ATOM Data-Driven Modeling Pipeline Documentation

Release 1.6.0

ATOM DDM Team

Feb 23, 2024

CONTENTS

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.

CHAPTER

ONE

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.

CHAPTER

TWO

BUILT WITH

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

CHAPTER

THREE

USER GUIDE

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 environemt
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_delany_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
```

CHAPTER

FOUR

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:

dists (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 wherre 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 paramters 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 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.

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 XGBoost 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

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

model_wrapper (ModelWrapper objct): 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 containting 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 wrappr

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 objct): 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 betweenthe 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/cdsw/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 datafraes containint 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

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_stds=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 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 varaibles 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 ModelWrapper 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 probabilites 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^2 scores for each task, averaged over folds, if per_task is True.
Otherwise, a float giving the R^2 score averaged over both folds and tasks.

r2_std: A numpy array of standard deviations over folds of R^2 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 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 probabilites 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 += 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*, *error_bars=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.

error_bars (bool): If true and if uncertainty estimates are included in the model predictions, draw error bars

at +- 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 thresholds[i-1] < value <= thresholds[i] (if right_inclusive is True) or thresholds[i-1] <= value < 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 containe 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 contian 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 tabluating the repeat requencies 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 occurances 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 compatibiltiy 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_4dtc_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_dtc_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_excape_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 atomsci.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 Excape 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.**ic50topic50**(*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 Excape 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 'Excape' to directores calcuated from 'data_root'

Args:
dir (str): root data directory containing folds 'dtc' and 'excape'

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 sybmols

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 atomsci.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 classiciation 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.

dtc_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 'incative' 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%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 '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_excape_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 Excape 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

utils.hyperparam_search_wrapper module

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 a 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

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

CHAPTER

FIVE

USEFUL LINKS

- ATOM Data-Driven Modeling Pipeline on GitHub
- Pipeline parameters (options)
- Library documentation

PYTHON MODULE INDEX

p

u

INDEX

H

I

K

L

M

N

O

P

R

S

T

U

V

X