\_dict*)**

This is implemented in the specific sub-classes

**generate\_combos(*params\_dict*)**

Calls sub-function `generate_combo` and then uses `itertools.product` to generate all desired combinations

**Args:**

`params_dict`:

**Returns:**

None

**generate\_maestro\_commands()**

Generates commands that can be used by maestro

Generates a list of commands that can be put directly into the shell to run model training.

**Args:**

None

**Returns:**

list: A list of shell commands

### **generate\_param\_combos()**

Performs additional parsing of parameters and generates all combinations

#### **Returns:**

None

### **generate\_searches()**

Generate a list of training jobs

Generates a list of model training jobs that spans the hyperparameter search space. This function filters out jobs that are redundant by calling filter\_jobs

#### **Args:**

None

#### **Returns:**

list(tuple): A list of tuples that contain assay parameters

### **generate\_split\_shortlist(retry\_time=60)**

Processes a shortlist, generates splits for each dataset on the list, and uploads a new shortlist file with the split\_uuids included. Generates splits for the split\_combos [[0.1,0.1], [0.1,0.2],[0.2,0.2]], [random, scaffold]

#### **Returns:**

None

### **generate\_split\_shortlist\_file()**

Processes a shortlist, generates splits for each dataset on the list, and uploads a new shortlist file with the split\_uuids included. Generates splits for the split\_combos [[0.1,0.1], [0.15,0.15], [0.1,0.2], [0.2,0.2]], [random, scaffold]

#### **Returns:**

None

### **get\_dataset\_metadata(assay\_params, retry\_time=60)**

Gather the required metadata for a dataset

#### **Args:**

assay\_params: dataset metadata

#### **Returns:**

None

### **get\_shortlist\_df(split\_uuids=False, retry\_time=60)**

Get dataframe short list

#### **Args:**

split\_uuids: Boolean value saying if you want just datasets returned or the split\_uuids as well

#### **Returns:**

The list of dataset\_keys, along with their accompanying bucket, split type, and split\_uuid if split\_uuids is True

### **return\_split\_uuid(dataset\_key, bucket=None, splitter=None, split\_combo=None, retry\_time=60)**

Loads a dataset, splits it, saves it, and returns the split\_uuid

#### **Args:**

dataset\_key: key for dataset to split

bucket: datastore-specific user group bucket

splitter: Type of splitter to use to split the dataset



split\_combo: tuple of form (split\_valid\_frac, split\_test\_frac)

**Returns:**

None

**return\_split\_uuid\_file**(*dataset\_key, response\_cols, bucket=None, splitter=None, split\_combo=None, retry\_time=60*)

Loads a dataset, splits it, saves it, and returns the split\_uuid.

**Args:**

dataset\_key: key for dataset to split

bucket: datastore-specific user group bucket

splitter: Type of splitter to use to split the dataset

split\_combo: tuple of form (split\_valid\_frac, split\_test\_frac)

**Returns:**

None

**run\_search()**

The driver code for generating hyperparameter combinations and submitting jobs

**Returns:**

None

**split\_and\_save\_dataset**(*assay\_params*)

Splits a given dataset, saves it, and sets the split\_uuid in the metadata

**Args:**

assay\_params: Dataset metadata

**Returns:**

None

**submit\_jobs**(*job\_list, retry\_time=60*)

Reformats parameters as necessary and then calls run\_command in a loop to submit a job for each param combo

**Returns:**

None

**class** `utils.hyperparam_search_wrapper.RandomSearch`(*params*)

Bases: [HyperparameterSearch](#)

Generates the specified number of random parameter values for within the specified range

**generate\_assay\_list()**

Generates the list of datasets to build models for, with their key, bucket, split, and split uuid

**Returns:**

None

**generate\_combo**(*params\_dict*)

Method to generate all combinations from a given set of key-value pairs

**Args:**

params\_dict: Set of key-value pairs with the key being the param name and the value being the list of values you want to try for that param

**Returns:**

new\_dict: The list of all combinations of parameters

**generate\_param\_combos()**

Performs additional parsing of parameters and generates all combinations

**Returns:**

None

**split\_and\_save\_dataset(*assay\_params*)**

Splits a given dataset, saves it, and sets the split\_uuid in the metadata

**Args:**

assay\_params: Dataset metadata

**Returns:**

None

**class** `utils.hyperparam_search_wrapper.UserSpecifiedSearch(params)`

Bases: [\*HyperparameterSearch\*](#)

Generates combinations using the user-specified steps

**generate\_assay\_list()**

Generates the list of datasets to build models for, with their key, bucket, split, and split uuid

**Returns:**

None

**generate\_combo(*params\_dict*)**

Method to generate all combinations from a given set of key-value pairs

**Args:**

params\_dict: Set of key-value pairs with the key being the param name and the value being the list of values you want to try for that param

**Returns:**

new\_dict: The list of all combinations of parameters

**generate\_param\_combos()**

Performs additional parsing of parameters and generates all combinations

**Returns:**

None

**split\_and\_save\_dataset(*assay\_params*)**

Splits a given dataset, saves it, and sets the split\_uuid in the metadata

**Args:**

assay\_params: Dataset metadata

**Returns:**

None

`utils.hyperparam_search_wrapper.build_hyperopt_search_domain(label, method, param_list)`

Generate HyperOpt search domain object from method and parameters, layer\_nums is only for NN models. This function is used by the HyperOptSearch class, not intended for standalone usage.

`utils.hyperparam_search_wrapper.build_search(params)`

Builds HyperparamterSearch object

Looks at params.search\_type and builds a HyperparamSearch object of the correct flavor. Will exit if the search\_type is not recognized.

**Args:**

**params (Namespace):** Namespace returned by  
atomscli.ddm.pipeline.parameter\_parser.wrapper()

**Returns:**

HyperparameterSearch

`utils.hyperparam_search_wrapper.gen_maestro_command(python_path, script_dir, params)`

Generates a string that can be fed into a command line.

**Side Effects:**

**Dataset key will be converted to an absolute path before**  
returned. It's difficult to predict the working directory used when maestro runs the script.

**Args:**

*shell\_script*: Name of shell script to run

*python\_path*: Path to python version

*script\_dir*: Directory where script lives

*params*: parameters in dictionary format

**Returns:**

str: Formatted command in the form of a string

`utils.hyperparam_search_wrapper.get_num_params(combo)`

Calculates the number of parameters in a fully-connected neural network

**Args:**

*combo*: Model parameters

**Returns:**

tmp\_sum: Calculated number of parameters

`utils.hyperparam_search_wrapper.main()`

Entry point when script is run

**Args:**

None

**Returns:**

None

`utils.hyperparam_search_wrapper.parse_params(param_list)`

Parse paramters

Parses parameters using parameter\_parser.wrapper and filters out unnecessary parameters. Returns what an  
argparse.Namespace

**Args:**

\**any\_arg*: any single input of a str, dict, argparse.Namespace, or list

**Returns:**

argparse.Namespace

`utils.hyperparam_search_wrapper.permutate_NNlayer_combo_params(layer_nums, node_nums,  
dropout_list,  
max_final_layer_size)`

Generate combos of layer\_sizes(str) and dropouts(str) params from the layer\_nums (list), node\_nums (list), dropout\_list (list).

The permutation will make the NN funnel shaped, so that the next layer can only be smaller or of the same size of the current layer.

**Example:**

permutate\_NNlayer\_combo\_params([2], [4,8,16], [0], 16) returns [[16, 4], [16, 8], [8,4]] [[0,0],[0,0],[0,0]]

If there are duplicates of the same size, it will create consecutive layers of the same size.

**Example:**

permutate\_NNlayer\_combo\_params([2], [4,8,8], [0], 16) returns [[8, 8], [8, 4]] [[0,0],[0,0]]

**Args:**

layer\_nums: specify numbers of layers.

node\_nums: specify numbers of nodes per layer.

dropout\_list: specify the dropouts.

max\_last\_layer\_size: sets the max size of the last layer. It will be set to the smallest node\_num if needed.

**Returns:**

layer\_sizes, dropouts: the layer sizes and dropouts generated based on the input parameters

`utils.hyperparam_search_wrapper.reformat_filter_dict(filter_dict)`

Function to reformat a filter dictionary to match the Model Tracker metadata structure. Updated 9/2020 by A. Paulson for new LC model tracker.

**Args:**

filter\_dict: Dictionary containing metadata for model of interest

**Returns:**

new\_filter\_dict: Filter dict reformatted

`utils.hyperparam_search_wrapper.run_cmd(cmd)`

Function to submit a job using subprocess

**Args:**

cmd: Command to run

**Returns:**

output: Output of command

`utils.hyperparam_search_wrapper.run_command(shell_script, python_path, script_dir, params)`

Function to submit jobs on a slurm system

**Args:**

shell\_script: Name of shell script to run

python\_path: Path to python version

script\_dir: Directory where script lives

params: parameters in dictionary format

**Returns:**

None

## utils.many\_to\_one module

**exception** `utils.many_to_one.ManyToOneException`

Bases: `Exception`

**exception** `utils.many_to_one.NANCompoundIDException`

Bases: `Exception`

**exception** `utils.many_to_one.NANSMILESException`

Bases: `Exception`

`utils.many_to_one.has_nans(df, col)`

`utils.many_to_one.many_to_one(fn, smiles_col, id_col)`

`utils.many_to_one.many_to_one_df(df, smiles_col, id_col)`

AMPL requires that SMILES and compound\_ids have a many to one mapping. This function opens the dataset and checks this restraint. It will also check if any SMILES or compound\_ids are empty/nan

### Arguments:

`df` (`pd.DataFrame`): The DataFrame in question. `smiles_col` (`str`): The column containing SMILES. `id_col` (`str`): The column containing compound ids

### Returns:

**True if there is a many to one mapping. Raises one of 3 errors if it:**

- Has nan compound\_ids
- Has nan SMILES
- Is not a many to one mapping between compound\_ids and SMILES

`utils.many_to_one.no_nan_ids_or_smiles(df, smiles_col, id_col)`

## utils.model\_file\_reader module

**class** `utils.model_file_reader.ModelFileReader(data_file_path)`

Bases: `object`

A class to encapsulate a model's metadata that you might want read out from a folder. Like read version number, get the dataset key, split uuid etc of a model.

### Attributes:

#### Set in `__init__`:

`data_file_path` (`str`): a model data file or a directory that contains the model

**`get_dataset_key()`**

Returns: (`str`): model dataset key

**`get_descriptor_type()`**

Returns: (`str`): model descriptor type

**`get_featurizer()`**

Returns: (`str`): model featurizer

**`get_id_col()`**

Returns: (`str`): model id column

**get\_model\_info()**

Extract the model metadata (and if applicable, model metrics)

**Returns:**

a dictionary of the most important model parameters and metrics.

**get\_model\_parameters()**

Returns: (str): model parameters

**get\_model\_type()**

Returns: (str): model type

**get\_model\_uuid()**

Returns: (str): model uuid

**get\_response\_cols()**

Returns: (str): model response columns

**get\_smiles\_col()**

Returns: (str): model smile columns

**get\_split\_csv()**

Returns: (str): model split csv

**get\_split\_strategy()**

Returns: (str): model split strategy

**get\_split\_uuid()**

Returns: (str): model split\_uuid

**get\_splitter()**

Returns: (str): model splitter

**get\_splitting\_parameters()**

Returns: (str): model splitting parameters

**get\_training\_dataset()**

Returns: (str): model training dataset

**get\_version()**

Returns: (str): model version

**utils.model\_file\_reader.get\_multiple\_models\_metadata(\*args)**

A function that takes model tar.gz file(s) and extract the metadata (and if applicable, model metrics)

**Args:**

\*args: Variable length argument list of model tar.gz file(s)

**Returns:**

a list of models' most important model parameters and metrics. or an empty array if it fails to parse the input file(s).

**Exception:**

IOError: Problem access the file or if fails to parse the input file to an AMPL model

**utils.model\_file\_reader.main(argv)**

## utils.model\_retrain module

`utils.model_retrain.main(argv)`

`utils.model_retrain.train_model(input, output, dskey="", production=False)`

Retrain a model saved in a model\_metadata.json file

**Args:**

input (str): path to model\_metadata.json file  
 output (str): path to output directory  
 dskey (str): new dataset key if file location has changed  
 production (bool): retrain the model using production mode

**Returns:**

None

`utils.model_retrain.train_model_from_tar(input, output, dskey="", production=False)`

Retrain a model saved in a tar.gz file

**Args:**

input (str): path to a tar.gz file  
 output (str): path to output directory  
 dskey (str): new dataset key if file location has changed

**Returns:**

None

`utils.model_retrain.train_model_from_tracker(model_uuid, output_dir, production=False)`

Retrain a model saved in the model tracker, but save it to output\_dir and don't insert it into the model tracker

**Args:**

model\_uuid (str): model tracker model\_uuid file  
 output\_dir (str): path to output directory

**Returns:**

the model pipeline object with trained model

`utils.model_retrain.train_models_from_dataset_keys(input, output, pred_type='regression',  
 production=False)`

Retrain a list of models from an input file

**Args:**

input (str): path to an Excel or csv file. the required columns are 'dataset\_key' and 'bucket' (public, private\_file or Filesystem).  
 output (str): path to output directory  
 pred\_type (str, optional): set the model prediction type. if not, uses the default 'regression'

**Returns:**

None

## utils.model\_version\_utils module

model\_version\_utils.py

Misc utilities to get the AMPL version(s) used to train one or more models and check them for compatibility with the currently running version of AMPL.:

To check the model version

usage: model\_version\_utils.py [-h] -i INPUT

**optional arguments:**

**-h, --help** show this help message and exit

**-i INPUT, --input INPUT** input directory/file (required)

utils.model\_version\_utils.**check\_version\_compatible**(input, ignore\_check=False)

Compare the input file's version against the running AMPL version to see if they are compatible

**Args:**

filename (str): file or version number

**Returns:**

True if the input model version matches the compatible AMPL version group

utils.model\_version\_utils.**get\_ampl\_version**()

Get the running ampl version

**Returns:**

the AMPL version

utils.model\_version\_utils.**get\_ampl\_version\_from\_dir**(dirname)

Get the AMPL versions for all the models stored under the given directory and its subdirectories, recursively.

**Args:**

dirname (str): directory

**Returns:**

list of AMPL versions

utils.model\_version\_utils.**get\_ampl\_version\_from\_json**(metadata\_path)

Parse model\_metadata.json to get the AMPL version

**Args:**

filename (str): tar file

**Returns:**

the AMPL version number

utils.model\_version\_utils.**get\_ampl\_version\_from\_model**(filename)

Get the AMPL version from the tar file's model\_metadata.json

**Args:**

filename (str): tar file

**Returns:**

the AMPL version number

utils.model\_version\_utils.**get\_major\_version**(full\_version)

utils.model\_version\_utils.**main**(argv)

utils.model\_version\_utils.**validate\_version**(input)



## utils.pubchem\_utils module

`utils.pubchem_utils.download_SID_from_bioactivity_assay(bioassayid)`

Retrieve summary info on bioactivity assays.

**Args:**

a single bioactivity id: PubChem AIDs (bioactivity assay ids)

**Returns:**

Returns the sids tested on this assay

`utils.pubchem_utils.download_activitytype(aid, sid)`

Retrieve data for assays for a select list of sids.

**Args:**

myList (list): a bioactivity id (aid)

sidlst (list): list of sids specified as integers

**Returns:**

Nothing returned yet, will return basic stats to help decide whether to use assay or not

`utils.pubchem_utils.download_bioactivity_assay(myList, intv=1)`

Retrieve summary info on bioactivity assays.

**Args:**

myList (list): List of PubChem AIDs (bioactivity assay ids)

intv (1): number of INCHIKEYS to submit queries for in one request, default is 1

**Returns:**

Nothing returned yet, will return basic stats to help decide whether to use assay or not

`utils.pubchem_utils.download_dose_response_from_bioactivity(aid, sidlst)`

Retrieve data for assays for a select list of sids.

**Args:**

myList (list): a bioactivity id (aid)

sidlst (list): list of sids specified as integers

**Returns:**

Nothing returned yet, will return basic stats to help decide whether to use assay or not

`utils.pubchem_utils.download_smiles(myList, intv=1)`

Retrieve canonical SMILES strings for a list of input INCHIKEYS. Will return only one SMILES string per INCHIKEY. If there are multiple values returned, the first is retained and the others are returned in the discard\_lst. INCHIKEYS that fail to return a SMILES string are put in the fail\_lst

**Args:**

myList (list): List of INCHIKEYS

intv (1): number of INCHIKEYS to submit queries for in one request, default is 1

**Returns:**

list of SMILES strings corresponding to INCHIKEYS

list of INCHIKEYS, which failed to return a SMILES string

list of CIDs and SMILES, which were returned beyond the first CID and SMILE found for input INCHIKEY

## utils.rdkit\_easy module

Utilities for clustering and visualizing compound structures using RDKit.

`utils.rdkit_easy.add_mol_column(df, smiles_col, molecule_col='mol')`

Converts SMILES strings in a data frame to RDKit Mol objects and adds them as a new column in the data frame.

**Args:**

`df` (pd.DataFrame): Data frame to add column to.

`smiles_col` (str): Column containing SMILES strings.

`molecule_col` (str): Name of column to create to hold Mol objects.

**Returns:**

pd.DataFrame: Modified data frame.

`utils.rdkit_easy.calculate_descriptors(df, molecule_column='mol')`

Uses RDKit to compute various descriptors for compounds specified by Mol objects in the given data frame.

**Args:**

`df` (pd.DataFrame): Data frame containing molecules.

`molecule_column` (str): Name of column containing Mol objects for compounds.

**Returns:**

pd.DataFrame: Modified data frame with added columns for the descriptors.

`utils.rdkit_easy.cluster_dataframe(df, molecule_column='mol', cluster_column='cluster', cutoff=0.2)`

Performs Butina clustering on compounds specified by Mol objects in a data frame.

Modifies the input dataframe to add a column 'cluster\_column' containing the cluster index for each molecule.

From RDKit cookbook [http://rdkit.org/docs\\_temp/Cookbook.html](http://rdkit.org/docs_temp/Cookbook.html).

**Args:**

`df` (pd.DataFrame): Data frame containing compounds to cluster.

`molecule_column` (str): Name of column containing rdkit Mol objects for compounds.

`cluster_column` (str): Column that will be created to hold cluster indices.

`cutoff` (float): Maximum Tanimoto distance parameter used by Butina algorithm to identify neighbors of each molecule.

**Returns:**

None. Input data frame will be modified in place.

`utils.rdkit_easy.cluster_fingerprints(fps, cutoff=0.2)`

Performs Butina clustering on compounds specified by a list of fingerprint bit vectors.

From RDKit cookbook [http://rdkit.org/docs\\_temp/Cookbook.html](http://rdkit.org/docs_temp/Cookbook.html).

**Args:**

`fps` (list of rdkit.ExplicitBitVect): List of fingerprint bit vectors.

`cutoff` (float): Cutoff distance parameter used to seed clusters in Butina algorithm.

**Returns:**

tuple of tuple: Indices of fingerprints assigned to each cluster.

`utils.rdkit_easy.matching_atoms_and_bonds(mol, match_mol)`

Returns lists of indices of atoms and bonds within molecule *mol* that are part of the substructure matched by *match\_mol*.

**Args:**

*mol* (rdkit.Chem.Mol): Object representing molecule.

*match\_mol* (rdkit.Chem.Mol): Object representing a substructure or SMARTS pattern to be compared against *mol*, typically created by *Chem.MolFromSmiles()* or *Chem.MolFromSmarts()*.

**Returns:**

*match\_atoms*, *match\_bonds* (tuple(list(int), list(int))): Lists of indices of atoms and bonds within *mol* contained in the substructure (if any) matched by *match\_mol*. Returns empty lists if there is no match.

`utils.rdkit_easy.mol_to_html(mol, highlight=None, name="", type='svg', directory='rdkit_svg', embed=False, width=400, height=200)`

Creates an image displaying the given molecule's 2D structure, and generates an HTML tag for it. The image can be embedded directly into the HTML tag or saved to a file.

**Args:**

*mol* (rdkit.Chem.Mol): Object representing molecule.

*highlight* (rdkit.Chem.Mol): Optional object representing a set of atoms and bonds to be highlighted in the image.

*name* (str): Filename of image file to create, relative to 'directory'; only used if *embed=False*.

*type* (str): Image format; must be 'png' or 'svg'.

*directory* (str): Path relative to notebook directory of subdirectory where image file will be written. The directory will be created if necessary. Note that absolute paths will not work in notebooks. Ignored if *embed=True*.

*embed* (bool): If True, image data will be embedded in the generated HTML tag. Otherwise it will be written to a file determined by the *directory* and *name* arguments.

*width* (int): Width of image bounding box.

*height* (int): Height of image bounding box.

**Returns:**

str: HTML image tag referencing the image file.

`utils.rdkit_easy.mol_to_pil(mol, size=(400, 200), highlight=None)`

Returns a Python Image Library (PIL) object containing an image of the given molecule's structure.

**Args:**

*mol* (rdkit.Chem.Mol): Object representing molecule.

*size* (tuple): Width and height of bounding box of image.

*highlight* (rdkit.Chem.Mol): Object representing substructure to highlight on molecule.

**Returns:**

PIL.PngImageFile: An object containing an image of the molecule's structure.

`utils.rdkit_easy.mol_to_svg(mol, size=(400, 200), highlight=None)`

Returns a RDKit MolDraw2DSVG object containing an image of the given molecule's structure.

**Args:**

*mol* (rdkit.Chem.Mol): Object representing molecule.

*size* (tuple): Width and height of bounding box of image.

highlight (rdkit.Chem.Mol): Object representing substructure to highlight on molecule.

**Returns:**

RDKit.rdMolDraw2D.MolDraw2DSVG text (str): An SVG object containing an image of the molecule's structure.

`utils.rdkit_easy.save_png(mol, name, size=(400, 200), highlight=None)`

Draws the molecule mol into a PNG file with filename 'name' and with the given size in pixels.

**Args:**

mol (rdkit.Chem.Mol): Object representing molecule.

name (str): Path to write PNG file to.

size (tuple): Width and height of bounding box of image.

highlight (rdkit.Chem.Mol): Object representing substructure to highlight on molecule.

`utils.rdkit_easy.save_svg(mol, name, size=(400, 200), highlight=None)`

Draws the molecule mol into an SVG file with filename 'name' and with the given size in pixels.

**Args:**

mol (rdkit.Chem.Mol): Object representing molecule.

name (str): Path to write SVG file to.

size (tuple): Width and height of bounding box of image.

highlight (rdkit.Chem.Mol): Object representing substructure to highlight on molecule.

`utils.rdkit_easy.setup_notebook()`

Set up current notebook for displaying plots and Bokeh output using full width of window

`utils.rdkit_easy.show_df(df)`

Convenience function to display a pandas DataFrame in the current notebook window with HTML images rendered in table cells.

**Args:**

df (pd.DataFrame): Data frame to display.

**Returns:**

None

`utils.rdkit_easy.show_html(html)`

Convenience function to display an HTML image specified by image tag 'html'.

**Args:**

html (str): HTML image tag to render.

**Returns:**

None

## utils.split\_response\_dist\_plots module

Module to plot distributions of response values in each subset of a dataset generated by a split

`utils.split_response_dist_plots.get_split_labeled_dataset(params)`

Add a column to a dataset labeling the split subset for each row. Given a dataset and split parameters (including `split_uuid`) referenced in *params*, returns a data frame containing the dataset with an extra 'split\_subset' column indicating the subset each data point belongs to. For standard 3-way splits, the labels will be 'train', 'valid' and 'test'. For a k-fold CV split, the labels will be 'fold\_0' through 'fold\_<k-1>' and 'test'.

### Args:

*params* (argparse.Namespace or dict): Structure containing dataset and split parameters. The following parameters are required, if not set to default values:

- dataset\_key
- split\_uuid
- split\_strategy
- splitter
- split\_valid\_frac
- split\_test\_frac
- num\_folds
- smiles\_col
- response\_cols

### Returns:

A tuple (dset\_df, split\_label):

- dset\_df (DataFrame): The dataset specified by *params.dataset\_key*, with additional column *split\_subset*.
- split\_label (str): A short description of the split, useful for plot labeling.

`utils.split_response_dist_plots.plot_split_subset_response_distrs(params)`

Plot the distributions of the response variable(s) in each split subset of a dataset.

### Args:

*params* (argparse.Namespace or dict): Structure containing dataset and split parameters. The following parameters are required, if not set to default values:

- dataset\_key
- split\_uuid
- split\_strategy
- splitter
- split\_valid\_frac
- split\_test\_frac
- num\_folds
- smiles\_col
- response\_cols

**Returns:**

None

## utils.struct\_utils module

Functions to manipulate and convert between various representations of chemical structures: SMILES, InChi and RDKit Mol objects. Many of these functions (those with a 'workers' argument) accept either a single SMILES or InChi string or a list of strings as their first argument, and return a value with the same datatype. If a list is passed and the 'workers' argument is > 1, the calculation is parallelized across multiple threads; this can save significant time when operating on thousands of molecules.

`utils.struct_utils.base_mol_from_inchi(inchi_str, useIsomericSmiles=True, removeCharges=False)`

Generate a standardized RDKit Mol object for the largest fragment of the molecule specified by InChi string `inchi_str`. Replace any rare isotopes with the most common ones for each element. If `removeCharges` is True, add hydrogens as needed to eliminate charges.

**Args:**

`inchi_str` (str): InChi string representing molecule.

`useIsomericSmiles` (bool): Whether to retain stereochemistry information in the generated string.

`removeCharges` (bool): If true, add or remove hydrogens to produce uncharged molecules.

**Returns:**

str: Standardized salt-stripped SMILES string.

`utils.struct_utils.base_mol_from_smiles(orig_smiles, useIsomericSmiles=True, removeCharges=False)`

Generate a standardized RDKit Mol object for the largest fragment of the molecule specified by `orig_smiles`. Replace any rare isotopes with the most common ones for each element. If `removeCharges` is True, add hydrogens as needed to eliminate charges.

**Args:**

`orig_smiles` (str): SMILES string to standardize.

`useIsomericSmiles` (bool): Whether to retain stereochemistry information in the generated string.

`removeCharges` (bool): If true, add or remove hydrogens to produce uncharged molecules.

**Returns:**

str: Standardized salt-stripped SMILES string.

`utils.struct_utils.base_smiles_from_inchi(inchi_str, useIsomericSmiles=True, removeCharges=False, workers=1)`

Generate standardized salt-stripped SMILES strings for the largest fragments of each molecule represented by InChi string(s) `inchi_str`. Replaces any rare isotopes with the most common ones for each element.

**Args:**

`inchi_str` (list or str): List of InChi strings to convert.

`useIsomericSmiles` (bool): Whether to retain stereochemistry information in the generated strings.

`removeCharges` (bool): If true, add or remove hydrogens to produce uncharged molecules.

`workers` (int): Number of parallel threads to use for calculation.

**Returns:**

list or str: Standardized SMILES strings.

`utils.struct_utils.base_smiles_from_smiles(orig_smiles, useIsomericSmiles=True, removeCharges=False, useCanonicalTautomers=False, workers=1)`

Generate standardized SMILES strings for the largest fragments of each molecule specified by `orig_smiles`. Strips salt groups and replaces any rare isotopes with the most common ones for each element.

**Args:**

- `orig_smiles` (list or str): List of SMILES strings to canonicalize.
- `useIsomericSmiles` (bool): Whether to retain stereochemistry information in the generated strings.
- `removeCharges` (bool): If true, add or remove hydrogens to produce uncharged molecules.
- `useCanonicalTautomers` (bool): Whether to convert the generated SMILES to their canonical tautomers. Defaults to False for backward compatibility.
- `workers` (int): Number of parallel threads to use for calculation.

**Returns:**

list or str: Canonicalized SMILES strings.

`utils.struct_utils.canonical_tautomers_from_smiles(smiles)`

Returns SMILES strings for the canonical tautomers of a SMILES string or list of SMILES strings

**Args:**

`smiles` (list or str): List of SMILES strings.

**Returns:**

(list of str) : List of SMILES strings for the canonical tautomers.

`utils.struct_utils.draw_structure(smiles_str, image_path, image_size=500)`

Draw structure for the compound with the given SMILES string as a PNG file.

Note that there are more flexible functions for drawing structures in the `rdkit_easy` module. This function is only retained for backward compatibility.

**Args:**

- `smiles_str` (str): SMILES representation of compound.
- `image_path` (str): Filepath for image file to be generated.
- `image_size` (int): Width of square bounding box for image.

**Returns:**

None.

`utils.struct_utils.fix_moe_smiles(smiles)`

Correct the SMILES strings generated by MOE to standardize the representation of protonated atoms, so that RDKit can read them.

**Args:**

`smiles` (str): SMILES string.

**Returns:**

str: The corrected SMILES string.

`utils.struct_utils.get_rdkit_smiles(orig_smiles, useIsomericSmiles=True)`

Given a SMILES string, regenerate a “canonical” SMILES string for the same molecule using the implementation in RDKit.

**Args:**

- `orig_smiles` (str): SMILES string to canonicalize.
- `useIsomericSmiles` (bool): Whether to retain stereochemistry information in the generated string.

**Returns:**

str: Canonicalized SMILES string.

`utils.struct_utils.kekulize_smiles(orig_smiles, useIsomericSmiles=True, workers=1)`

Generate Kekulized SMILES strings for the molecules specified by `orig_smiles`. Kekulized SMILES strings are ones in which aromatic rings are represented by uppercase letters with alternating single and double bonds, rather than lowercase letters; they are needed by some external applications.

**Args:**

`orig_smiles` (list or str): List of SMILES strings to Kekulize.

`useIsomericSmiles` (bool): Whether to retain stereochemistry information in the generated strings.

`workers` (int): Number of parallel threads to use for calculation.

**Returns:**

list or str: Kekulized SMILES strings.

`utils.struct_utils.mol_wt_from_smiles(smiles, workers=1)`

Calculate molecular weights for molecules represented by SMILES strings.

**Args:**

`smiles` (list or str): List of SMILES strings.

`workers` (int): Number of parallel threads to use for calculations.

**Returns:**

list or float: Molecular weights. NaN is returned for SMILES strings that could not be read by RDKit.

`utils.struct_utils.mols_from_smiles(orig_smiles, workers=1)`

Parallel function to create RDKit Mol objects for a list of SMILES strings. If `orig_smiles` is a list and `workers` is  $> 1$ , spawn ‘workers’ threads to convert input SMILES strings to Mol objects.

**Args:**

`orig_smiles` (list or str): List of SMILES strings to convert to Mol objects.

`workers` (int): Number of parallel threads to use for calculation.

**Returns:**

list of `rdkit.Chem.Mol`: RDKit objects representing molecules.

`utils.struct_utils.rdkit_smiles_from_smiles(orig_smiles, useIsomericSmiles=True, useCanonicalTautomers=False, workers=1)`

Parallel version of `get_rdkit_smiles`. If `orig_smiles` is a list and `workers` is  $> 1$ , spawn ‘workers’ threads to convert input SMILES strings to standardized RDKit format.

**Args:**

`orig_smiles` (list or str): List of SMILES strings to canonicalize.

`useIsomericSmiles` (bool): Whether to retain stereochemistry information in the generated strings.

`useCanonicalTautomers` (bool): Whether to convert the generated SMILES to their canonical tautomers. Defaults to False for backward compatibility.

`workers` (int): Number of parallel threads to use for calculation.

**Returns:**

list or str: Canonicalized SMILES strings.

`utils.struct_utils.smiles_to_inchi_key(smiles)`

Generates an InChI key from a SMILES string. Note that an InChI key is different from an InChI *string*; it can be used as a unique identifier, but doesn’t hold the information needed to reconstruct a molecule.



**Args:**

smiles (str): SMILES string.

**Returns:**

str: An InChI key. Returns None if RDKit cannot convert the SMILES string to an RDKit Mol object.

**Module contents**



## USEFUL LINKS

- [ATOM Data-Driven Modeling Pipeline on GitHub](#)
- [Pipeline parameters \(options\)](#)
- [Library documentation](#)



## PYTHON MODULE INDEX

### p

- [pipeline](#), 62
- [pipeline.chem\\_diversity](#), 13
- [pipeline.compare\\_models](#), 15
- [pipeline.dist\\_metrics](#), 21
- [pipeline.diversity\\_plots](#), 21
- [pipeline.feature\\_importance](#), 23
- [pipeline.hyper\\_perf\\_plots](#), 25
- [pipeline.model\\_pipeline](#), 27
- [pipeline.model\\_tracker](#), 34
- [pipeline.parameter\\_parser](#), 36
- [pipeline.perf\\_data](#), 43
- [pipeline.perf\\_plots](#), 57
- [pipeline.predict\\_from\\_model](#), 59

### U

- [utils](#), 109
- [utils.compare\\_splits\\_plots](#), 62
- [utils.curate\\_data](#), 63
- [utils.data\\_curation\\_functions](#), 69
- [utils.datastore\\_functions](#), 80
- [utils.hyperparam\\_search\\_wrapper](#), 89
- [utils.many\\_to\\_one](#), 97
- [utils.model\\_file\\_reader](#), 97
- [utils.model\\_retrain](#), 99
- [utils.model\\_version\\_utils](#), 100
- [utils.pubchem\\_utils](#), 101
- [utils.rdkit\\_easy](#), 102
- [utils.split\\_response\\_dist\\_plots](#), 105
- [utils.struct\\_utils](#), 106



## INDEX

### A

`accumulate()` (`pipeline.perf_data.EpochManager` method), 44  
`accumulate_preds()` (`pipeline.perf_data.ClassificationPerfData` method), 43  
`accumulate_preds()` (`pipeline.perf_data.HybridPerfData` method), 47  
`accumulate_preds()` (`pipeline.perf_data.KFoldClassificationPerfData` method), 47  
`accumulate_preds()` (`pipeline.perf_data.KFoldRegressionPerfData` method), 49  
`accumulate_preds()` (`pipeline.perf_data.PerfData` method), 50  
`accumulate_preds()` (`pipeline.perf_data.RegressionPerfData` method), 51  
`accumulate_preds()` (`pipeline.perf_data.SimpleClassificationPerfData` method), 52  
`accumulate_preds()` (`pipeline.perf_data.SimpleHybridPerfData` method), 54  
`accumulate_preds()` (`pipeline.perf_data.SimpleRegressionPerfData` method), 55  
`add_classification_column()` (in module `utils.curate_data`), 63  
`add_mol_column()` (in module `utils.rdkit_easy`), 102  
`add_to_parser()` (`pipeline.parameter_parser.AutoArgumentAdder` method), 37  
`aggregate_assay_data()` (in module `utils.curate_data`), 63  
`all_auto_arguments()` (in module `pipeline.parameter_parser`), 38  
`all_auto_float_lists()` (in module `pipeline.parameter_parser`), 38  
`all_auto_int_lists()` (in module `pipeline.parameter_parser`), 38  
`all_auto_lists()` (in module `pipeline.parameter_parser`), 38  
`all_prefixed_names()` (`pipeline.parameter_parser.AutoArgumentAdder` method), 37  
`already_run()` (`utils.hyperparam_search_wrapper.HyperparameterSearch` method), 91  
`assemble_layers()` (`utils.hyperparam_search_wrapper.HyperparameterSearch` method), 91  
`atom_curation()` (in module `utils.data_curation_functions`), 69  
`atom_curation_escape()` (in module `utils.data_curation_functions`), 69  
`AutoArgumentAdder` (class in module `pipeline.parameter_parser`), 36  
`average_and_remove_duplicates()` (in module `utils.curate_data`), 64

### B

`base_feature_importance()` (in module `pipeline.feature_importance`), 23  
`base_mol_from_inchi()` (in module `utils.struct_utils`), 106  
`base_mol_from_smiles()` (in module `utils.struct_utils`), 106  
`base_smiles_from_inchi()` (in module `utils.struct_utils`), 106  
`base_smiles_from_smiles()` (in module `utils.struct_utils`), 106  
`build_dataset_name()` (in module `pipeline.model_pipeline`), 31  
`build_hyperopt_search_domain()` (in module `utils.hyperparam_search_wrapper`), 94  
`build_jobs()` (`utils.hyperparam_search_wrapper.HyperparameterSearch` method), 91  
`build_search()` (in module `utils.hyperparam_search_wrapper`), 94  
`build_tarball_name()` (in module `pipeline.model_pipeline`), 31  
`bulk_export_kv_for_files()` (in module `utils.datastore_functions`), 80  
`bulk_update_kv()` (in module `utils.datastore_functions`), 80

### C

`calc_AD_kmean_dist()` (in module `pipeline.model_pipeline`), 31  
`calc_AD_kmean_local_density()` (in module `pipeline.model_pipeline`), 31

[calc\\_dist\\_diskdataset\(\)](#) (in module [pipeline.chem\\_diversity](#)), 13  
[calc\\_dist\\_feat\\_array\(\)](#) (in module [pipeline.chem\\_diversity](#)), 13  
[calc\\_dist\\_smiles\(\)](#) (in module [pipeline.chem\\_diversity](#)), 14  
[calc\\_summary\(\)](#) (in module [pipeline.chem\\_diversity](#)), 14  
[calc\\_train\\_dset\\_pair\\_dis\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), 27  
[calculate\\_descriptors\(\)](#) (in module [utils.rdkit\\_easy](#)), 102  
[canonical\\_tautomers\\_from\\_smiles\(\)](#) (in module [utils.struct\\_utils](#)), 107  
[check\\_key\\_val\(\)](#) (in module [utils.datastore\\_functions](#)), 80  
[check\\_version\\_compatible\(\)](#) (in module [utils.model\\_version\\_utils](#)), 100  
[ClassificationPerfData](#) (class in [pipeline.perf\\_data](#)), 43  
[cluster\\_dataframe\(\)](#) (in module [utils.rdkit\\_easy](#)), 102  
[cluster\\_fingerprints\(\)](#) (in module [utils.rdkit\\_easy](#)), 102  
[cluster\\_permutation\\_importance\(\)](#) (in module [pipeline.feature\\_importance](#)), 23  
[compute\(\)](#) ([pipeline.perf\\_data.EpochManager](#) method), 44  
[compute\(\)](#) ([pipeline.perf\\_data.EpochManagerKFold](#) method), 46  
[compute\\_negative\\_log\\_responses\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 69  
[compute\\_perf\\_metrics\(\)](#) ([pipeline.perf\\_data.HybridPerfData](#) method), 47  
[compute\\_perf\\_metrics\(\)](#) ([pipeline.perf\\_data.KFoldClassificationPerfData](#) method), 48  
[compute\\_perf\\_metrics\(\)](#) ([pipeline.perf\\_data.KFoldRegressionPerfData](#) method), 49  
[compute\\_perf\\_metrics\(\)](#) ([pipeline.perf\\_data.PerfData](#) method), 51  
[compute\\_perf\\_metrics\(\)](#) ([pipeline.perf\\_data.RegressionPerfData](#) method), 51  
[compute\\_perf\\_metrics\(\)](#) ([pipeline.perf\\_data.SimpleClassificationPerfData](#) method), 52  
[compute\\_perf\\_metrics\(\)](#) ([pipeline.perf\\_data.SimpleHybridPerfData](#) method), 54  
[compute\\_perf\\_metrics\(\)](#) ([pipeline.perf\\_data.SimpleRegressionPerfData](#) method), 55  
[config\\_client\(\)](#) (in module [utils.datastore\\_functions](#)), 81  
[convert\\_IC50\\_to\\_pIC50\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 70  
[convert\\_metadata\(\)](#) (in module [pipeline.model\\_tracker](#)), 34  
[copy\\_best\\_filesystem\\_models\(\)](#) (in module [pipeline.compare\\_models](#)), 15  
[copy\\_datasets\\_to\\_bucket\(\)](#) (in module [utils.datastore\\_functions](#)), 81  
[create\\_model\\_metadata\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), 27  
[create\\_new\\_rows\\_for\\_extra\\_results\(\)](#) (in module [utils.curate\\_data](#)), 64  
[create\\_perf\\_data\(\)](#) (in module [pipeline.perf\\_data](#)), 56  
[create\\_prediction\\_metadata\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), 27  
[create\\_prediction\\_pipeline\(\)](#) (in module [pipeline.model\\_pipeline](#)), 31  
[create\\_prediction\\_pipeline\\_from\\_file\(\)](#) (in module [pipeline.model\\_pipeline](#)), 32

## D

[dataset\\_key\\_exists\(\)](#) (in module [utils.datastore\\_functions](#)), 81  
[DatastoreInsertionException](#), 34  
[del\\_ignored\\_params\(\)](#) (in module [pipeline.compare\\_models](#)), 16  
[dict\\_to\\_list\(\)](#) (in module [pipeline.parameter\\_parser](#)), 38  
[display\\_feature\\_clusters\(\)](#) (in module [pipeline.feature\\_importance](#)), 24  
[dist\\_hist\\_plot\(\)](#) ([utils.compare\\_splits\\_plots.SplitStats](#) method), 62  
[dist\\_hist\\_train\\_v\\_test\\_plot\(\)](#) ([utils.compare\\_splits\\_plots.SplitStats](#) method), 62  
[dist\\_hist\\_train\\_v\\_valid\\_plot\(\)](#) ([utils.compare\\_splits\\_plots.SplitStats](#) method), 62  
[diversity\\_plots\(\)](#) (in module [pipeline.diversity\\_plots](#)), 21  
[down\\_select\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 70  
[download\\_activitytype\(\)](#) (in module [utils.pubchem\\_utils](#)), 101  
[download\\_bioactivity\\_assay\(\)](#) (in module [utils.pubchem\\_utils](#)), 101  
[download\\_dose\\_response\\_from\\_bioactivity\(\)](#) (in module [utils.pubchem\\_utils](#)), 101



download\_SID\_from\_bioactivity\_assay() (in module *utils.pubchem\_utils*), 101  
download\_smiles() (in module *utils.pubchem\_utils*), 101  
draw\_structure() (in module *utils.struct\_utils*), 107

## E

ensemble\_predict() (in module *pipeline.model\_pipeline*), 32  
EpochManager (class in *pipeline.perf\_data*), 44  
EpochManagerKfold (class in *pipeline.perf\_data*), 46  
exclude\_organometallics() (in module *utils.data\_curation\_functions*), 70  
export\_model() (in module *pipeline.model\_tracker*), 34  
extract\_collection\_perf\_metrics() (in module *pipeline.compare\_models*), 16  
extract\_datastore\_model\_tarball() (in module *pipeline.model\_tracker*), 34  
extract\_featurizer\_params() (in module *pipeline.parameter\_parser*), 38  
extract\_model\_and\_feature\_parameters() (in module *pipeline.compare\_models*), 16  
extract\_model\_params() (in module *pipeline.parameter\_parser*), 39  
extract\_params() (*pipeline.parameter\_parser.AutoArgumentParser* method), 37

## F

filter\_datasets\_interactive() (in module *utils.datastore\_functions*), 81  
filter\_dtc\_data() (in module *utils.data\_curation\_functions*), 70  
filter\_in\_by\_column\_values() (in module *utils.curate\_data*), 65  
filter\_in\_out\_by\_column\_values() (in module *utils.curate\_data*), 65  
filter\_jobs() (*utils.hyperparam\_search\_wrapper.HyperparameterSearch* method), 91  
filter\_out\_by\_column\_values() (in module *utils.curate\_data*), 65  
filter\_out\_comments() (in module *utils.curate\_data*), 66  
fix\_moe\_smiles() (in module *utils.struct\_utils*), 107  
flatten\_dict() (in module *pipeline.parameter\_parser*), 39  
freq\_table() (in module *utils.curate\_data*), 66

## G

gen\_maestro\_command() (in module *utils.hyperparam\_search\_wrapper*), 95  
generate\_assay\_list() (*utils.hyperparam\_search\_wrapper.GeometricSearch* method), 89

generate\_assay\_list() (*utils.hyperparam\_search\_wrapper.GridSearch* method), 89  
generate\_assay\_list() (*utils.hyperparam\_search\_wrapper.HyperparameterSearch* method), 91  
generate\_assay\_list() (*utils.hyperparam\_search\_wrapper.RandomSearch* method), 93  
generate\_assay\_list() (*utils.hyperparam\_search\_wrapper.UserSpecifiedSearch* method), 94  
generate\_combo() (*utils.hyperparam\_search\_wrapper.GeometricSearch* method), 89  
generate\_combo() (*utils.hyperparam\_search\_wrapper.GridSearch* method), 89  
generate\_combo() (*utils.hyperparam\_search\_wrapper.HyperparameterSearch* method), 91  
generate\_combo() (*utils.hyperparam\_search\_wrapper.RandomSearch* method), 93  
generate\_combo() (*utils.hyperparam\_search\_wrapper.UserSpecifiedSearch* method), 94  
generate\_combos() (*utils.hyperparam\_search\_wrapper.HyperparameterSearch* method), 91  
generate\_maestro\_commands() (*utils.hyperparam\_search\_wrapper.HyperparameterSearch* method), 91  
generate\_param\_combos() (*utils.hyperparam\_search\_wrapper.GeometricSearch* method), 89  
generate\_param\_combos() (*utils.hyperparam\_search\_wrapper.GridSearch* method), 89  
generate\_param\_combos() (*utils.hyperparam\_search\_wrapper.HyperparameterSearch* method), 91  
generate\_param\_combos() (*utils.hyperparam\_search\_wrapper.RandomSearch* method), 94  
generate\_param\_combos() (*utils.hyperparam\_search\_wrapper.UserSpecifiedSearch* method), 94  
generate\_searches() (*utils.hyperparam\_search\_wrapper.HyperparameterSearch* method), 92  
generate\_split\_shortlist() (*utils.hyperparam\_search\_wrapper.HyperparameterSearch* method), 92  
generate\_split\_shortlist\_file() (*utils.hyperparam\_search\_wrapper.HyperparameterSearch* method), 92  
GeometricSearch (class in *utils.hyperparam\_search\_wrapper*), 89  
get\_ampl\_version() (in module

`utils.model_version_utils`), 100  
`get_ampl_version_from_dir()` (in module `utils.model_version_utils`), 100  
`get_ampl_version_from_json()` (in module `utils.model_version_utils`), 100  
`get_ampl_version_from_model()` (in module `utils.model_version_utils`), 100  
`get_best_models_info()` (in module `pipeline.compare_models`), 16  
`get_best_perf_table()` (in module `pipeline.compare_models`), 17  
`get_collection_datasets()` (in module `pipeline.compare_models`), 17  
`get_dataset_key()` (`utils.model_file_reader.ModelFileReader` method), 97  
`get_dataset_metadata()` (`utils.hyperparam_search_wrapper.HyperparameterSearch` method), 92  
`get_dataset_models()` (in module `pipeline.compare_models`), 17  
`get_descriptor_type()` (`utils.model_file_reader.ModelFileReader` method), 97  
`get_featurizer()` (`utils.model_file_reader.ModelFileReader` method), 97  
`get_filesystem_models()` (in module `pipeline.compare_models`), 18  
`get_filesystem_perf_results()` (in module `pipeline.compare_models`), 18  
`get_full_metadata()` (in module `pipeline.model_tracker`), 35  
`get_full_metadata_by_uuid()` (in module `pipeline.model_tracker`), 35  
`get_id_col()` (`utils.model_file_reader.ModelFileReader` method), 97  
`get_key_val()` (in module `utils.datastore_functions`), 82  
`get_keyval()` (in module `utils.datastore_functions`), 82  
`get_list_args()` (`pipeline.parameter_parser.AutoArgumentAdder` method), 37  
`get_list_float_args()` (`pipeline.parameter_parser.AutoArgumentAdder` method), 37  
`get_list_int_args()` (`pipeline.parameter_parser.AutoArgumentAdder` method), 37  
`get_major_version()` (in module `utils.model_version_utils`), 100  
`get_metadata_by_uuid()` (in module `pipeline.model_tracker`), 35  
`get_metrics()` (`pipeline.model_pipeline.ModelPipeline` method), 28  
`get_model_collection_by_uuid()` (in module `pipeline.model_tracker`), 35  
`get_model_info()` (`utils.model_file_reader.ModelFileReader` method), 97  
`get_model_parameters()` (`utils.model_file_reader.ModelFileReader` method), 98  
`get_model_training_data_by_uuid()` (in module `pipeline.model_tracker`), 36  
`get_model_type()` (`utils.model_file_reader.ModelFileReader` method), 98  
`get_model_uuid()` (`utils.model_file_reader.ModelFileReader` method), 98  
`get_multiple_models_metadata()` (in module `utils.model_file_reader`), 98  
`get_multitask_perf_from_files()` (in module `pipeline.compare_models`), 18  
`get_multitask_perf_from_files_new()` (in module `pipeline.compare_models`), 18  
`get_multitask_perf_from_tracker()` (in module `pipeline.compare_models`), 18  
`get_num_params()` (in module `utils.hyperparam_search_wrapper`), 95  
`get_parser()` (in module `pipeline.parameter_parser`), 39  
`get_pred_values()` (`pipeline.perf_data.ClassificationPerfData` method), 43  
`get_pred_values()` (`pipeline.perf_data.HybridPerfData` method), 47  
`get_pred_values()` (`pipeline.perf_data.KFoldClassificationPerfData` method), 48  
`get_pred_values()` (`pipeline.perf_data.KFoldRegressionPerfData` method), 50  
`get_pred_values()` (`pipeline.perf_data.PerfData` method), 51  
`get_pred_values()` (`pipeline.perf_data.RegressionPerfData` method), 51  
`get_pred_values()` (`pipeline.perf_data.SimpleClassificationPerfData` method), 53  
`get_pred_values()` (`pipeline.perf_data.SimpleHybridPerfData` method), 54  
`get_pred_values()` (`pipeline.perf_data.SimpleRegressionPerfData` method), 55  
`get_prediction_results()` (`pipeline.perf_data.ClassificationPerfData` method), 43  
`get_prediction_results()` (`pipeline.perf_data.HybridPerfData` method), 47  
`get_prediction_results()` (`pipeline.perf_data.PerfData` method), 51  
`get_prediction_results()` (`pipeline.perf_data.RegressionPerfData` method), 51  
`get_rdkit_smiles()` (in module `utils.struct_utils`), 107  
`get_rdkit_smiles_parent()` (in module

[utils.curate\\_data](#)), 66  
[get\\_real\\_values\(\)](#) ([pipeline.perf\\_data.KFoldClassificationPerfData](#) method), 48  
[get\\_real\\_values\(\)](#) ([pipeline.perf\\_data.KFoldRegressionPerfData](#) method), 50  
[get\\_real\\_values\(\)](#) ([pipeline.perf\\_data.PerfData](#) method), 51  
[get\\_real\\_values\(\)](#) ([pipeline.perf\\_data.SimpleClassificationPerfData](#) method), 53  
[get\\_real\\_values\(\)](#) ([pipeline.perf\\_data.SimpleHybridPerfData](#) method), 54  
[get\\_real\\_values\(\)](#) ([pipeline.perf\\_data.SimpleRegressionPerfData](#) method), 55  
[get\\_response\\_cols\(\)](#) ([utils.model\\_file\\_reader.ModelFileReader](#) method), 98  
[get\\_score\\_types\(\)](#) (in module [pipeline.hyper\\_perf\\_plots](#)), 25  
[get\\_shortlist\\_df\(\)](#) ([utils.hyperparam\\_search\\_wrapper.HyperparameterSearch](#) method), 92  
[get\\_smiles\\_4dtc\\_data\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 70  
[get\\_smiles\\_col\(\)](#) ([utils.model\\_file\\_reader.ModelFileReader](#) method), 98  
[get\\_smiles\\_dtc\\_data\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 71  
[get\\_smiles\\_escape\\_data\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 71  
[get\\_split\\_csv\(\)](#) ([utils.model\\_file\\_reader.ModelFileReader](#) method), 98  
[get\\_split\\_labeled\\_dataset\(\)](#) (in module [utils.split\\_response\\_dist\\_plots](#)), 105  
[get\\_split\\_strategy\(\)](#) ([utils.model\\_file\\_reader.ModelFileReader](#) method), 98  
[get\\_split\\_uuid\(\)](#) ([utils.model\\_file\\_reader.ModelFileReader](#) method), 98  
[get\\_splitter\(\)](#) ([utils.model\\_file\\_reader.ModelFileReader](#) method), 98  
[get\\_splitting\\_parameters\(\)](#) ([utils.model\\_file\\_reader.ModelFileReader](#) method), 98  
[get\\_summary\\_metadata\\_table\(\)](#) (in module [pipeline.compare\\_models](#)), 19  
[get\\_summary\\_perf\\_tables\(\)](#) (in module [pipeline.compare\\_models](#)), 19  
[get\\_tarball\\_perf\\_table\(\)](#) (in module [pipeline.compare\\_models](#)), 19  
[get\\_training\\_dataset\(\)](#) ([utils.model\\_file\\_reader.ModelFileReader](#) method), 98  
[get\\_training\\_datasets\(\)](#) (in module [pipeline.compare\\_models](#)), 20  
[get\\_training\\_perf\\_table\(\)](#) (in module [pipeline.compare\\_models](#)), 20  
[get\\_weights\(\)](#) ([pipeline.perf\\_data.KFoldClassificationPerfData](#) method), 98  
[get\\_weights\(\)](#) ([pipeline.perf\\_data.KFoldRegressionPerfData](#) method), 49  
[get\\_weights\(\)](#) ([pipeline.perf\\_data.PerfData](#) method), 50  
[get\\_weights\(\)](#) ([pipeline.perf\\_data.SimpleClassificationPerfData](#) method), 53  
[get\\_weights\(\)](#) ([pipeline.perf\\_data.SimpleHybridPerfData](#) method), 54  
[get\\_weights\(\)](#) ([pipeline.perf\\_data.SimpleRegressionPerfData](#) method), 56  
[GridSearch](#) (class in [utils.hyperparam\\_search\\_wrapper](#)), 89  

## H

[HyperparameterSearch](#)  
[has\\_nans\(\)](#) (in module [utils.many\\_to\\_one](#)), 97  
[HybridPerfData](#) (class in [pipeline.perf\\_data](#)), 46  
[HyperOptSearch](#) (class in [utils.hyperparam\\_search\\_wrapper](#)), 90  
[HyperparameterSearch](#) (class in [utils.hyperparam\\_search\\_wrapper](#)), 91  

## I

[ic50topic50\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 72  
[initialize\\_model\\_tracker\(\)](#) (in module [utils.datastore\\_functions](#)), 82  
[is\\_list\(\)](#) (in module [pipeline.parameter\\_parser](#)), 39  
[is\\_list\\_float\(\)](#) (in module [pipeline.parameter\\_parser](#)), 39  
[is\\_list\\_int\(\)](#) (in module [pipeline.parameter\\_parser](#)), 40  
[is\\_organometallic\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 72  
[is\\_primitive\\_type\(\)](#) (in module [pipeline.parameter\\_parser](#)), 40  

## K

[kekulize\\_smiles\(\)](#) (in module [utils.struct\\_utils](#)), 108  
[key\\_exists\(\)](#) (in module [utils.datastore\\_functions](#)), 82  
[KFoldClassificationPerfData](#) (class in [pipeline.perf\\_data](#)), 47  
[KFoldRegressionPerfData](#) (class in [pipeline.perf\\_data](#)), 49  

## L

[labeled\\_freq\\_table\(\)](#) (in module [utils.curate\\_data](#)), 66  
[list\\_defaults\(\)](#) (in module [pipeline.parameter\\_parser](#)), 40

`list_key_values()` (in module `utils.datastore_functions`), 83

`load_featurize_data()` (`pipeline.model_pipeline.ModelPipeline` method), 28

`load_from_tracker()` (in module `pipeline.model_pipeline`), 33

## M

`main()` (in module `pipeline.model_pipeline`), 33

`main()` (in module `pipeline.parameter_parser`), 40

`main()` (in module `utils.hyperparam_search_wrapper`), 95

`main()` (in module `utils.model_file_reader`), 98

`main()` (in module `utils.model_retrain`), 99

`main()` (in module `utils.model_version_utils`), 100

`make_all_plots()` (`utils.compare_splits_plots.SplitStats` method), 62

`make_dataset_key_absolute()` (in module `pipeline.parameter_parser`), 40

`many_to_one()` (in module `utils.many_to_one`), 97

`many_to_one_df()` (in module `utils.many_to_one`), 97

`ManyToOneException`, 97

`matching_atoms_and_bonds()` (in module `utils.rdkit_easy`), 102

`mcs()` (in module `pipeline.dist_metrics`), 21

`mle_censored_mean()` (in module `utils.curate_data`), 67

`MLMTClientInstantiationException`, 34

`model_choice_score()` (`pipeline.perf_data.ClassificationPerfData` method), 43

`model_choice_score()` (`pipeline.perf_data.HybridPerfData` method), 47

`model_choice_score()` (`pipeline.perf_data.RegressionPerfData` method), 51

`ModelFileReader` (class in `utils.model_file_reader`), 97

`ModelPipeline` (class in `pipeline.model_pipeline`), 27

module

- `pipeline`, 62
- `pipeline.chem_diversity`, 13
- `pipeline.compare_models`, 15
- `pipeline.dist_metrics`, 21
- `pipeline.diversity_plots`, 21
- `pipeline.feature_importance`, 23
- `pipeline.hyper_perf_plots`, 25
- `pipeline.model_pipeline`, 27
- `pipeline.model_tracker`, 34
- `pipeline.parameter_parser`, 36
- `pipeline.perf_data`, 43
- `pipeline.perf_plots`, 57
- `pipeline.predict_from_model`, 59

- `utils`, 109
- `utils.compare_splits_plots`, 62
- `utils.curate_data`, 63
- `utils.data_curation_functions`, 69
- `utils.datastore_functions`, 80
- `utils.hyperparam_search_wrapper`, 89
- `utils.many_to_one`, 97
- `utils.model_file_reader`, 97
- `utils.model_retrain`, 99
- `utils.model_version_utils`, 100
- `utils.pubchem_utils`, 101
- `utils.rdkit_easy`, 102
- `utils.split_response_dist_plots`, 105
- `utils.struct_utils`, 106

`mol_to_html()` (in module `utils.rdkit_easy`), 103

`mol_to_pil()` (in module `utils.rdkit_easy`), 103

`mol_to_svg()` (in module `utils.rdkit_easy`), 103

`mol_wt_from_smiles()` (in module `utils.struct_utils`), 108

`mols_from_smiles()` (in module `utils.struct_utils`), 108

## N

`NANCompoundIDException`, 97

`NANSMILESEException`, 97

`negative_predictive_value()` (in module `pipeline.perf_data`), 56

`no_nan_ids_or_smiles()` (in module `utils.many_to_one`), 97

`num_trainable_parameters_from_file()` (in module `pipeline.compare_models`), 20

## O

`on_new_best_valid()` (`pipeline.perf_data.EpochManager` method), 45

## P

`parse_args()` (in module `utils.compare_splits_plots`), 62

`parse_command_line()` (in module `pipeline.parameter_parser`), 40

`parse_config_file()` (in module `pipeline.parameter_parser`), 41

`parse_namespace()` (in module `pipeline.parameter_parser`), 41

`parse_params()` (in module `utils.hyperparam_search_wrapper`), 95

`PerfData` (class in `pipeline.perf_data`), 50

`permute_NNlayer_combo_params()` (in module `utils.hyperparam_search_wrapper`), 95

`permutation_feature_importance()` (in module `pipeline.feature_importance`), 24

`pipeline`

- module, 62

[pipeline.chem\\_diversity](#) (module, [13](#))  
[pipeline.compare\\_models](#) (module, [15](#))  
[pipeline.dist\\_metrics](#) (module, [21](#))  
[pipeline.diversity\\_plots](#) (module, [21](#))  
[pipeline.feature\\_importance](#) (module, [23](#))  
[pipeline.hyper\\_perf\\_plots](#) (module, [25](#))  
[pipeline.model\\_pipeline](#) (module, [27](#))  
[pipeline.model\\_tracker](#) (module, [34](#))  
[pipeline.parameter\\_parser](#) (module, [36](#))  
[pipeline.perf\\_data](#) (module, [43](#))  
[pipeline.perf\\_plots](#) (module, [57](#))  
[pipeline.predict\\_from\\_model](#) (module, [59](#))  
[plot\\_dataset\\_dist\\_distr\(\)](#) (in module [pipeline.diversity\\_plots](#)), [22](#)  
[plot\\_feature\\_importances\(\)](#) (in module [pipeline.feature\\_importance](#)), [25](#)  
[plot\\_nn\\_perf\(\)](#) (in module [pipeline.hyper\\_perf\\_plots](#)), [25](#)  
[plot\\_perf\\_vs\\_epoch\(\)](#) (in module [pipeline.perf\\_plots](#)), [57](#)  
[plot\\_prec\\_recall\\_curve\(\)](#) (in module [pipeline.perf\\_plots](#)), [57](#)  
[plot\\_pred\\_vs\\_actual\(\)](#) (in module [pipeline.perf\\_plots](#)), [57](#)  
[plot\\_pred\\_vs\\_actual\\_from\\_df\(\)](#) (in module [pipeline.perf\\_plots](#)), [58](#)  
[plot\\_pred\\_vs\\_actual\\_from\\_file\(\)](#) (in module [pipeline.perf\\_plots](#)), [58](#)  
[plot\\_rf\\_nn\\_xg\\_perf\(\)](#) (in module [pipeline.hyper\\_perf\\_plots](#)), [26](#)  
[plot\\_rf\\_perf\(\)](#) (in module [pipeline.hyper\\_perf\\_plots](#)), [26](#)  
[plot\\_ROC\\_curve\(\)](#) (in module [pipeline.perf\\_plots](#)), [57](#)  
[plot\\_split\\_perf\(\)](#) (in module [pipeline.hyper\\_perf\\_plots](#)), [26](#)  
[plot\\_split\\_subset\\_response\\_distrs\(\)](#) (in module [utils.split\\_response\\_dist\\_plots](#)), [105](#)  
[plot\\_tani\\_dist\\_distr\(\)](#) (in module [pipeline.diversity\\_plots](#)), [22](#)  
[plot\\_train\\_valid\\_test\\_scores\(\)](#) (in module [pipeline.hyper\\_perf\\_plots](#)), [26](#)  
[plot\\_umap\\_feature\\_projections\(\)](#) (in module [pipeline.perf\\_plots](#)), [58](#)  
[plot\\_umap\\_train\\_set\\_neighbors\(\)](#) (in module [pipeline.perf\\_plots](#)), [59](#)  
[plot\\_xg\\_perf\(\)](#) (in module [pipeline.hyper\\_perf\\_plots](#)), [26](#)  
[postprocess\\_args\(\)](#) (in module [pipeline.parameter\\_parser](#)), [41](#)  
[predict\\_embedding\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), [28](#)  
[predict\\_from\\_model\\_file\(\)](#) (in module [pipeline.predict\\_from\\_model](#)), [59](#)  
[predict\\_from\\_tracker\\_model\(\)](#) (in module [pipeline.predict\\_from\\_model](#)), [60](#)  
[predict\\_full\\_dataset\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), [28](#)  
[predict\\_on\\_dataframe\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), [29](#)  
[predict\\_on\\_smiles\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), [29](#)  
[primitive\\_type\\_only\(\)](#) (in module [pipeline.parameter\\_parser](#)), [42](#)  
[print\\_stats\(\)](#) ([utils.compare\\_splits\\_plots.SplitStats](#) method), [62](#)  
[prune\\_defaults\(\)](#) (in module [pipeline.parameter\\_parser](#)), [42](#)

## R

[RandomSearch](#) (class in [utils.hyperparam\\_search\\_wrapper](#)), [93](#)  
[rdkit\\_smiles\\_from\\_smiles\(\)](#) (in module [utils.struct\\_utils](#)), [108](#)  
[reformat\\_filter\\_dict\(\)](#) (in module [utils.hyperparam\\_search\\_wrapper](#)), [96](#)  
[regenerate\\_results\(\)](#) (in module [pipeline.model\\_pipeline](#)), [33](#)  
[RegressionPerfData](#) (class in [pipeline.perf\\_data](#)), [51](#)  
[remove\\_outlier\\_replicates\(\)](#) (in module [utils.curate\\_data](#)), [67](#)  
[remove\\_unrecognized\\_arguments\(\)](#) (in module [pipeline.parameter\\_parser](#)), [42](#)  
[repeat\\_defined\\_search\(\)](#) (in module [utils.datastore\\_functions](#)), [83](#)  
[replicate\\_rmsd\(\)](#) (in module [utils.curate\\_data](#)), [67](#)  
[retrain\\_model\(\)](#) (in module [pipeline.model\\_pipeline](#)), [33](#)  
[retrieve\\_bucket\\_names\(\)](#) (in module [utils.datastore\\_functions](#)), [83](#)  
[retrieve\\_columns\\_from\\_dataset\(\)](#) (in module [utils.datastore\\_functions](#)), [83](#)



[retrieve\\_dataset\\_by\\_dataset\\_oid\(\)](#) (in module [utils.datastore\\_functions](#)), 84  
[retrieve\\_dataset\\_by\\_datasetkey\(\)](#) (in module [utils.datastore\\_functions](#)), 84  
[retrieve\\_keys\(\)](#) (in module [utils.datastore\\_functions](#)), 85  
[retrieve\\_values\\_for\\_key\(\)](#) (in module [utils.datastore\\_functions](#)), 85  
[return\\_split\\_uuid\(\)](#) ([utils.hyperparam\\_search\\_wrapper.HyperparameterSearch](#) method), 92  
[return\\_split\\_uuid\\_file\(\)](#) ([utils.hyperparam\\_search\\_wrapper.HyperparameterSearch](#) method), 93  
[rms\\_error\(\)](#) (in module [pipeline.perf\\_data](#)), 56  
[run\\_cmd\(\)](#) (in module [utils.hyperparam\\_search\\_wrapper](#)), 96  
[run\\_command\(\)](#) (in module [utils.hyperparam\\_search\\_wrapper](#)), 96  
[run\\_models\(\)](#) (in module [pipeline.model\\_pipeline](#)), 34  
[run\\_predictions\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), 29  
[run\\_search\(\)](#) ([utils.hyperparam\\_search\\_wrapper.HyperparameterSearch](#) method), 91  
[run\\_search\(\)](#) ([utils.hyperparam\\_search\\_wrapper.HyperparameterSearch](#) method), 93

## S

[save\\_figure\(\)](#) (in module [utils.compare\\_splits\\_plots](#)), 63  
[save\\_metrics\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), 30  
[save\\_model\(\)](#) (in module [pipeline.model\\_tracker](#)), 36  
[save\\_model\\_metadata\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), 30  
[save\\_model\\_tarball\(\)](#) (in module [pipeline.model\\_tracker](#)), 36  
[save\\_png\(\)](#) (in module [utils.rdkit\\_easy](#)), 104  
[save\\_svg\(\)](#) (in module [utils.rdkit\\_easy](#)), 104  
[search\\_datasets\\_by\\_key\\_value\(\)](#) (in module [utils.datastore\\_functions](#)), 85  
[search\\_files\\_interactive\(\)](#) (in module [utils.datastore\\_functions](#)), 85  
[set\\_data\\_root\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 72  
[set\\_group\\_permissions\(\)](#) (in module [utils.curate\\_data](#)), 68  
[set\\_make\\_pred\(\)](#) ([pipeline.perf\\_data.EpochManager](#) method), 45  
[setup\\_notebook\(\)](#) (in module [utils.rdkit\\_easy](#)), 104  
[should\\_stop\(\)](#) ([pipeline.perf\\_data.EpochManager](#) method), 45  
[show\\_df\(\)](#) (in module [utils.rdkit\\_easy](#)), 104  
[show\\_html\(\)](#) (in module [utils.rdkit\\_easy](#)), 104  
[SimpleClassificationPerfData](#) (class in [pipeline.perf\\_data](#)), 52  
[SimpleHybridPerfData](#) (class in [pipeline.perf\\_data](#)), 53  
[SimpleRegressionPerfData](#) (class in [pipeline.perf\\_data](#)), 54  
[smiles\\_to\\_inchi\\_key\(\)](#) (in module [utils.struct\\_utils](#)), 108  
[split\\_and\\_save\\_dataset\(\)](#) ([utils.hyperparam\\_search\\_wrapper.GeometricSearch](#) method), 90  
[split\\_and\\_save\\_dataset\(\)](#) ([utils.hyperparam\\_search\\_wrapper.GridSearch](#) method), 90  
[split\\_and\\_save\\_dataset\(\)](#) ([utils.hyperparam\\_search\\_wrapper.HyperparameterSearch](#) method), 93  
[split\\_and\\_save\\_dataset\(\)](#) ([utils.hyperparam\\_search\\_wrapper.RandomSearch](#) method), 94  
[split\\_and\\_save\\_dataset\(\)](#) ([utils.hyperparam\\_search\\_wrapper.UserSpecifiedSearch](#) method), 94  
[split\\_dataset\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), 30  
[SplitStats](#) (class in [utils.compare\\_splits\\_plots](#)), 62  
[standardize\\_relations\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 72  
[string\\_to\\_dict\(\)](#) (in module [utils.datastore\\_functions](#)), 86  
[string\\_to\\_list\(\)](#) (in module [utils.datastore\\_functions](#)), 86  
[strip\\_optional\(\)](#) (in module [pipeline.parameter\\_parser](#)), 42  
[submit\\_jobs\(\)](#) ([utils.hyperparam\\_search\\_wrapper.HyperparameterSearch](#) method), 93  
[subset\\_frac\\_plot\(\)](#) ([utils.compare\\_splits\\_plots.SplitStats](#) method), 62  
[summarize\\_data\(\)](#) (in module [utils.curate\\_data](#)), 68  
[summarize\\_datasets\(\)](#) (in module [utils.datastore\\_functions](#)), 86

## T

[tanimoto\(\)](#) (in module [pipeline.dist\\_metrics](#)), 21  
[tanimoto\\_single\(\)](#) (in module [pipeline.dist\\_metrics](#)), 21  
[to\\_str\(\)](#) (in module [pipeline.parameter\\_parser](#)), 42  
[train\\_model\(\)](#) (in module [utils.model\\_retrain](#)), 99  
[train\\_model\(\)](#) ([pipeline.model\\_pipeline.ModelPipeline](#) method), 30  
[train\\_model\\_from\\_tar\(\)](#) (in module [utils.model\\_retrain](#)), 99

[train\\_model\\_from\\_tracker\(\)](#) (in module [utils.model\\_retrain](#)), 99  
[train\\_models\\_from\\_dataset\\_keys\(\)](#) (in module [utils.model\\_retrain](#)), 99

## U

[umap\\_plot\(\)](#) ([utils.compare\\_splits\\_plots.SplitStats](#) method), 62  
[update\(\)](#) ([pipeline.perf\\_data.EpochManager](#) method), 45  
[update\\_distribution\\_kv\(\)](#) (in module [utils.datastore\\_functions](#)), 86  
[update\\_epoch\(\)](#) ([pipeline.perf\\_data.EpochManager](#) method), 46  
[update\\_kv\(\)](#) (in module [utils.datastore\\_functions](#)), 87  
[update\\_valid\(\)](#) ([pipeline.perf\\_data.EpochManager](#) method), 46  
[upload\\_df\\_dtc\\_base\\_smiles\\_all\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 72  
[upload\\_df\\_dtc\\_mleqonly\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 73  
[upload\\_df\\_dtc\\_mleqonly\\_class\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 74  
[upload\\_df\\_dtc\\_smiles\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 75  
[upload\\_df\\_dtc\\_smiles\\_regr\\_all\\_class\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 75  
[upload\\_df\\_escape\\_mleqonly\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 76  
[upload\\_df\\_escape\\_mleqonly\\_class\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 77  
[upload\\_df\\_escape\\_smiles\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 77  
[upload\\_df\\_to\\_DS\(\)](#) (in module [utils.datastore\\_functions](#)), 87  
[upload\\_distmatrix\\_to\\_DS\(\)](#) (in module [pipeline.chem\\_diversity](#)), 15  
[upload\\_file\\_dtc\\_raw\\_data\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 78  
[upload\\_file\\_dtc\\_smiles\\_regr\\_all\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 79  
[upload\\_file\\_escape\\_raw\\_data\(\)](#) (in module [utils.data\\_curation\\_functions](#)), 79  
[upload\\_file\\_to\\_DS\(\)](#) (in module [utils.datastore\\_functions](#)), 87  
[upload\\_pickle\\_to\\_DS\(\)](#) (in module [utils.datastore\\_functions](#)), 88  
[UserSpecifiedSearch](#) (class in [utils.hyperparam\\_search\\_wrapper](#)), 94  
[utils](#)  
     module, 109  
[utils.compare\\_splits\\_plots](#)  
     module, 62  
[utils.curate\\_data](#)

[utils.data\\_curation\\_functions](#)  
     module, 69  
[utils.datastore\\_functions](#)  
     module, 80  
[utils.hyperparam\\_search\\_wrapper](#)  
     module, 89  
[utils.many\\_to\\_one](#)  
     module, 97  
[utils.model\\_file\\_reader](#)  
     module, 97  
[utils.model\\_retrain](#)  
     module, 99  
[utils.model\\_version\\_utils](#)  
     module, 100  
[utils.pubchem\\_utils](#)  
     module, 101  
[utils.rdkit\\_easy](#)  
     module, 102  
[utils.split\\_response\\_dist\\_plots](#)  
     module, 105  
[utils.struct\\_utils](#)  
     module, 106

## V

[validate\\_version\(\)](#) (in module [utils.model\\_version\\_utils](#)), 100

## W

[wrapper\(\)](#) (in module [pipeline.parameter\\_parser](#)), 43

## X

[xc50topxc50\\_for\\_nm\(\)](#) (in module [utils.curate\\_data](#)), 68