chemprop

Release 1.4.0

Kyle Swanson, Kevin Yang, Wengong Jin, Lior Hirschfeld, Allison

Oct 16, 2021

CONTENTS:

Chemprop is a message passing neural network for molecular property prediction.

At its core, Chemprop contains a directed message passing neural network (D-MPNN), which was first presented in Analyzing Learned Molecular Representations for Property Prediction. The Chemprop D-MPNN shows strong molecular property prediction capabilities across a range of properties, from quantum mechanical energy to human toxicity.

Chemprop was later used in the paper A Deep Learning Approach to Antibiotic Discovery to discover promising new antibiotics by predicting the likelihood that a molecule would inhibit the growth of *E. coli*.

CHAPTER

ONE

REQUIREMENTS

For small datasets (~1000 molecules), it is possible to train models within a few minutes on a standard laptop with CPUs only. However, for larger datasets and larger chemprop models, we recommend using a GPU for significantly faster training.

To use chemprop with GPUs, you will need:

- cuda >= 8.0
- cuDNN

Chemprop is uses Python 3.6+ and all models are built with PyTorch. See *Installation* for details on how to install Chemprop and its dependencies.

CHAPTER

TWO

INSTALLATION

2.1 Overview

Chemprop can either be installed from PyPi via pip or from source (i.e., directly from the git repo). The PyPi version includes a vast majority of Chemprop functionality, but some functionality is only accessible when installed from source.

2.2 Conda

Both options require conda, so first install Miniconda from https://conda.io/miniconda.html.

Then proceed to either option below to complete the installation. Note that on machines with GPUs, you may need to manually install a GPU-enabled version of PyTorch by following the instructions here.

2.3 Option 1: Installing from PyPi

1. `conda create -n chemprop python=3.8`
2. `conda activate chemprop`
3. `conda install -c conda-forge rdkit`
4. `pip install git+https://github.com/bp-kelley/descriptastorus`
5. `pip install chemprop`

2.4 Option 2: Installing from source

1. `git clone https://github.com/chemprop/chemprop.git`
2. `cd chemprop`
3. `conda env create -f environment.yml`
4. `conda activate chemprop`
5. `pip install -e .`

2.5 Docker

Chemprop can also be installed with Docker. Docker makes it possible to isolate the Chemprop code and environment. To install and run our code in a Docker container, follow these steps:

1. `git clone https://github.com/chemprop/chemprop.git`
2. `cd chemprop`
3. Install Docker from https://docs.docker.com/install/
4. `docker build -t chemprop .`
5. `docker run -it chemprop:latest`

Note that you will need to run the latter command with nvidia-docker if you are on a GPU machine in order to be able to access the GPUs. Alternatively, with Docker 19.03+, you can specify the `--gpus` command line option instead.

In addition, you will also need to ensure that the CUDA toolkit version in the Docker image is compatible with the CUDA driver on your host machine. Newer CUDA driver versions are backward-compatible with older CUDA toolkit versions. To set a specific CUDA toolkit version, add `cudatoolkit=X.Y` to `environment.yml` before building the Docker image.

CHAPTER

THREE

TUTORIAL

3.1 Data

In order to train a model, you must provide training data containing molecules (as SMILES strings) and known target values. Targets can either be real numbers, if performing regression, or binary (i.e. 0s and 1s), if performing classification. Target values which are unknown can be left as blanks.

Our model can either train on a single target ("single tasking") or on multiple targets simultaneously ("multi-tasking").

The data file must be be a **CSV file with a header row**. For example:

```
smiles,NR-AR,NR-AR-LBD,NR-AhR,NR-Aromatase,NR-ER,NR-ER-LBD,NR-PPAR-gamma,SR-ARE,SR-ATAD5,
↪SR-HSE,SR-MMP,SR-p53
CCOc1ccc2nc(S(N)(=O)=O)sc2c1,0,0,1,,,0,0,1,0,0,0,0
CCN1C(=O)NC(c2ccccc2)C1=O,0,0,0,0,0,0,0,,0,,0,0
...
```

By default, it is assumed that the SMILES are in the first column and the targets are in the remaining columns. However, the specific columns containing the SMILES and targets can be specified using the `--smiles_column <column>` and `--target_columns <column_1> <column_2> ...` flags, respectively.

Datasets from MoleculeNet and a 450K subset of ChEMBL from http://www.bioinf.jku.at/research/lsc/index.html have been preprocessed and are available in data.tar.gz. To uncompress them, run `tar xvzf data.tar.gz`.

3.2 Training

To train a model, run:

```
chemprop_train --data_path <path> --dataset_type <type> --save_dir <dir>
```

where <path> is the path to a CSV file containing a dataset, <type> is either "classification" or "regression" depending on the type of the dataset, and <dir> is the directory where model checkpoints will be saved.

For example:

```
chemprop_train --data_path data/tox21.csv --dataset_type classification --save_dir tox21_
↪checkpoints
```

A full list of available command-line arguments can be found in *Command Line Arguments*.

If installed from source, `chemprop_train` can be replaced with `python train.py`.

Notes:

- The default metric for classification is AUC and the default metric for regression is RMSE. Other metrics may be specified with `--metric <metric>`.
- `--save_dir` may be left out if you don't want to save model checkpoints.
- `--quiet` can be added to reduce the amount of debugging information printed to the console. Both a quiet and verbose version of the logs are saved in the `save_dir`.

3.2.1 Train/Validation/Test Splits

Our code supports several methods of splitting data into train, validation, and test sets.

Random: By default, the data will be split randomly into train, validation, and test sets.

Scaffold: Alternatively, the data can be split by molecular scaffold so that the same scaffold never appears in more than one split. This can be specified by adding `--split_type scaffold_balanced`.

Separate val/test: If you have separate data files you would like to use as the validation or test set, you can specify them with `--separate_val_path <val_path>` and/or `--separate_test_path <test_path>`.

Note: By default, both random and scaffold split the data into 80% train, 10% validation, and 10% test. This can be changed with `--split_sizes <train_frac> <val_frac> <test_frac>`. For example, the default setting is `--split_sizes 0.8 0.1 0.1`. Both also involve a random component and can be seeded with `--seed <seed>`. The default setting is `--seed 0`.

3.2.2 Cross validation

k-fold cross-validation can be run by specifying `--num_folds <k>`. The default is `--num_folds 1`.

3.2.3 Ensembling

To train an ensemble, specify the number of models in the ensemble with `--ensemble_size <n>`. The default is `--ensemble_size 1`.

3.2.4 Hyperparameter Optimization

Although the default message passing architecture works quite well on a variety of datasets, optimizing the hyperparameters for a particular dataset often leads to marked improvement in predictive performance. We have automated hyperparameter optimization via Bayesian optimization (using the hyperopt package), which will find the optimal hidden size, depth, dropout, and number of feed-forward layers for our model. Optimization can be run as follows:

```
chemprop_hyperopt --data_path <data_path> --dataset_type <type> --num_iters <n> --config_
↪save_path <config_path>
```

where `<n>` is the number of hyperparameter settings to try and `<config_path>` is the path to a `.json` file where the optimal hyperparameters will be saved.

If installed from source, `chemprop_hyperopt` can be replaced with `python hyperparameter_optimization.py`.

Once hyperparameter optimization is complete, the optimal hyperparameters can be applied during training by specifying the config path as follows:

```
chemprop_train --data_path <data_path> --dataset_type <type> --config_path <config_path>
```

Note that the hyperparameter optimization script sees all the data given to it. The intended use is to run the hyperparameter optimization script on a dataset with the eventual test set held out. If you need to optimize hyperparameters separately for several different cross validation splits, you should e.g. set up a bash script to run hyperparameter_optimization.py separately on each split's training and validation data with test held out.

3.2.5 Additional Features

While the model works very well on its own, especially after hyperparameter optimization, we have seen that additional features can further improve performance on certain datasets. The additional features can be added at the atom-, bond, or molecule-level. Molecule-level features can be either automatically generated by RDKit or custom features provided by the user.

Molecule-Level RDKit 2D Features

As a starting point, we recommend using pre-normalized RDKit features by using the `--features_generator rdkit_2d_normalized --no_features_scaling` flags. In general, we recommend NOT using the `--no_features_scaling` flag (i.e. allow the code to automatically perform feature scaling), but in the case of `rdkit_2d_normalized`, those features have been pre-normalized and don't require further scaling.

The full list of available features for `--features_generator` is as follows.

`morgan` is binary Morgan fingerprints, radius 2 and 2048 bits. `morgan_count` is count-based Morgan, radius 2 and 2048 bits. `rdkit_2d` is an unnormalized version of 200 assorted rdkit descriptors. Full list can be found at the bottom of our paper: https://arxiv.org/pdf/1904.01561.pdf `rdkit_2d_normalized` is the CDF-normalized version of the 200 rdkit descriptors.

Molecule-Level Custom Features

If you install from source, you can modify the code to load custom features as follows:

1. **Generate features:** If you want to generate features in code, you can write a custom features generator function in `chemprop/features/features_generators.py`. Scroll down to the bottom of that file to see a features generator code template.
2. **Load features:** If you have features saved as a numpy `.npy` file or as a `.csv` file, you can load the features by using `--features_path /path/to/features`. Note that the features must be in the same order as the SMILES strings in your data file. Also note that `.csv` files must have a header row and the features should be comma-separated with one line per molecule.

Atom-Level Features

Similar to the additional molecular features described above, you can also provide additional atomic features via `--atom_descriptors_path /path/to/features` with valid file formats:

- `.npz` file, where descriptors are saved as 2D array for each molecule in the exact same order as the SMILES strings in your data file.
- `.pkl` / `.pckl` / `.pickle` containing a pandas dataframe with smiles as index and numpy array of descriptors as columns.
- `.sdf` containing all mol blocks with descriptors as entries.

The order of the descriptors for each atom per molecule must match the ordering of atoms in the RDKit molecule object. Further information on supplying atomic descriptors can be found here.

Users must select in which way atom descriptors are used. The command line option `--atom_descriptors descriptor` concatenates the new features to the embedded atomic features after the D-MPNN with an additional linear layer. The option `--atom_descriptors feature` concatenates the features to each atomic feature vector before the D-MPNN, so that they are used during message-passing. Alternatively, the user can overwrite the default atom features with the custom features using the option `--overwrite_default_atom_features`.

Similar to the molecule-level features, the atom-level descriptors and features are scaled by default. This can be disabled with the option `--no_atom_descriptor_scaling`

Bond-Level Features

Bond-level features can be provided in the same format as the atom-level features, using the option `--bond_features_path /path/to/features`. The order of the features for each molecule must match the bond ordering in the RDKit molecule object.

The bond-level features are concatenated with the bond feature vectors before the D-MPNN, such that they are used during message-passing. Alternatively, the user can overwrite the default bond features with the custom features using the option `--overwrite_default_bond_features`.

Similar to molecule-, and atom-level features, the bond-level features are scaled by default. This can be disabled with the option `--no_bond_features_scaling`.

3.2.6 Reaction

As an alternative to molecule SMILES, Chemprop can also process atom-mapped reaction SMILES (see Daylight manual for details on reaction SMILES), which consist of three parts denoting reactants, agents and products, separated by ">". Use the option `--reaction` to enable the input of reactions, which transforms the reactants and products of each reaction to the corresponding condensed graph of reaction and changes the initial atom and bond features to hold information from both the reactant and product (option `--reaction_mode reac_prod`), or from the reactant and the difference upon reaction (option `--reaction_mode reac_diff`, default) or from the product and the difference upon reaction (option `--reaction_mode prod_diff`). In reaction mode, Chemprop thus concatenates information to each atomic and bond feature vector, for example, with option `--reaction_mode reac_prod`, each atomic feature vector holds information on the state of the atom in the reactant (similar to default Chemprop), and concatenates information on the state of the atom in the product, so that the size of the D-MPNN increases slightly. Agents are discarded. Functions incompatible with a reaction as input (scaffold splitting and feature generation) are carried out on the reactants only. If the atom-mapped reaction SMILES contain mapped hydrogens, enable explicit hydrogens via `--explicit_h`. Example of an atom-mapped reaction SMILES denoting the reaction of methanol to formaldehyde without hydrogens: `[CH3:1][OH:2]>>[CH2:1]=[O:2]` and with hydrogens: `[C:1]([H:3])([H:4])([H:5])[O:2][H:6]>>[C:1]([H:3])([H:4])=[O:2].[H:5][H:6]`. The reactions do not need to be balanced and can thus contain unmapped parts, for example leaving groups, if necessary. For further details and benchmarking, as well as a citable reference, please see DOI 10.33774/chemrxiv-2021-frfhz.

3.2.7 Pretraining

An existing model, for example from training on a larger, lower quality dataset, can be used for parameter-initialization of a new model by providing a checkpoint of the existing model using either:

- `--checkpoint_dir` <dir> Directory where the model checkpoint(s) are saved (i.e. `--save_dir` during training of the old model). This will walk the directory, and load all `.pt` files it finds.
- `--checkpoint_path` <path> Path to a model checkpoint file (`.pt` file).

when training the new model. The model architecture of the new model should resemble the architecture of the old model - otherwise some or all parameters might not be loaded correctly. Please note that the old model is only used to

initialize the parameters of the new model, but all parameters remain trainable (no frozen layers). Depending on the quality of the old model, the new model might only need a few epochs to train.

3.2.8 Missing target values

When training multitask models (models which predict more than one target simultaneously), sometimes not all target values are known for all molecules in the dataset. Chemprop automatically handles missing entries in the dataset by masking out the respective values in the loss function, so that partial data can be utilized, too. The loss function is rescaled according to all non-missing values, and missing values furthermore do not contribute to validation or test errors. Training on partial data is therefore possible and encouraged (versus taking out datapoints with missing target entries). No keyword is needed for this behavior, it is the default.

In contrast, when using `sklearn_train.py` (a utility script provided within Chemprop that trains standard models such as random forests on Morgan fingerprints via the python package scikit-learn), multi-task models cannot be trained on datasets with partially missing targets. However, one can instead train individual models for each task (via the argument `--single_task`), where missing values are automatically removed from the dataset. Thus, the training still makes use of all non-missing values, but by training individual models for each task, instead of one model with multiple output values. This restriction only applies to sklearn models (via `sklearn_train` or `python sklearn_train.py`), but NOT to default Chemprop models via `chemprop_train` or `python train.py`.

3.2.9 Caching

By default, the molecule objects created from each SMILES string are cached for all dataset sizes, and the graph objects created from each molecule object are cached for datasets up to 10000 molecules. If memory permits, you may use the keyword `--cache_cutoff inf` to set this cutoff from 10000 to infinity to always keep the generated graphs in cache (or to another integer value for custom behavior). This may speed up training (depending on the dataset size, molecule size, number of epochs and GPU support), since the graphs do not need to be recreated each epoch, but increases memory usage considerably. Below the cutoff, graphs are created sequentially in the first epoch. Above the cutoff, graphs are created in parallel (on `--num_workers <int>` workers) for each epoch. If training on a GPU, training without caching and creating graphs on the fly in parallel is often preferable. On CPU, training with caching if often preferable for medium-sized datasets and a very low number of CPUs. If a very large dataset causes memory issues, you might turn off caching even of the molecule objects via the commands `--no_cache_mol` to reduce memory usage further.

3.3 Predicting

To load a trained model and make predictions, run `predict.py` and specify:

- `--test_path <path>` Path to the data to predict on.
- A checkpoint by using either:
  - `--checkpoint_dir <dir>` Directory where the model checkpoint(s) are saved (i.e. `--save_dir` during training). This will walk the directory, load all `.pt` files it finds, and treat the models as an ensemble.
  - `--checkpoint_path <path>` Path to a model checkpoint file (`.pt` file).
- `--preds_path` Path where a CSV file containing the predictions will be saved.

For example:

```
chemprop_predict --test_path data/tox21.csv --checkpoint_dir tox21_checkpoints --preds_
↪path tox21_preds.csv
```

or

```
chemprop_predict --test_path data/tox21.csv --checkpoint_path tox21_checkpoints/fold_0/
↪model_0/model.pt --preds_path tox21_preds.csv
```

If installed from source, `chemprop_predict` can be replaced with `python predict.py`.

3.3.1 Interpreting

It is often helpful to provide explanation of model prediction (i.e., this molecule is toxic because of this substructure). Given a trained model, you can interpret the model prediction using the following command:

```
chemprop_interpret --data_path data/tox21.csv --checkpoint_dir tox21_checkpoints/fold_0/␣
↪--property_id 1
```

If installed from source, `chemprop_interpret` can be replaced with `python interpret.py`.

The output will be like the following:

- The first column is a molecule and second column is its predicted property (in this case NR-AR toxicity).
- The third column is the smallest substructure that made this molecule classified as toxic (which we call rationale).
- The fourth column is the predicted toxicity of that substructure.

As shown in the first row, when a molecule is predicted to be non-toxic, we will not provide any rationale for its prediction.

smiles	NR-AR	rationale	rationale_score
O=[N+]([O-])c1cc(C(F)(F)F)cc([N+](=O)[O-])c1Cl	0.014		
CC1(C)O[C@@H]2C[C@H]3[C@@H]4C[C@H](F)C5=CC(=O)C=C[C@]5(C)[C@H]4[C@@H](O)C[C@]3(C)[C@]2(C(=O)CO)O1	0.896	C[C@]12C=CC(=O)C=C1[CH2:1]C[CH2:1][CH2:1]2	0.769
C[C@]12CC[C@H]3[C@@H](CC[C@@]45O[C@@H]4C(O)=C(C#N)C[C@]35C)[C@@H]1CC[C@@H]2O	0.945	C[C@]12C[CH:1]=C(C#N)C[C@]1(C)[C@@H]1[CH2:1]...	0.808
C[C@]12C[C@H](O)[C@H]3[C@@H](CCC4=CC(=O)CC[C@@]43C)[C@@H]1CC[C@]2(O)C(=O)COP(=O)([O-])[O-]	0.957	C1C[CH2:1]... [C:1]1C[CH2:1...	0.653

Chemprop's interpretation script explains model prediction one property at a time. `--property_id 1` tells the script to provide explanation for the first property in the dataset (which is NR-AR). In a multi-task training setting, you will need to change `--property_id` to provide explanation for each property in the dataset.

For computational efficiency, we currently restricted the rationale to have maximum 20 atoms and minimum 8 atoms. You can adjust these constraints through `--max_atoms` and `--min_atoms` argument.

Please note that the interpreting framework is currently only available for models trained on properties of single molecules, that is, multi-molecule models generated via the `--number_of_molecules` command are not supported.

3.3.2 TensorBoard

During training, TensorBoard logs are automatically saved to the same directory as the model checkpoints. To view TensorBoard logs, run `tensorboard --logdir=<dir>` where `<dir>` is the path to the checkpoint directory. Then navigate to http://localhost:6006.

3.4 Web Interface

For those less familiar with the command line, Chemprop also includes a web interface which allows for basic training and predicting. See *Web Interface* for more details.

3.5 Within a python script

Model training and predicting can also be embedded within a python script. To train a model, provide arguments as a list of strings (arguments are identical to command line mode), parse the arguments, and then call `chemprop.train.cross_validate()`:

```
import chemprop

arguments = [
    '--data_path', 'data/tox21.csv',
    '--dataset_type', 'classification',
    '--save_dir', 'tox21_checkpoints'
]

args = chemprop.args.TrainArgs().parse_args(arguments)
mean_score, std_score = chemprop.train.cross_validate(args=args, train_func=chemprop.
↪train.run_training)
```

For predicting with a given model, either a list of smiles or a csv file can be used as input. To use a csv file

```
import chemprop

arguments = [
    '--test_path', 'data/tox21.csv',
    '--preds_path', 'tox21_preds.csv',
    '--checkpoint_dir', 'tox21_checkpoints'
]

args = chemprop.args.PredictArgs().parse_args(arguments)
preds = chemprop.train.make_predictions(args=args)
```

If you only want to use the predictions `preds` within the script, and not save the file, set `preds_path` to `/dev/null`. To predict on a list of smiles, run:

```
import chemprop

smiles = [['CCC', 'CCCC', 'OCC']]
arguments = [
    '--test_path', '/dev/null',
```

(continues on next page)

(continued from previous page)

```
    '--preds_path', '/dev/null',
    '--checkpoint_dir', 'tox21_checkpoints'
]

args = chemprop.args.PredictArgs().parse_args(arguments)
preds = chemprop.train.make_predictions(args=args, smiles=smiles)
```

where the given `test_path` will be discarded if a list of smiles is provided. If you want to predict multiple sets of molecules consecutively, it is more efficient to only load the chemprop model once, and then predict with the preloaded model (instead of loading the model for every prediction):

```
import chemprop

arguments = [
    '--test_path', '/dev/null',
    '--preds_path', '/dev/null',
    '--checkpoint_dir', 'tox21_checkpoints'
]

args = chemprop.args.PredictArgs().parse_args(arguments)

model_objects = chemprop.train.load_model(args=args)

smiles = [['CCC', 'CCCC', 'OCC']]
preds = chemprop.train.make_predictions(args=args, smiles=smiles, model_objects=model_
↪objects)

smiles = [['CCCC', 'CCCCC', 'COCC']]
preds = chemprop.train.make_predictions(args=args, smiles=smiles, model_objects=model_
↪objects)
```

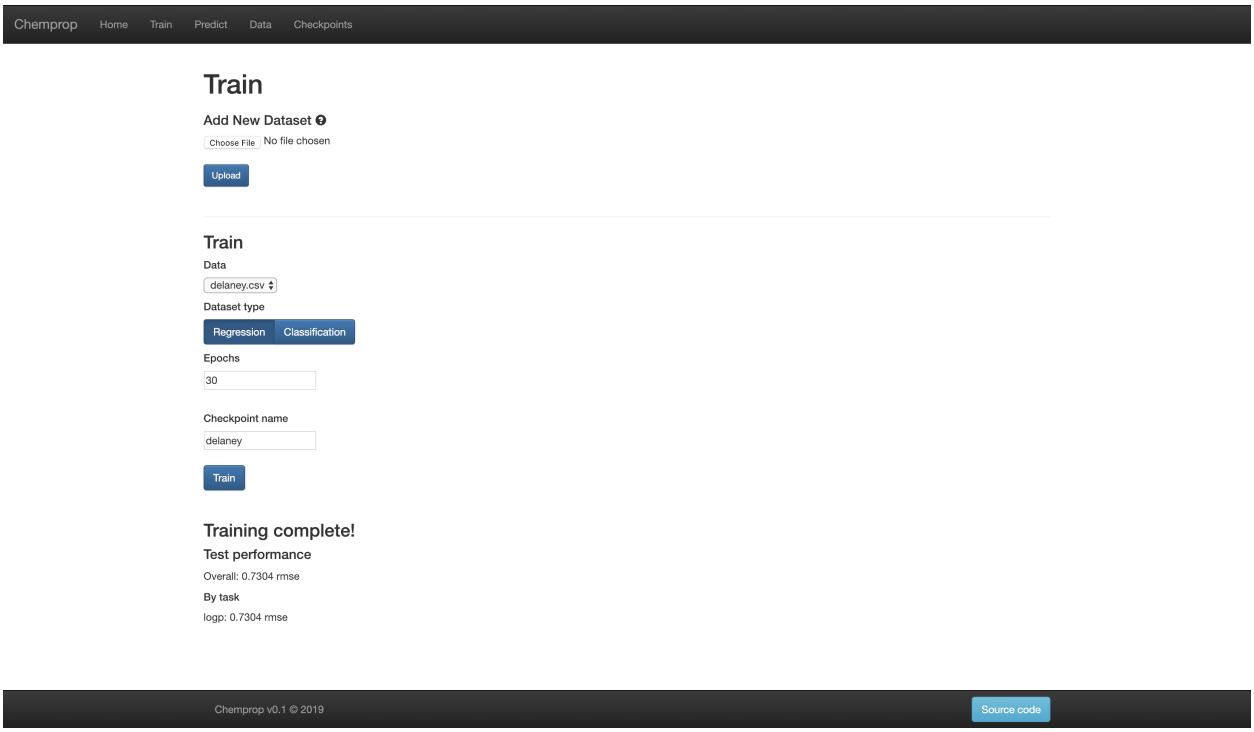
CHAPTER

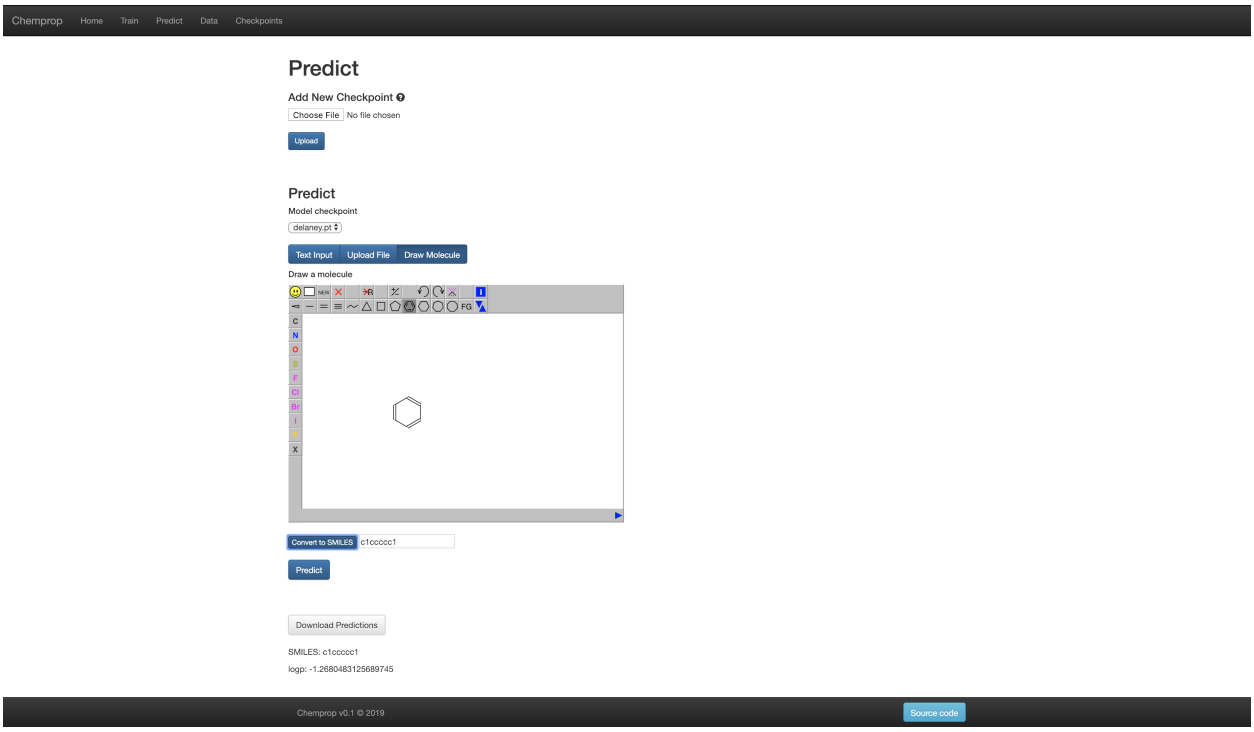
FOUR

WEB INTERFACE

4.1 Overview

For those less familiar with the command line, Chemprop also includes a web interface which allows for basic training and predicting. An example of the website (in demo mode with training disabled) is available here: chemprop.csail.mit.edu.





You can start the web interface on your local machine in two ways. Flask is used for development mode while gunicorn is used for production mode.

4.2 Flask

Run `chemprop_web` (or optionally `python web.py` if installed from source) and then navigate to localhost:5000 in a web browser.

4.3 Gunicorn

Gunicorn is only available for a UNIX environment, meaning it will not work on Windows. It is not installed by default with the rest of Chemprop, so first run:

```
pip install gunicorn
```

Next, navigate to `chemprop/web` and run `gunicorn --bind {host}:{port} 'wsgi:build_app()'`. This will start the site in production mode.

- To run this server in the background, add the `--daemon` flag.
- Arguments including `init_db` and `demo` can be passed with this pattern: `'wsgi:build_app(init_db=True, demo=True)'`
- Gunicorn documentation can be found [here](http://docs.gunicorn.org/en/stable/index.html).

CHAPTER

FIVE

DATA

chemprop.data contains functions and classes for loading, containing, and splitting data.

5.1 Data

Classes and functions from chemprop.data.data.py.

class chemprop.data.data.**MoleculeDataLoader**(*dataset:* chemprop.data.data.MoleculeDataset, *batch_size: int = 50*, *num_workers: int = 8*, *class_balance: bool = False*, *shuffle: bool = False*, *seed: int = 0*)

A *MoleculeDataLoader* is a PyTorch `DataLoader` for loading a *MoleculeDataset*.

Parameters

- **dataset** – The *MoleculeDataset* containing the molecules to load.
- **batch_size** – Batch size.
- **num_workers** – Number of workers used to build batches.
- **class_balance** – Whether to perform class balancing (i.e., use an equal number of positive and negative molecules). Class balance is only available for single task classification datasets. Set shuffle to True in order to get a random subset of the larger class.
- **shuffle** – Whether to shuffle the data.
- **seed** – Random seed. Only needed if shuffle is True.

property iter_size: int

Returns the number of data points included in each full iteration through the *MoleculeDataLoader*.

property targets: List[List[Optional[float]]]

Returns the targets associated with each molecule.

Returns A list of lists of floats (or None) containing the targets.

class chemprop.data.data.**MoleculeDatapoint**(*smiles: List[str]*, *targets: Optional[List[Optional[float]]] = None*, *row: Optional[collections.OrderedDict] = None*, *data_weight: float = 1*, *features: Optional[numpy.ndarray] = None*, *features_generator: Optional[List[str]] = None*, *phase_features: Optional[List[float]] = None*, *atom_features: Optional[numpy.ndarray] = None*, *atom_descriptors: Optional[numpy.ndarray] = None*, *bond_features: Optional[numpy.ndarray] = None*, *overwrite_default_atom_features: bool = False*, *overwrite_default_bond_features: bool = False*)

A *MoleculeDatapoint* contains a single molecule and its associated features and targets.

Parameters

- **smiles** – A list of the SMILES strings for the molecules.
- **targets** – A list of targets for the molecule (contains None for unknown target values).
- **row** – The raw CSV row containing the information for this molecule.
- **data_weight** – Weighting of the datapoint for the loss function.
- **features** – A numpy array containing additional features (e.g., Morgan fingerprint).
- **features_generator** – A list of features generators to use.
- **phase_features** – A one-hot vector indicating the phase of the data, as used in spectra data.
- **atom_descriptors** – A numpy array containing additional atom descriptors to featurize the molecule
- **bond_features** – A numpy array containing additional bond features to featurize the molecule
- **overwrite_default_atom_features** – Boolean to overwrite default atom features by atom_features
- **overwrite_default_bond_features** – Boolean to overwrite default bond features by bond_features

extend_features(*features: numpy.ndarray*) → None

Extends the features of the molecule.

Parameters **features** – A 1D numpy array of extra features for the molecule.

property mol: Union[List[rdkit.Chem.rdchem.Mol], List[Tuple[rdkit.Chem.rdchem.Mol, rdkit.Chem.rdchem.Mol]]]

Gets the corresponding list of RDKit molecules for the corresponding SMILES list.

num_tasks() → int

Returns the number of prediction tasks.

Returns The number of tasks.

property number_of_molecules: int

Gets the number of molecules in the *MoleculeDatapoint*.

Returns The number of molecules.

reset_features_and_targets() → None

Resets the features (atom, bond, and molecule) and targets to their raw values.

set_atom_descriptors(*atom_descriptors: numpy.ndarray*) → None
Sets the atom descriptors of the molecule.

Parameters **atom_descriptors** – A 1D numpy array of features for the molecule.

set_atom_features(*atom_features: numpy.ndarray*) → None
Sets the atom features of the molecule.

Parameters **atom_features** – A 1D numpy array of features for the molecule.

set_bond_features(*bond_features: numpy.ndarray*) → None
Sets the bond features of the molecule.

Parameters **bond_features** – A 1D numpy array of features for the molecule.

set_features(*features: numpy.ndarray*) → None
Sets the features of the molecule.

Parameters **features** – A 1D numpy array of features for the molecule.

set_targets(*targets: List[Optional[float]]*)
Sets the targets of a molecule.

Parameters **targets** – A list of floats containing the targets.

class chemprop.data.data.**MoleculeDataset**(*data: List[*chemprop.data.data.MoleculeDatapoint*]*)
A *MoleculeDataset* contains a list of *MoleculeDatapoint*s with access to their attributes.

Parameters **data** – A list of *MoleculeDatapoint*s.

atom_descriptors() → List[numpy.ndarray]
Returns the atom descriptors associated with each molecule (if they exit).

Returns A list of 2D numpy arrays containing the atom descriptors for each molecule or None if there are no features.

atom_descriptors_size() → int
Returns the size of custom additional atom descriptors vector associated with the molecules.

Returns The size of the additional atom descriptor vector.

atom_features() → List[numpy.ndarray]
Returns the atom descriptors associated with each molecule (if they exit).

Returns A list of 2D numpy arrays containing the atom descriptors for each molecule or None if there are no features.

atom_features_size() → int
Returns the size of custom additional atom features vector associated with the molecules.

Returns The size of the additional atom feature vector.

batch_graph() → List[*chemprop.features.featurization.BatchMolGraph*]
Constructs a BatchMolGraph with the graph featurization of all the molecules.

Note: The BatchMolGraph is cached in after the first time it is computed and is simply accessed upon subsequent calls to *batch_graph()*. This means that if the underlying set of *MoleculeDatapoint*s changes, then the returned BatchMolGraph will be incorrect for the underlying data.

Returns A list of BatchMolGraph containing the graph featurization of all the molecules in each *MoleculeDatapoint*.

bond_features() → List[numpy.ndarray]
Returns the bond features associated with each molecule (if they exit).

Returns A list of 2D numpy arrays containing the bond features for each molecule or None if there are no features.

bond_features_size() → int
Returns the size of custom additional bond features vector associated with the molecules.

Returns The size of the additional bond feature vector.

data_weights() → List[float]
Returns the loss weighting associated with each molecule

features() → List[numpy.ndarray]
Returns the features associated with each molecule (if they exist).

Returns A list of 1D numpy arrays containing the features for each molecule or None if there are no features.

features_size() → int
Returns the size of the additional features vector associated with the molecules.

Returns The size of the additional features vector.

mols(*flatten: bool = False*) → Union[List[rdkit.Chem.rdchem.Mol], List[List[rdkit.Chem.rdchem.Mol]], List[Tuple[rdkit.Chem.rdchem.Mol, rdkit.Chem.rdchem.Mol]], List[List[Tuple[rdkit.Chem.rdchem.Mol, rdkit.Chem.rdchem.Mol]]]]
Returns a list of the RDKit molecules associated with each *MoleculeDatapoint*.

Parameters **flatten** – Whether to flatten the returned RDKit molecules to a list instead of a list of lists.

Returns A list of SMILES or a list of lists of RDKit molecules, depending on `flatten`.

normalize_features(*scaler: Optional[*chemprop.data.scaler.StandardScaler*] = None*, *replace_nan_token: int = 0*, *scale_atom_descriptors: bool = False*, *scale_bond_features: bool = False*) → *chemprop.data.scaler.StandardScaler*
Normalizes the features of the dataset using a `StandardScaler`.

The `StandardScaler` subtracts the mean and divides by the standard deviation for each feature independently.

If a `StandardScaler` is provided, it is used to perform the normalization. Otherwise, a `StandardScaler` is first fit to the features in this dataset and is then used to perform the normalization.

Parameters

- **scaler** – A fitted `StandardScaler`. If it is provided it is used, otherwise a new `StandardScaler` is first fitted to this data and is then used.
- **replace_nan_token** – A token to use to replace NaN entries in the features.
- **scale_atom_descriptors** – If the features that need to be scaled are atom features rather than molecule.
- **scale_bond_features** – If the features that need to be scaled are bond descriptors rather than molecule.

Returns A fitted `StandardScaler`. If a `StandardScaler` is provided as a parameter, this is the same `StandardScaler`. Otherwise, this is a new `StandardScaler` that has been fit on this dataset.

normalize_targets() → *chemprop.data.scaler.StandardScaler*
Normalizes the targets of the dataset using a StandardScaler.

The StandardScaler subtracts the mean and divides by the standard deviation for each task independently.

This should only be used for regression datasets.

Returns A StandardScaler fitted to the targets.

num_tasks() → int
Returns the number of prediction tasks.

Returns The number of tasks.

property number_of_molecules: int
Gets the number of molecules in each *MoleculeDatapoint*.

Returns The number of molecules.

phase_features() → List[numpy.ndarray]
Returns the phase features associated with each molecule (if they exist).

Returns A list of 1D numpy arrays containing the phase features for each molecule or None if there are no features.

reset_features_and_targets() → None
Resets the features (atom, bond, and molecule) and targets to their raw values.

set_targets(*targets: List[List[Optional[float]]]*) → None
Sets the targets for each molecule in the dataset. Assumes the targets are aligned with the datapoints.

Parameters **targets** – A list of lists of floats (or None) containing targets for each molecule. This must be the same length as the underlying dataset.

smiles(*flatten: bool = False*) → Union[List[str], List[List[str]]]
Returns a list containing the SMILES list associated with each *MoleculeDatapoint*.

Parameters **flatten** – Whether to flatten the returned SMILES to a list instead of a list of lists.

Returns A list of SMILES or a list of lists of SMILES, depending on flatten.

targets() → List[List[Optional[float]]]
Returns the targets associated with each molecule.

Returns A list of lists of floats (or None) containing the targets.

class chemprop.data.data.**MoleculeSampler**(*dataset:* chemprop.data.data.MoleculeDataset, *class_balance: bool = False*, *shuffle: bool = False*, *seed: int = 0*)
A *MoleculeSampler* samples data from a *MoleculeDataset* for a *MoleculeDataLoader*.

Parameters

- **class_balance** – Whether to perform class balancing (i.e., use an equal number of positive and negative molecules). Set shuffle to True in order to get a random subset of the larger class.
- **shuffle** – Whether to shuffle the data.
- **seed** – Random seed. Only needed if shuffle is True.

chemprop.data.data.**cache_graph**() → bool
Returns whether MolGraphs will be cached.

chemprop.data.data.**cache_mol**() → bool
Returns whether RDKit molecules will be cached.

chemprop.data.data.**construct_molecule_batch**(*data: List[*chemprop.data.data.MoleculeDatapoint*]*) → *chemprop.data.data.MoleculeDataset*

Constructs a *MoleculeDataset* from a list of *MoleculeDatapoint*s.

Additionally, precomputes the BatchMolGraph for the constructed *MoleculeDataset*.

Parameters **data** – A list of *MoleculeDatapoint*s.

Returns A *MoleculeDataset* containing all the *MoleculeDatapoint*s.

chemprop.data.data.**empty_cache**()

Empties the cache of MolGraph and RDKit molecules.

chemprop.data.data.**make_mols**(*smiles: List[str]*, *reaction: bool*, *keep_h: bool*)

Builds a list of RDKit molecules (or a list of tuples of molecules if reaction is True) for a list of smiles.

Parameters

- **smiles** – List of SMILES strings.
- **reaction** – Boolean whether the SMILES strings are to be treated as a reaction.
- **keep_h** – Boolean whether to keep hydrogens in the input smiles. This does not add hydrogens, it only keeps them if they are specified.

Returns List of RDKit molecules or list of tuple of molecules.

chemprop.data.data.**set_cache_graph**(*cache_graph: bool*) → None

Sets whether MolGraphs will be cached.

chemprop.data.data.**set_cache_mol**(*cache_mol: bool*) → None

Sets whether RDKit molecules will be cached.

5.2 Scaffold

Classes and functions from chemprop.data.scaffold.py.

chemprop.data.scaffold.**generate_scaffold**(*mol: Union[str, rdkit.Chem.rdchem.Mol, Tuple[rdkit.Chem.rdchem.Mol, rdkit.Chem.rdchem.Mol]]*, *include_chirality: bool = False*) → str

Computes the Bemis-Murcko scaffold for a SMILES string.

Parameters

- **mol** – A SMILES or an RDKit molecule.
- **include_chirality** – Whether to include chirality in the computed scaffold..

Returns The Bemis-Murcko scaffold for the molecule.

chemprop.data.scaffold.**log_scaffold_stats**(*data:* chemprop.data.data.MoleculeDataset, *index_sets: List[Set[int]]*, *num_scaffolds: int = 10*, *num_labels: int = 20*, *logger: Optional[logging.Logger] = None*) → List[Tuple[List[float], List[int]]]

Logs and returns statistics about counts and average target values in molecular scaffolds.

Parameters

- **data** – A MoleculeDataset.
- **index_sets** – A list of sets of indices representing splits of the data.
- **num_scaffolds** – The number of scaffolds about which to display statistics.

- **num_labels** – The number of labels about which to display statistics.
- **logger** – A logger for recording output.

Returns A list of tuples where each tuple contains a list of average target values across the first `num_labels` labels and a list of the number of non-zero values for the first `num_scaffolds` scaffolds, sorted in decreasing order of scaffold frequency.

chemprop.data.scaffold.**scaffold_split**(*data:* chemprop.data.data.MoleculeDataset, *sizes: Tuple[float, float, float] = (0.8, 0.1, 0.1)*, *balanced: bool = False*, *seed: int = 0*, *logger: Optional[logging.Logger] = None*) → Tuple[*chemprop.data.data.MoleculeDataset*, *chemprop.data.data.MoleculeDataset*, *chemprop.data.data.MoleculeDataset*]

Splits a `MoleculeDataset` by scaffold so that no molecules sharing a scaffold are in different splits.

Parameters

- **data** – A `MoleculeDataset`.
- **sizes** – A length-3 tuple with the proportions of data in the train, validation, and test sets.
- **balanced** – Whether to balance the sizes of scaffolds in each set rather than putting the smallest in test set.
- **seed** – Random seed for shuffling when doing balanced splitting.
- **logger** – A logger for recording output.

Returns A tuple of `MoleculeDataset`s containing the train, validation, and test splits of the data.

chemprop.data.scaffold.**scaffold_to_smiles**(*mols: Union[List[str], List[rdkit.Chem.rdchem.Mol], List[Tuple[rdkit.Chem.rdchem.Mol, rdkit.Chem.rdchem.Mol]]]*, *use_indices: bool = False*) → Dict[str, Union[Set[str], Set[int]]]

Computes the scaffold for each SMILES and returns a mapping from scaffolds to sets of smiles (or indices).

Parameters

- **mols** – A list of SMILES or RDKit molecules.
- **use_indices** – Whether to map to the SMILES's index in `mols` rather than mapping to the smiles string itself. This is necessary if there are duplicate smiles.

Returns A dictionary mapping each unique scaffold to all SMILES (or indices) which have that scaffold.

5.3 Scaler

Classes and functions from chemprop.data.scaler.py.

class chemprop.data.scaler.**StandardScaler**(*means: Optional[numpy.ndarray] = None*, *stds: Optional[numpy.ndarray] = None*, *replace_nan_token: Optional[Any] = None*)

A *StandardScaler* normalizes the features of a dataset.

When it is fit on a dataset, the *StandardScaler* learns the mean and standard deviation across the 0th axis. When transforming a dataset, the *StandardScaler* subtracts the means and divides by the standard deviations.

Parameters

- **means** – An optional 1D numpy array of precomputed means.

- **stds** – An optional 1D numpy array of precomputed standard deviations.
- **replace_nan_token** – A token to use to replace NaN entries in the features.

fit(*X: List[List[Optional[float]]]*) → *chemprop.data.scaler.StandardScaler*
Learns means and standard deviations across the 0th axis of the data **X**.

Parameters **X** – A list of lists of floats (or None).

Returns The fitted *StandardScaler* (self).

inverse_transform(*X: List[List[Optional[float]]]*) → numpy.ndarray
Performs the inverse transformation by multiplying by the standard deviations and adding the means.

Parameters **X** – A list of lists of floats.

Returns The inverse transformed data with NaNs replaced by `self.replace_nan_token`.

transform(*X: List[List[Optional[float]]]*) → numpy.ndarray
Transforms the data by subtracting the means and dividing by the standard deviations.

Parameters **X** – A list of lists of floats (or None).

Returns The transformed data with NaNs replaced by `self.replace_nan_token`.

5.4 Utils

Classes and functions from chemprop.data.utils.py.

chemprop.data.utils.**filter_invalid_smiles**(*data:* chemprop.data.data.MoleculeDataset) → *chemprop.data.data.MoleculeDataset*
Filters out invalid SMILES.

Parameters **data** – A `MoleculeDataset`.

Returns A `MoleculeDataset` with only the valid molecules.

chemprop.data.utils.**get_class_sizes**(*data:* chemprop.data.data.MoleculeDataset) → List[List[float]]
Determines the proportions of the different classes in a classification dataset.

Parameters **data** – A classification `MoleculeDataset`.

Returns A list of lists of class proportions. Each inner list contains the class proportions for a task.

chemprop.data.utils.**get_data**(*path: str*, *smiles_columns: Optional[Union[str, List[str]]] = None*, *target_columns: Optional[List[str]] = None*, *ignore_columns: Optional[List[str]] = None*, *skip_invalid_smiles: bool = True*, *args: Optional[Union[*chemprop.args.TrainArgs, chemprop.args.PredictArgs*]] = None*, *data_weights_path: Optional[str] = None*, *features_path: Optional[List[str]] = None*, *features_generator: Optional[List[str]] = None*, *phase_features_path: Optional[str] = None*, *atom_descriptors_path: Optional[str] = None*, *bond_features_path: Optional[str] = None*, *max_data_size: Optional[int] = None*, *store_row: bool = False*, *logger: Optional[logging.Logger] = None*, *skip_none_targets: bool = False*) → *chemprop.data.data.MoleculeDataset*
Gets SMILES and target values from a CSV file.

Parameters

- **path** – Path to a CSV file.

- **smiles_columns** – The names of the columns containing SMILES. By default, uses the first `number_of_molecules` columns.
- **target_columns** – Name of the columns containing target values. By default, uses all columns except the `smiles_column` and the `ignore_columns`.
- **ignore_columns** – Name of the columns to ignore when `target_columns` is not provided.
- **skip_invalid_smiles** – Whether to skip and filter out invalid smiles using *filter_invalid_smiles()*.
- **args** – Arguments, either *TrainArgs* or *PredictArgs*.
- **data_weights_path** – A path to a file containing weights for each molecule in the loss function.
- **features_path** – A list of paths to files containing features. If provided, it is used in place of `args.features_path`.
- **features_generator** – A list of features generators to use. If provided, it is used in place of `args.features_generator`.
- **phase_features_path** – A path to a file containing phase features as applicable to spectra.
- **atom_descriptors_path** – The path to the file containing the custom atom descriptors.
- **bond_features_path** – The path to the file containing the custom bond features.
- **max_data_size** – The maximum number of data points to load.
- **logger** – A logger for recording output.
- **store_row** – Whether to store the raw CSV row in each *MoleculeDatapoint*.
- **skip_none_targets** – Whether to skip targets that are all 'None'. This is mostly relevant when –target_columns are passed in, so only a subset of tasks are examined.

Returns A `MoleculeDataset` containing SMILES and target values along with other info such as additional features when desired.

`chemprop.data.utils.`**get_data_from_smiles**(*smiles: List[List[str]]*, *skip_invalid_smiles: bool = True*, *logger: Optional[logging.Logger] = None*, *features_generator: Optional[List[str]] = None*) → *chemprop.data.data.MoleculeDataset*

Converts a list of SMILES to a `MoleculeDataset`.

Parameters

- **smiles** – A list of lists of SMILES with length depending on the number of molecules.
- **skip_invalid_smiles** – Whether to skip and filter out invalid smiles using *filter_invalid_smiles()*
- **logger** – A logger for recording output.
- **features_generator** – List of features generators.

Returns A `MoleculeDataset` with all of the provided SMILES.

`chemprop.data.utils.`**get_data_weights**(*path: str*) → List[float]

Returns the list of data weights for the loss function as stored in a CSV file.

Parameters **path** – Path to a CSV file.

Returns A list of floats containing the data weights.

chemprop.data.utils.**get_header**(*path: str*) → List[str]
Returns the header of a data CSV file.

Parameters **path** – Path to a CSV file.

Returns A list of strings containing the strings in the comma-separated header.

chemprop.data.utils.**get_smiles**(*path: str*, *smiles_columns: Optional[Union[str, List[str]]] = None*, *header: bool = True*, *flatten: bool = False*) → Union[List[str], List[List[str]]]
Returns the SMILES from a data CSV file.

Parameters

- **path** – Path to a CSV file.
- **smiles_columns** – A list of the names of the columns containing SMILES. By default, uses the first `number_of_molecules` columns.
- **header** – Whether the CSV file contains a header.
- **flatten** – Whether to flatten the returned SMILES to a list instead of a list of lists.

Returns A list of SMILES or a list of lists of SMILES, depending on `flatten`.

chemprop.data.utils.**get_task_names**(*path: str*, *smiles_columns: Optional[Union[str, List[str]]] = None*, *target_columns: Optional[List[str]] = None*, *ignore_columns: Optional[List[str]] = None*) → List[str]
Gets the task names from a data CSV file.

If `target_columns` is provided, returns *target_columns*. Otherwise, returns all columns except the `smiles_columns` (or the first column, if the `smiles_columns` is None) and the `ignore_columns`.

Parameters

- **path** – Path to a CSV file.
- **smiles_columns** – The names of the columns containing SMILES. By default, uses the first `number_of_molecules` columns.
- **target_columns** – Name of the columns containing target values. By default, uses all columns except the `smiles_columns` and the `ignore_columns`.
- **ignore_columns** – Name of the columns to ignore when `target_columns` is not provided.

Returns A list of task names.

chemprop.data.utils.**preprocess_smiles_columns**(*path: str*, *smiles_columns: Optional[Union[str, List[Optional[str]]]]*, *number_of_molecules: int = 1*) → List[Optional[str]]
Preprocesses the `smiles_columns` variable to ensure that it is a list of column headings corresponding to the columns in the data file holding SMILES.

Parameters

- **path** – Path to a CSV file.
- **smiles_columns** – The names of the columns containing SMILES. By default, uses the first `number_of_molecules` columns.
- **number_of_molecules** – The number of molecules with associated SMILES for each data point.

Returns The preprocessed version of `smiles_columns` which is guaranteed to be a list.

chemprop.data.utils.**split_data**(*data:* chemprop.data.data.MoleculeDataset, *split_type: str = 'random'*, *sizes: Tuple[float, float, float] = (0.8, 0.1, 0.1)*, *seed: int = 0*, *num_folds: int = 1*, *args: Optional[*chemprop.args.TrainArgs*] = None*, *logger: Optional[logging.Logger] = None*) → Tuple[*chemprop.data.data.MoleculeDataset*, *chemprop.data.data.MoleculeDataset*, *chemprop.data.data.MoleculeDataset*]

Splits data into training, validation, and test splits.

Parameters

- **data** – A MoleculeDataset.
- **split_type** – Split type.
- **sizes** – A length-3 tuple with the proportions of data in the train, validation, and test sets.
- **seed** – The random seed to use before shuffling data.
- **num_folds** – Number of folds to create (only needed for "cv" split type).
- **args** – A *TrainArgs* object.
- **logger** – A logger for recording output.

Returns A tuple of MoleculeDatasets containing the train, validation, and test splits of the data.

chemprop.data.utils.**validate_data**(*data_path: str*) → Set[str]

Validates a data CSV file, returning a set of errors.

Parameters **data_path** – Path to a data CSV file.

Returns A set of error messages.

chemprop.data.utils.**validate_dataset_type**(*data:* chemprop.data.data.MoleculeDataset, *dataset_type: str*) → None

Validates the dataset type to ensure the data matches the provided type.

Parameters

- **data** – A MoleculeDataset.
- **dataset_type** – The dataset type to check.

CHAPTER

SIX

FEATURES

chemprop.features contains functions for featurizing molecules. This includes both atom/bond features used in message passing and additional molecule-level features appended after message passing.

6.1 Featurization

Classes and functions from chemprop.features.featurization.py. Featurization specifically includes computation of the atom and bond features used in message passing.

class chemprop.features.featurization.**BatchMolGraph**(*mol_graphs: List[*chemprop.features.featurization.MolGraph*]*)

A *BatchMolGraph* represents the graph structure and featurization of a batch of molecules.

A BatchMolGraph contains the attributes of a *MolGraph* plus:

- atom_fdim: The dimensionality of the atom feature vector.
- bond_fdim: The dimensionality of the bond feature vector (technically the combined atom/bond features).
- a_scope: A list of tuples indicating the start and end atom indices for each molecule.
- b_scope: A list of tuples indicating the start and end bond indices for each molecule.
- max_num_bonds: The maximum number of bonds neighboring an atom in this batch.
- b2b: (Optional) A mapping from a bond index to incoming bond indices.
- a2a: (Optional): A mapping from an atom index to neighboring atom indices.

Parameters **mol_graphs** – A list of *MolGraph*s from which to construct the *BatchMolGraph*.

get_a2a() → torch.LongTensor

Computes (if necessary) and returns a mapping from each atom index to all neighboring atom indices.

Returns A PyTorch tensor containing the mapping from each atom index to all the neighboring atom indices.

get_b2b() → torch.LongTensor

Computes (if necessary) and returns a mapping from each bond index to all the incoming bond indices.

Returns A PyTorch tensor containing the mapping from each bond index to all the incoming bond indices.

get_components(*atom_messages: bool = False*) → Tuple[torch.FloatTensor, torch.FloatTensor, torch.LongTensor, torch.LongTensor, torch.LongTensor, List[Tuple[int, int]], List[Tuple[int, int]]]

Returns the components of the *BatchMolGraph*.

The returned components are, in order:

- `f_atoms`
- `f_bonds`
- `a2b`
- `b2a`
- `b2revb`
- `a_scope`
- `b_scope`

Parameters **atom_messages** – Whether to use atom messages instead of bond messages. This changes the bond feature vector to contain only bond features rather than both atom and bond features.

Returns A tuple containing PyTorch tensors with the atom features, bond features, graph structure, and scope of the atoms and bonds (i.e., the indices of the molecules they belong to).

class chemprop.features.featurization.**Featurization_parameters**

A class holding molecule featurization parameters as attributes.

class chemprop.features.featurization.**MolGraph**(*mol: Union[str, rdkit.Chem.rdchem.Mol, Tuple[rdkit.Chem.rdchem.Mol, rdkit.Chem.rdchem.Mol]]*, *atom_features_extra: Optional[numpy.ndarray] = None*, *bond_features_extra: Optional[numpy.ndarray] = None*, *overwrite_default_atom_features: bool = False*, *overwrite_default_bond_features: bool = False*)

A *MolGraph* represents the graph structure and featurization of a single molecule.

A MolGraph computes the following attributes:

- `n_atoms`: The number of atoms in the molecule.
- `n_bonds`: The number of bonds in the molecule.
- `f_atoms`: A mapping from an atom index to a list of atom features.
- `f_bonds`: A mapping from a bond index to a list of bond features.
- `a2b`: A mapping from an atom index to a list of incoming bond indices.
- `b2a`: A mapping from a bond index to the index of the atom the bond originates from.
- `b2revb`: A mapping from a bond index to the index of the reverse bond.
- `overwrite_default_atom_features`: A boolean to overwrite default atom descriptors.
- `overwrite_default_bond_features`: A boolean to overwrite default bond descriptors.

Parameters

- **mol** – A SMILES or an RDKit molecule.
- **atom_features_extra** – A list of 2D numpy array containing additional atom features to featurize the molecule
- **bond_features_extra** – A list of 2D numpy array containing additional bond features to featurize the molecule

- **overwrite_default_atom_features** – Boolean to overwrite default atom features by atom_features instead of concatenating
- **overwrite_default_bond_features** – Boolean to overwrite default bond features by bond_features instead of concatenating

chemprop.features.featurization.**atom_features**(*atom: rdkit.Chem.rdchem.Atom*, *functional_groups: Optional[List[int]] = None*) → List[Union[bool, int, float]]

Builds a feature vector for an atom.

Parameters

- **atom** – An RDKit atom.
- **functional_groups** – A k-hot vector indicating the functional groups the atom belongs to.

Returns A list containing the atom features.

chemprop.features.featurization.**atom_features_zeros**(*atom: rdkit.Chem.rdchem.Atom*) → List[Union[bool, int, float]]

Builds a feature vector for an atom containing only the atom number information.

Parameters **atom** – An RDKit atom.

Returns A list containing the atom features.

chemprop.features.featurization.**bond_features**(*bond: rdkit.Chem.rdchem.Bond*) → List[Union[bool, int, float]]

Builds a feature vector for a bond.

Parameters **bond** – An RDKit bond.

Returns A list containing the bond features.

chemprop.features.featurization.**get_atom_fdim**(*overwrite_default_atom: bool = False*) → int

Gets the dimensionality of the atom feature vector.

Parameters **overwrite_default_atom** – Whether to overwrite the default atom descriptors

Returns The dimensionality of the atom feature vector.

chemprop.features.featurization.**get_bond_fdim**(*atom_messages: bool = False*, *overwrite_default_bond: bool = False*, *overwrite_default_atom: bool = False*) → int

Gets the dimensionality of the bond feature vector.

Parameters

- **atom_messages** – Whether atom messages are being used. If atom messages are used, then the bond feature vector only contains bond features. Otherwise it contains both atom and bond features.
- **overwrite_default_bond** – Whether to overwrite the default bond descriptors
- **overwrite_default_atom** – Whether to overwrite the default atom descriptors

Returns The dimensionality of the bond feature vector.

chemprop.features.featurization.**is_explicit_h**() → bool

Returns whether to use retain explicit Hs

chemprop.features.featurization.**is_reaction**() → bool

Returns whether to use reactions as input

chemprop.features.featurization.**map_reac_to_prod**(*mol_reac: rdkit.Chem.rdchem.Mol*, *mol_prod: rdkit.Chem.rdchem.Mol*)

Build a dictionary of mapping atom indices in the reactants to the products.

Parameters

- **mol_reac** – An RDKit molecule of the reactants.
- **mol_prod** – An RDKit molecule of the products.

Returns A dictionary of corresponding reactant and product atom indices.

chemprop.features.featurization.**mol2graph**(*mols: Union[List[str], List[rdkit.Chem.rdchem.Mol], List[Tuple[rdkit.Chem.rdchem.Mol, rdkit.Chem.rdchem.Mol]]]*, *atom_features_batch: List[numpy.array] = (None,)*, *bond_features_batch: List[numpy.array] = (None,)*, *overwrite_default_atom_features: bool = False*, *overwrite_default_bond_features: bool = False*) → *chemprop.features.featurization.BatchMolGraph*

Converts a list of SMILES or RDKit molecules to a `BatchMolGraph` containing the batch of molecular graphs.

Parameters

- **mols** – A list of SMILES or a list of RDKit molecules.
- **atom_features_batch** – A list of 2D numpy array containing additional atom features to featurize the molecule
- **bond_features_batch** – A list of 2D numpy array containing additional bond features to featurize the molecule
- **overwrite_default_atom_features** – Boolean to overwrite default atom descriptors by atom_descriptors instead of concatenating
- **overwrite_default_bond_features** – Boolean to overwrite default bond descriptors by bond_descriptors instead of concatenating

Returns A `BatchMolGraph` containing the combined molecular graph for the molecules.

chemprop.features.featurization.**onek_encoding_unk**(*value: int*, *choices: List[int]*) → List[int]

Creates a one-hot encoding with an extra category for uncommon values.

Parameters

- **value** – The value for which the encoding should be one.
- **choices** – A list of possible values.

Returns A one-hot encoding of the `value` in a list of length `len(choices) + 1`. If `value` is not in `choices`, then the final element in the encoding is 1.

chemprop.features.featurization.**reaction_mode**() → str

Returns the reaction mode

chemprop.features.featurization.**reset_featurization_parameters**(*logger: Optional[logging.Logger] = None*) → None

Function resets feature parameter values to defaults by replacing the parameters instance.

chemprop.features.featurization.**set_explicit_h**(*explicit_h: bool*) → None

Sets whether RDKit molecules will be constructed with explicit Hs.

Parameters **explicit_h** – Boolean whether to keep explicit Hs from input.

chemprop.features.featurization.**set_extra_atom_fdim**(*extra*)
Change the dimensionality of the atom feature vector.

chemprop.features.featurization.**set_extra_bond_fdim**(*extra*)
Change the dimensionality of the bond feature vector.

chemprop.features.featurization.**set_reaction**(*reaction: bool*, *mode: str*) → None
Sets whether to use a reaction or molecule as input and adapts feature dimensions.

Parameters

- **reaction** – Boolean whether to except reactions as input.
- **mode** – Reaction mode to construct atom and bond feature vectors.

6.2 Features Generators

Classes and functions from chemprop.features.features_generators.py. Features generators are used for computing additional molecule-level features that are appended after message passing.

chemprop.features.features_generators.**get_available_features_generators**() → List[str]
Returns a list of names of available features generators.

chemprop.features.features_generators.**get_features_generator**(*features_generator_name: str*) → Callable[[Union[str, rdkit.Chem.rdchem.Mol]], numpy.ndarray]
Gets a registered features generator by name.

Parameters **features_generator_name** – The name of the features generator.

Returns The desired features generator.

chemprop.features.features_generators.**morgan_binary_features_generator**(*mol: Union[str, rdkit.Chem.rdchem.Mol]*, *radius: int = 2*, *num_bits: int = 2048*) → numpy.ndarray
Generates a binary Morgan fingerprint for a molecule.

Parameters

- **mol** – A molecule (i.e., either a SMILES or an RDKit molecule).
- **radius** – Morgan fingerprint radius.
- **num_bits** – Number of bits in Morgan fingerprint.

Returns A 1D numpy array containing the binary Morgan fingerprint.

chemprop.features.features_generators.**morgan_counts_features_generator**(*mol: Union[str, rdkit.Chem.rdchem.Mol]*, *radius: int = 2*, *num_bits: int = 2048*) → numpy.ndarray
Generates a counts-based Morgan fingerprint for a molecule.

Parameters

- **mol** – A molecule (i.e., either a SMILES or an RDKit molecule).
- **radius** – Morgan fingerprint radius.

- **num_bits** – Number of bits in Morgan fingerprint.

Returns A 1D numpy array containing the counts-based Morgan fingerprint.

chemprop.features.features_generators.**rdkit_2d_features_generator**(*mol: Union[str, rdkit.Chem.rdchem.Mol]*) → numpy.ndarray

Generates RDKit 2D features for a molecule.

Parameters **mol** – A molecule (i.e., either a SMILES or an RDKit molecule).

Returns A 1D numpy array containing the RDKit 2D features.

chemprop.features.features_generators.**rdkit_2d_normalized_features_generator**(*mol: Union[str, rdkit.Chem.rdchem.Mol]*) → numpy.ndarray

Generates RDKit 2D normalized features for a molecule.

Parameters **mol** – A molecule (i.e., either a SMILES or an RDKit molecule).

Returns A 1D numpy array containing the RDKit 2D normalized features.

chemprop.features.features_generators.**register_features_generator**(*features_generator_name: str*) → Callable[[Callable[[Union[str, rdkit.Chem.rdchem.Mol]], numpy.ndarray]], Callable[[Union[str, rdkit.Chem.rdchem.Mol]], numpy.ndarray]]

Creates a decorator which registers a features generator in a global dictionary to enable access by name.

Parameters **features_generator_name** – The name to use to access the features generator.

Returns A decorator which will add a features generator to the registry using the specified name.

6.3 Utils

Classes and functions from chemprop.features.utils.py.

chemprop.features.utils.**load_features**(*path: str*) → numpy.ndarray

Loads features saved in a variety of formats.

Supported formats:

- `.npz` compressed (assumes features are saved with name "features")
- .npy
- `.csv` / `.txt` (assumes comma-separated features with a header and with one line per molecule)
- `.pkl` / `.pckl` / `.pickle` containing a sparse numpy array

Note: All formats assume that the SMILES loaded elsewhere in the code are in the same order as the features loaded here.

Parameters **path** – Path to a file containing features.

Returns A 2D numpy array of size `(num_molecules, features_size)` containing the features.

chemprop.features.utils.**load_valid_atom_or_bond_features**(*path: str*, *smiles: List[str]*) → List[numpy.ndarray]

Loads features saved in a variety of formats.

Supported formats:

- `.npz` descriptors are saved as 2D array for each molecule in the order of that in the data.csv
- `.pkl` / `.pckl` / `.pickle` containing a pandas dataframe with smiles as index and numpy array of descriptors as columns
- :code:'.sdf' containing all mol blocks with descriptors as entries

Parameters **path** – Path to file containing atomwise features.

Returns A list of 2D array.

chemprop.features.utils.**save_features**(*path: str*, *features: List[numpy.ndarray]*) → None

Saves features to a compressed `.npz` file with array name "features".

Parameters

- **path** – Path to a `.npz` file where the features will be saved.
- **features** – A list of 1D numpy arrays containing the features for molecules.

CHAPTER

SEVEN

MODELS

chemprop.models.py contains the core Chemprop message passing neural network.

7.1 Model

chemprop.models.model.py contains the *MoleculeModel* class, which contains the full Chemprop model. It consists of an *MPN*, which performs message passing, along with a feed-forward neural network which combines the output of the message passing network along with any additional molecule-level features and makes the final property predictions.

class chemprop.models.model.**MoleculeModel**(*args:* chemprop.args.TrainArgs)

A *MoleculeModel* is a model which contains a message passing network following by feed-forward layers.

Parameters **args** – A *TrainArgs* object containing model arguments.

create_encoder(*args:* chemprop.args.TrainArgs) → None

Creates the message passing encoder for the model.

Parameters **args** – A *TrainArgs* object containing model arguments.

create_ffn(*args:* chemprop.args.TrainArgs) → None

Creates the feed-forward layers for the model.

Parameters **args** – A *TrainArgs* object containing model arguments.

fingerprint(*batch: Union[List[List[str]], List[List[rdkit.Chem.rdchem.Mol]], List[List[Tuple[rdkit.Chem.rdchem.Mol, rdkit.Chem.rdchem.Mol]]], List[*chemprop.features.featurization.BatchMolGraph*]], features_batch: Optional[List[numpy.ndarray]] = None, atom_descriptors_batch: Optional[List[numpy.ndarray]] = None, atom_features_batch: Optional[List[numpy.ndarray]] = None, bond_features_batch: Optional[List[numpy.ndarray]] = None, fingerprint_type='MPN'*) → torch.FloatTensor

Encodes the latent representations of the input molecules from intermediate stages of the model.

Parameters

- **batch** – A list of list of SMILES, a list of list of RDKit molecules, or a list of *BatchMolGraph*. The outer list or BatchMolGraph is of length `num_molecules` (number of datapoints in batch), the inner list is of length `number_of_molecules` (number of molecules per datapoint).
- **features_batch** – A list of numpy arrays containing additional features.
- **atom_descriptors_batch** – A list of numpy arrays containing additional atom descriptors.

- **fingerprint_type** – The choice of which type of latent representation to return as the molecular fingerprint. Currently supported MPN for the output of the MPNN portion of the model or last_FFN for the input to the final readout layer.

Returns The latent fingerprint vectors.

forward(*batch: Union[List[List[str]], List[List[rdkit.Chem.rdchem.Mol]], List[List[Tuple[rdkit.Chem.rdchem.Mol, rdkit.Chem.rdchem.Mol]]], List[*chemprop.features.featurization.BatchMolGraph*]*, *features_batch: Optional[List[numpy.ndarray]] = None*, *atom_descriptors_batch: Optional[List[numpy.ndarray]] = None*, *atom_features_batch: Optional[List[numpy.ndarray]] = None*, *bond_features_batch: Optional[List[numpy.ndarray]] = None*) → torch.FloatTensor

Runs the *MoleculeModel* on input.

Parameters

- **batch** – A list of list of SMILES, a list of list of RDKit molecules, or a list of *BatchMolGraph*. The outer list or BatchMolGraph is of length `num_molecules` (number of datapoints in batch), the inner list is of length `number_of_molecules` (number of molecules per datapoint).
- **features_batch** – A list of numpy arrays containing additional features.
- **atom_descriptors_batch** – A list of numpy arrays containing additional atom descriptors.
- **atom_features_batch** – A list of numpy arrays containing additional atom features.
- **bond_features_batch** – A list of numpy arrays containing additional bond features.

Returns The output of the *MoleculeModel*, containing a list of property predictions

7.2 MPN

chemprop.models.model.py contains the *MPNEncoder* class, which is the core message passing network, along with a wrapper *MPN* which is used within a *MoleculeModel*.

class chemprop.models.mpn.**MPN**(*args:* chemprop.args.TrainArgs, *atom_fdim: Optional[int] = None*, *bond_fdim: Optional[int] = None*)

An *MPN* is a wrapper around *MPNEncoder* which featurizes input as needed.

Parameters

- **args** – A *TrainArgs* object containing model arguments.
- **atom_fdim** – Atom feature vector dimension.
- **bond_fdim** – Bond feature vector dimension.

forward(*batch: Union[List[List[str]], List[List[rdkit.Chem.rdchem.Mol]], List[List[Tuple[rdkit.Chem.rdchem.Mol, rdkit.Chem.rdchem.Mol]]], List[*chemprop.features.featurization.BatchMolGraph*]*, *features_batch: Optional[List[numpy.ndarray]] = None*, *atom_descriptors_batch: Optional[List[numpy.ndarray]] = None*, *atom_features_batch: Optional[List[numpy.ndarray]] = None*, *bond_features_batch: Optional[List[numpy.ndarray]] = None*) → torch.FloatTensor

Encodes a batch of molecules.

Parameters

- **batch** – A list of list of SMILES, a list of list of RDKit molecules, or a list of *BatchMolGraph*. The outer list or BatchMolGraph is of length `num_molecules` (number of datapoints in batch), the inner list is of length `number_of_molecules` (number of molecules per datapoint).
- **features_batch** – A list of numpy arrays containing additional features.
- **atom_descriptors_batch** – A list of numpy arrays containing additional atom descriptors.
- **atom_features_batch** – A list of numpy arrays containing additional atom features.
- **bond_features_batch** – A list of numpy arrays containing additional bond features.

Returns A PyTorch tensor of shape `(num_molecules, hidden_size)` containing the encoding of each molecule.

class chemprop.models.mpn.**MPNEncoder**(*args:* chemprop.args.TrainArgs, *atom_fdim: int*, *bond_fdim: int*)

An *MPNEncoder* is a message passing neural network for encoding a molecule.

Parameters

- **args** – A *TrainArgs* object containing model arguments.
- **atom_fdim** – Atom feature vector dimension.
- **bond_fdim** – Bond feature vector dimension.

forward(*mol_graph:* chemprop.features.featurization.BatchMolGraph, *atom_descriptors_batch: Optional[List[numpy.ndarray]] = None*) → torch.FloatTensor

Encodes a batch of molecular graphs.

Parameters

- **mol_graph** – A *BatchMolGraph* representing a batch of molecular graphs.
- **atom_descriptors_batch** – A list of numpy arrays containing additional atomic descriptors

Returns A PyTorch tensor of shape `(num_molecules, hidden_size)` containing the encoding of each molecule.

CHAPTER

EIGHT

TRAINING AND PREDICTING

chemprop.train contains functions to train and make predictions with message passing neural networks.

8.1 Train

chemprop.train.train.py trains a model for a single epoch.

chemprop.train.train.**train**(*model:* chemprop.models.model.MoleculeModel, *data_loader:* chemprop.data.data.MoleculeDataLoader, *loss_func: Callable*, *optimizer:* *torch.optim.optimizer.Optimizer*, *scheduler:* *torch.optim.lr_scheduler._LRScheduler*, *args:* chemprop.args.TrainArgs, *n_iter:* *int = 0*, *logger: Optional[logging.Logger] = None*, *writer:* *Optional[tensorboardX.writer.SummaryWriter] = None*) → int

Trains a model for an epoch.

Parameters

- **model** – A *MoleculeModel*.
- **data_loader** – A *MoleculeDataLoader*.
- **loss_func** – Loss function.
- **optimizer** – An optimizer.
- **scheduler** – A learning rate scheduler.
- **args** – A *TrainArgs* object containing arguments for training the model.
- **n_iter** – The number of iterations (training examples) trained on so far.
- **logger** – A logger for recording output.
- **writer** – A tensorboardX SummaryWriter.

Returns The total number of iterations (training examples) trained on so far.

8.2 Run Training

chemprop.train.run_training.py loads data, initializes the model, and runs training, validation, and testing of the model.

chemprop.train.run_training.**run_training**(*args:* chemprop.args.TrainArgs, *data:* chemprop.data.data.MoleculeDataset, *logger:* *Optional[logging.Logger] = None*) → Dict[str, List[float]]

Loads data, trains a Chemprop model, and returns test scores for the model checkpoint with the highest validation score.

Parameters

- **args** – A *TrainArgs* object containing arguments for loading data and training the Chemprop model.
- **data** – A `MoleculeDataset` containing the data.
- **logger** – A logger to record output.

Returns A dictionary mapping each metric in `args.metrics` to a list of values for each task.

8.3 Cross-Validation

chemprop.train.cross_validate.py provides an outer loop around chemprop.train.run_training.py that runs training and evaluating for each of several splits of the data.

chemprop.train.cross_validate.**chemprop_train**() → None

Parses Chemprop training arguments and trains (cross-validates) a Chemprop model.

This is the entry point for the command line command `chemprop_train`.

chemprop.train.cross_validate.**cross_validate**(*args:* chemprop.args.TrainArgs, *train_func:* *Callable[[*chemprop.args.TrainArgs, chemprop.data.data.MoleculeDataset, *logging.Logger], Dict[str, List[float]]]*) → Tuple[float, float]

Runs k-fold cross-validation.

For each of k splits (folds) of the data, trains and tests a model on that split and aggregates the performance across folds.

Parameters

- **args** – A *TrainArgs* object containing arguments for loading data and training the Chemprop model.
- **train_func** – Function which runs training.

Returns A tuple containing the mean and standard deviation performance across folds.

8.4 Predict

chemprop.train.predict.py uses a trained model to make predicts on data.

chemprop.train.predict.**predict**(*model:* chemprop.models.model.MoleculeModel, *data_loader:* chemprop.data.data.MoleculeDataLoader, *disable_progress_bar: bool = False*, *scaler: Optional[*chemprop.data.scaler.StandardScaler*] = None*) → List[List[float]]

Makes predictions on a dataset using an ensemble of models.

Parameters

- **model** – A *MoleculeModel*.
- **data_loader** – A *MoleculeDataLoader*.
- **disable_progress_bar** – Whether to disable the progress bar.
- **scaler** – A StandardScaler object fit on the training targets.

Returns A list of lists of predictions. The outer list is molecules while the inner list is tasks.

8.5 Make Predictions

chemprop.train.make_predictions.py is a wrapper aoround chemprop.train.predict.py which loads data, loads a trained model, makes predictions, and saves those predictions.

chemprop.train.make_predictions.**chemprop_predict**() → None

Parses Chemprop predicting arguments and runs prediction using a trained Chemprop model.

This is the entry point for the command line command chemprop_predict.

chemprop.train.make_predictions.**load_data**(*args:* chemprop.args.PredictArgs, *smiles: List[List[str]]*)

Function to load data from a list of smiles or a file.

Parameters

- **args** – A *PredictArgs* object containing arguments for loading data and a model and making predictions.
- **smiles** – A list of list of smiles, or None if data is to be read from file

Returns A tuple of a MoleculeDataset containing all datapoints, a MoleculeDataset containing only valid datapoints, a MoleculeDataLoader and a dictionary mapping full to valid indices.

chemprop.train.make_predictions.**load_model**(*args:* chemprop.args.PredictArgs, *generator: bool = False*)

Function to load a model or ensemble of models from file. If generator is True, a generator of the respective model and scaler objects is returned (memory efficient), else the full list (holding all models in memory, necessary for preloading).

Parameters

- **args** – A *PredictArgs* object containing arguments for loading data and a model and making predictions.
- **generator** – A boolean to return a generator instead of a list of models and scalers.

Returns A tuple of updated prediction arguments, training arguments, a list or generator object of models, a list or generator object of scalers, the number of tasks and their respective names.

chemprop.train.make_predictions.**make_predictions**(*args:* chemprop.args.PredictArgs, *smiles: List[List[str]] = None*, *model_objects: Tuple[*chemprop.args.PredictArgs, chemprop.args.TrainArgs, *List[*chemprop.models.model.MoleculeModel*], List[*chemprop.data.scaler.StandardScaler*], int, List[str]] = None*) → List[List[Optional[float]]]

Loads data and a trained model and uses the model to make predictions on the data.

If SMILES are provided, then makes predictions on smiles. Otherwise makes predictions on `args.test_data`.

Parameters

- **args** – A *PredictArgs* object containing arguments for loading data and a model and making predictions.
- **smiles** – List of list of SMILES to make predictions on.
- **model_objects** – Tuple of output of load_model function which can be called separately.

Returns A list of lists of target predictions.

chemprop.train.make_predictions.**predict_and_save**(*args:* chemprop.args.PredictArgs, *train_args:* chemprop.args.TrainArgs, *test_data:* chemprop.data.data.MoleculeDataset, *task_names: List[str]*, *num_tasks: int*, *test_data_loader:* chemprop.data.data.MoleculeDataLoader, *full_data:* chemprop.data.data.MoleculeDataset, *full_to_valid_indices: dict*, *models: List[*chemprop.models.model.MoleculeModel*]*, *scalers: List[List[*chemprop.data.scaler.StandardScaler*]]*)

Function to predict with a model and save the predictions to file.

Parameters

- **args** – A *PredictArgs* object containing arguments for loading data and a model and making predictions.
- **train_args** – A *TrainArgs* object containing arguments for training the model.
- **test_data** – A `MoleculeDataset` containing valid datapoints.
- **task_names** – A list of task names.
- **num_tasks** – Number of tasks.
- **test_data_loader** – A `MoleculeDataLoader` to load the test data.
- **full_data** – A `MoleculeDataset` containing all (valid and invalid) datapoints.
- **full_to_valid_indices** – A dictionary dictionary mapping full to valid indices.
- **models** – A list or generator object of `MoleculeModels`.
- **scalers** – A list or generator object of `StandardScaler` objects.

Returns A list of lists of target predictions.

chemprop.train.make_predictions.**set_features**(*args:* chemprop.args.PredictArgs, *train_args:* chemprop.args.TrainArgs)

Function to set extra options.

Parameters

- **args** – A *PredictArgs* object containing arguments for loading data and a model and making predictions.
- **train_args** – A *TrainArgs* object containing arguments for training the model.

8.6 Evaluate

chemprop.train.evaluate.py contains functions for evaluating the quality of predictions by comparing them to the true values.

chemprop.train.evaluate.**evaluate**(*model:* chemprop.models.model.MoleculeModel, *data_loader:* chemprop.data.data.MoleculeDataLoader, *num_tasks: int*, *metrics: List[str]*, *dataset_type: str*, *scaler: Optional[*chemprop.data.scaler.StandardScaler*] = None*, *logger: Optional[logging.Logger] = None*) → Dict[str, List[float]]

Evaluates an ensemble of models on a dataset by making predictions and then evaluating the predictions.

Parameters

- **model** – A *MoleculeModel*.
- **data_loader** – A *MoleculeDataLoader*.
- **num_tasks** – Number of tasks.
- **metrics** – A list of names of metric functions.
- **dataset_type** – Dataset type.
- **scaler** – A `StandardScaler` object fit on the training targets.
- **logger** – A logger to record output.

Returns A dictionary mapping each metric in `metrics` to a list of values for each task.

chemprop.train.evaluate.**evaluate_predictions**(*preds: List[List[float]]*, *targets: List[List[float]]*, *num_tasks: int*, *metrics: List[str]*, *dataset_type: str*, *logger: Optional[logging.Logger] = None*) → Dict[str, List[float]]

Evaluates predictions using a metric function after filtering out invalid targets.

Parameters

- **preds** – A list of lists of shape `(data_size, num_tasks)` with model predictions.
- **targets** – A list of lists of shape `(data_size, num_tasks)` with targets.
- **num_tasks** – Number of tasks.
- **metrics** – A list of names of metric functions.
- **dataset_type** – Dataset type.
- **logger** – A logger to record output.

Returns A dictionary mapping each metric in `metrics` to a list of values for each task.

CHAPTER

NINE

HYPERPARAMETER OPTIMIZATION

chemprop.hyperparameter_optimization.py runs hyperparameter optimization on Chemprop models.

Optimizes hyperparameters using Bayesian optimization.

chemprop.hyperparameter_optimization.**chemprop_hyperopt**() → None
Runs hyperparameter optimization for a Chemprop model.

This is the entry point for the command line command `chemprop_hyperopt`.

chemprop.hyperparameter_optimization.**hyperopt**(*args:* chemprop.args.HyperoptArgs) → None
Runs hyperparameter optimization on a Chemprop model.

Hyperparameter optimization optimizes the following parameters:

- `hidden_size`: The hidden size of the neural network layers is selected from {300, 400, …, 2400}
- `depth`: The number of message passing iterations is selected from {2, 3, 4, 5, 6}
- `dropout`: The dropout probability is selected from {0.0, 0.05, …, 0.4}
- `ffn_num_layers`: The number of feed-forward layers after message passing is selected from {1, 2, 3}

The best set of hyperparameters is saved as a JSON file to `args.config_save_path`.

Parameters **`args`** – A *HyperoptArgs* object containing arguments for hyperparameter optimization in addition to all arguments needed for training.

CHAPTER

TEN

INTERPRETATION

chemprop.interpret.py uses a Monte Carlo Tree Search to interpret trained Chemprop models by identifying substructures of a molecule which are primarily responsible for Chemprop's prediction.

class chemprop.interpret.**ChempropModel**(*args:* chemprop.args.InterpretArgs)
A *ChempropModel* is a wrapper around a *MoleculeModel* for interpretation.

Parameters **args** – A *InterpretArgs* object containing arguments for interpretation.

class chemprop.interpret.**MCTSNode**(*smiles: str*, *atoms: List[int]*, *W: float = 0*, *N: int = 0*, *P: float = 0*)
A *MCTSNode* represents a node in a Monte Carlo Tree Search.

Parameters

- **smiles** – The SMILES for the substructure at this node.
- **atoms** – A list of atom indices represented by this node.
- **W** – The W value of this node.
- **N** – The N value of this node.
- **P** – The P value of this node.

chemprop.interpret.**chemprop_interpret**() → None
Runs interpretation of a Chemprop model.

This is the entry point for the command line command `chemprop_interpret`.

chemprop.interpret.**extract_subgraph**(*smiles: str*, *selected_atoms: Set[int]*) → Tuple[str, List[int]]
Extracts a subgraph from a SMILES given a set of atom indices.

Parameters

- **smiles** – A SMILES from which to extract a subgraph.
- **selected_atoms** – The atoms which form the subgraph to be extracted.

Returns A tuple containing a SMILES representing the subgraph and a list of root atom indices from the selected indices.

chemprop.interpret.**find_clusters**(*mol: rdkit.Chem.rdchem.Mol*) → Tuple[List[Tuple[int, ...]], List[List[int]]]
Finds clusters within the molecule.

Parameters **mol** – An RDKit molecule.

Returns A tuple containing a list of atom tuples representing the clusters and a list of lists of atoms in each cluster.

chemprop.interpret.**interpret**(*args:* chemprop.args.InterpretArgs) → None
Runs interpretation of a Chemprop model using the Monte Carlo Tree Search algorithm.

Parameters **args** – A *InterpretArgs* object containing arguments for interpretation.

chemprop.interpret.**mcts**(*smiles: str*, *scoring_function: Callable[[List[str]], List[float]]*, *n_rollout: int*, *max_atoms: int*, *prop_delta: float*) → List[*chemprop.interpret.MCTSNode*]

Runs the Monte Carlo Tree Search algorithm.

Parameters

- **smiles** – The SMILES of the molecule to perform the search on.
- **scoring_function** – A function for scoring subgraph SMILES using a Chemprop model.
- **n_rollout** – THe number of MCTS rollouts to perform.
- **max_atoms** – The maximum number of atoms allowed in an extracted rationale.
- **prop_delta** – The minimum required property value for a satisfactory rationale.

Returns A list of rationales each represented by a *MCTSNode*.

chemprop.interpret.**mcts_rollout**(*node:* chemprop.interpret.MCTSNode, *state_map: Dict[str,* chemprop.interpret.MCTSNode*]*, *orig_smiles: str*, *clusters: List[Set[int]]*, *atom_cls: List[Set[int]]*, *nei_cls: List[Set[int]]*, *scoring_function: Callable[[List[str]], List[float]]*) → float

A Monte Carlo Tree Search rollout from a given *MCTSNode*.

Parameters

- **node** – The *MCTSNode* from which to begin the rollout.
- **state_map** – A mapping from SMILES to *MCTSNode*.
- **orig_smiles** – The original SMILES of the molecule.
- **clusters** – Clusters of atoms.
- **atom_cls** – Atom indices in the clusters.
- **nei_cls** – Neighboring clusters.
- **scoring_function** – A function for scoring subgraph SMILES using a Chemprop model.

Returns The score of this MCTS rollout.

CHAPTER

ELEVEN

COMMAND LINE ARGUMENTS

chemprop.args.py contains all command line arguments, which are processed using the Typed Argument Parser (`Tap`) package.

11.1 Common Arguments

class chemprop.args.**CommonArgs**(**args*, ***kwargs*)

CommonArgs contains arguments that are used in both *TrainArgs* and *PredictArgs*.

Initializes the Tap instance.

Parameters

- **args** – Arguments passed to the super class ArgumentParser.
- **underscores_to_dashes** – If True, convert underscores in flags to dashes.
- **explicit_bool** – Booleans can be specified on the command line as "–arg True" or "–arg False" rather than "–arg". Additionally, booleans can be specified by prefixes of True and False with any capitalization as well as 1 or 0.
- **config_files** – A list of paths to configuration files containing the command line arguments (e.g., '–arg1 a1 –arg2 a2'). Arguments passed in from the command line overwrite arguments from the configuration files. Arguments in configuration files that appear later in the list overwrite the arguments in previous configuration files.
- **kwargs** – Keyword arguments passed to the super class ArgumentParser.

atom_descriptors: Literal['feature', 'descriptor'] = None

Custom extra atom descriptors. `feature`: used as atom features to featurize a given molecule. `descriptor`: used as descriptor and concatenated to the machine learned atomic representation.

atom_descriptors_path: str = None

Path to the extra atom descriptors.

property atom_descriptors_size: int

The size of the atom descriptors.

property atom_features_size: int

The size of the atom features.

batch_size: int = 50

Batch size.

bond_features_path: str = None

Path to the extra bond descriptors that will be used as bond features to featurize a given molecule.

property bond_features_size: int
The size of the atom features.

checkpoint_dir: str = None
Directory from which to load model checkpoints (walks directory and ensembles all models that are found).

checkpoint_path: str = None
Path to model checkpoint (`.pt` file).

checkpoint_paths: List[str] = None
List of paths to model checkpoints (`.pt` files).

configure() → None
Overwrite this method to configure the parser during initialization.

For example,

self.add_argument('–sum', dest='accumulate', action='store_const', const=sum, default=max)

self.add_subparsers(help='sub-command help') self.add_subparser('a', SubparserA, help='a help')

property cuda: bool
Whether to use CUDA (i.e., GPUs) or not.

property device: torch.device
The `torch.device` on which to load and process data and models.

empty_cache: bool = False
Whether to empty all caches before training or predicting. This is necessary if multiple jobs are run within a single script and the atom or bond features change.

features_generator: List[str] = None
Method(s) of generating additional features.

features_path: List[str] = None
Path(s) to features to use in FNN (instead of features_generator).

property features_scaling: bool
Whether to apply normalization with a *StandardScaler* to the additional molecule-level features.

gpu: int = None
Which GPU to use.

max_data_size: int = None
Maximum number of data points to load.

no_cache_mol: bool = False
Whether to not cache the RDKit molecule for each SMILES string to reduce memory usage (cached by default).

no_cuda: bool = False
Turn off cuda (i.e., use CPU instead of GPU).

no_features_scaling: bool = False
Turn off scaling of features.

num_workers: int = 8
Number of workers for the parallel data loading (0 means sequential).

number_of_molecules: int = 1
Number of molecules in each input to the model. This must equal the length of `smiles_columns` (if not `None`).

phase_features_path: str = None
Path to features used to indicate the phase of the data in one-hot vector form. Used in spectra datatype.

process_args() → None
Perform additional argument processing and/or validation.

smiles_columns: List[str] = None
List of names of the columns containing SMILES strings. By default, uses the first `number_of_molecules` columns.

11.2 Train Arguments

class chemprop.args.**TrainArgs**(**args*, ***kwargs*)
TrainArgs includes *CommonArgs* along with additional arguments used for training a Chemprop model.

Initializes the Tap instance.

Parameters

- **args** – Arguments passed to the super class ArgumentParser.
- **underscores_to_dashes** – If True, convert underscores in flags to dashes.
- **explicit_bool** – Booleans can be specified on the command line as "–arg True" or "–arg False" rather than "–arg". Additionally, booleans can be specified by prefixes of True and False with any capitalization as well as 1 or 0.
- **config_files** – A list of paths to configuration files containing the command line arguments (e.g., '–arg1 a1 –arg2 a2'). Arguments passed in from the command line overwrite arguments from the configuration files. Arguments in configuration files that appear later in the list overwrite the arguments in previous configuration files.
- **kwargs** – Keyword arguments passed to the super class ArgumentParser.

activation: Literal['ReLU', 'LeakyReLU', 'PReLU', 'tanh', 'SELU', 'ELU'] = 'ReLU'
Activation function.

aggregation: Literal['mean', 'sum', 'norm'] = 'mean'
Aggregation scheme for atomic vectors into molecular vectors

aggregation_norm: int = 100
For norm aggregation, number by which to divide summed up atomic features

alternative_loss_function: Literal['wasserstein'] = None
Option to replace the default loss function, with an alternative. Only currently applied for spectra data type and wasserstein loss.

property atom_descriptor_scaling: bool
Whether to apply normalization with a *StandardScaler* to the additional atom features."

atom_messages: bool = False
Centers messages on atoms instead of on bonds.

bias: bool = False
Whether to add bias to linear layers.

property bond_feature_scaling: bool
Whether to apply normalization with a *StandardScaler* to the additional bond features."

cache_cutoff: float = 10000
Maximum number of molecules in dataset to allow caching. Below this number, caching is used and data

loading is sequential. Above this number, caching is not used and data loading is parallel. Use "inf" to always cache.

checkpoint_frzn: str = None
Path to model checkpoint file to be loaded for overwriting and freezing weights.

class_balance: bool = False
Trains with an equal number of positives and negatives in each batch.

config_path: str = None
Path to a `.json` file containing arguments. Any arguments present in the config file will override arguments specified via the command line or by the defaults.

crossval_index_dir: str = None
Directory in which to find cross validation index files.

crossval_index_file: str = None
Indices of files to use as train/val/test. Overrides `--num_folds` and `--seed`.

property crossval_index_sets: List[List[List[int]]]
Index sets used for splitting data into train/validation/test during cross-validation

data_path: str
Path to data CSV file.

data_weights_path: str = None
Path to weights for each molecule in the training data, affecting the relative weight of molecules in the loss function

dataset_type: Literal['regression', 'classification', 'multiclass', 'spectra']
Type of dataset. This determines the loss function used during training.

depth: int = 3
Number of message passing steps.

dropout: float = 0.0
Dropout probability.

ensemble_size: int = 1
Number of models in ensemble.

epochs: int = 30
Number of epochs to run.

explicit_h: bool = False
Whether H are explicitly specified in input (and should be kept this way).

extra_metrics: List[Literal['auc', 'prc-auc', 'rmse', 'mae', 'mse', 'r2', 'accuracy', 'cross_entropy', 'binary_cross_entropy', 'sid', 'wasserstein']] = []
Additional metrics to use to evaluate the model. Not used for early stopping.

features_only: bool = False
Use only the additional features in an FFN, no graph network.

property features_size: int
The dimensionality of the additional molecule-level features.

ffn_hidden_size: int = None
Hidden dim for higher-capacity FFN (defaults to hidden_size).

ffn_num_layers: int = 2
Number of layers in FFN after MPN encoding.

final_lr: float = 0.0001
Final learning rate.

folds_file: str = None
Optional file of fold labels.

freeze_first_only: bool = False
Determines whether or not to use checkpoint_frzn for just the first encoder. Default (False) is to use the checkpoint to freeze all encoders. (only relevant for number_of_molecules > 1, where checkpoint model has number_of_molecules = 1)

frzn_ffn_layers: int = 0
Overwrites weights for the first n layers of the ffn from checkpoint model (specified checkpoint_frzn), where n is specified in the input. Automatically also freezes mpnn weights.

grad_clip: float = None
Maximum magnitude of gradient during training.

hidden_size: int = 300
Dimensionality of hidden layers in MPN.

ignore_columns: List[str] = None
Name of the columns to ignore when `target_columns` is not provided.

init_lr: float = 0.0001
Initial learning rate.

log_frequency: int = 10
The number of batches between each logging of the training loss.

max_lr: float = 0.001
Maximum learning rate.

metric: Literal['auc', 'prc-auc', 'rmse', 'mae', 'mse', 'r2', 'accuracy', 'cross_entropy', 'binary_cross_entropy', 'sid', 'wasserstein'] = None
Metric to use during evaluation. It is also used with the validation set for early stopping. Defaults to "auc" for classification, "rmse" for regression, and "sid" for spectra.

property metrics: List[str]
The list of metrics used for evaluation. Only the first is used for early stopping.

property minimize_score: bool
Whether the model should try to minimize the score metric or maximize it.

mpn_shared: bool = False
Whether to use the same message passing neural network for all input molecules Only relevant if `number_of_molecules > 1`

multiclass_num_classes: int = 3
Number of classes when running multiclass classification.

no_atom_descriptor_scaling: bool = False
Turn off atom feature scaling.

no_bond_features_scaling: bool = False
Turn off atom feature scaling.

num_folds: int = 1
Number of folds when performing cross validation.

property num_lrs: int
The number of learning rates to use (currently hard-coded to 1).

property num_tasks: int
The number of tasks being trained on.

overwrite_default_atom_features: bool = False
Overwrites the default atom descriptors with the new ones instead of concatenating them. Can only be used if atom_descriptors are used as a feature.

overwrite_default_bond_features: bool = False
Overwrites the default atom descriptors with the new ones instead of concatenating them

process_args() → None
Perform additional argument processing and/or validation.

pytorch_seed: int = 0
Seed for PyTorch randomness (e.g., random initial weights).

quiet: bool = False
Skip non-essential print statements.

reaction: bool = False
Whether to adjust MPNN layer to take reactions as input instead of molecules.

reaction_mode: Literal['reac_prod', 'reac_diff', 'prod_diff', 'reac_prod_balance', 'reac_diff_balance', 'prod_diff_balance'] = 'reac_diff'
Choices for construction of atom and bond features for reactions `reac_prod`: concatenates the reactants feature with the products feature. `reac_diff`: concatenates the reactants feature with the difference in features between reactants and products. `prod_diff`: concatenates the products feature with the difference in features between reactants and products. `reac_prod_balance`: concatenates the reactants feature with the products feature, balances imbalanced reactions. `reac_diff_balance`: concatenates the reactants feature with the difference in features between reactants and products, balances imbalanced reactions. `prod_diff_balance`: concatenates the products feature with the difference in features between reactants and products, balances imbalanced reactions.

resume_experiment: bool = False
Whether to resume the experiment. Loads test results from any folds that have already been completed and skips training those folds.

save_dir: str = None
Directory where model checkpoints will be saved.

save_preds: bool = False
Whether to save test split predictions during training.

save_smiles_splits: bool = False
Save smiles for each train/val/test splits for prediction convenience later.

seed: int = 0
Random seed to use when splitting data into train/val/test sets. When :code`num_folds > 1`, the first fold uses this seed and all subsequent folds add 1 to the seed.

separate_test_atom_descriptors_path: str = None
Path to file with extra atom descriptors for separate test set.

separate_test_bond_features_path: str = None
Path to file with extra atom descriptors for separate test set.

separate_test_features_path: List[str] = None
Path to file with features for separate test set.

separate_test_path: str = None
Path to separate test set, optional.

separate_test_phase_features_path: str = None
Path to file with phase features for separate test set.

separate_val_atom_descriptors_path: str = None
Path to file with extra atom descriptors for separate val set.

separate_val_bond_features_path: str = None
Path to file with extra atom descriptors for separate val set.

separate_val_features_path: List[str] = None
Path to file with features for separate val set.

separate_val_path: str = None
Path to separate val set, optional.

separate_val_phase_features_path: str = None
Path to file with phase features for separate val set.

show_individual_scores: bool = False
Show all scores for individual targets, not just average, at the end.

spectra_activation: Literal['exp', 'softplus'] = 'exp'
Indicates which function to use in dataset_type spectra training to constrain outputs to be positive.

spectra_phase_mask_path: str = None
Path to a file containing a phase mask array, used for excluding particular regions in spectra predictions.

spectra_target_floor: float = 1e-08
Values in targets for dataset type spectra are replaced with this value, intended to be a small positive number used to enforce positive values.

split_sizes: Tuple[float, float, float] = (0.8, 0.1, 0.1)
Split proportions for train/validation/test sets.

split_type: Literal['random', 'scaffold_balanced', 'predetermined', 'crossval', 'cv', 'cv-no-test', 'index_predetermined', 'random_with_repeated_smiles'] = 'random'
Method of splitting the data into train/val/test.

target_columns: List[str] = None
Name of the columns containing target values. By default, uses all columns except the SMILES column and the `ignore_columns`.

target_weights: List[float] = None
Weights associated with each target, affecting the relative weight of targets in the loss function. Must match the number of target columns.

property task_names: List[str]
A list of names of the tasks being trained on.

test: bool = False
Whether to skip training and only test the model.

test_fold_index: int = None
Which fold to use as test for leave-one-out cross val.

property train_data_size: int
The size of the training data set.

undirected: bool = False
Undirected edges (always sum the two relevant bond vectors).

property use_input_features: bool
Whether the model is using additional molecule-level features.

val_fold_index: int = None
Which fold to use as val for leave-one-out cross val.

warmup_epochs: float = 2.0
Number of epochs during which learning rate increases linearly from `init_lr` to `max_lr`. Afterwards, learning rate decreases exponentially from `max_lr` to `final_lr`.

11.3 Predict Arguments

class chemprop.args.**PredictArgs**(**args*, ***kwargs*)
PredictArgs includes *CommonArgs* along with additional arguments used for predicting with a Chemprop model.

Initializes the Tap instance.

Parameters

- **args** – Arguments passed to the super class ArgumentParser.
- **underscores_to_dashes** – If True, convert underscores in flags to dashes.
- **explicit_bool** – Booleans can be specified on the command line as "–arg True" or "–arg False" rather than "–arg". Additionally, booleans can be specified by prefixes of True and False with any capitalization as well as 1 or 0.
- **config_files** – A list of paths to configuration files containing the command line arguments (e.g., '–arg1 a1 –arg2 a2'). Arguments passed in from the command line overwrite arguments from the configuration files. Arguments in configuration files that appear later in the list overwrite the arguments in previous configuration files.
- **kwargs** – Keyword arguments passed to the super class ArgumentParser.

drop_extra_columns: bool = False
Whether to drop all columns from the test data file besides the SMILES columns and the new prediction columns.

property ensemble_size: int
The number of models in the ensemble.

ensemble_variance: bool = False
Whether to calculate the variance of ensembles as a measure of epistemic uncertainty. If True, the variance is saved as an additional column for each target in the preds_path.

individual_ensemble_predictions: bool = False
Whether to return the predictions made by each of the individual models rather than the average of the ensemble

preds_path: str
Path to CSV file where predictions will be saved.

process_args() → None
Perform additional argument processing and/or validation.

test_path: str
Path to CSV file containing testing data for which predictions will be made.

11.4 Interpret Arguments

class chemprop.args.**InterpretArgs**(**args*, ***kwargs*)

InterpretArgs includes *CommonArgs* along with additional arguments used for interpreting a trained Chemprop model.

Initializes the Tap instance.

Parameters

- **args** – Arguments passed to the super class ArgumentParser.
- **underscores_to_dashes** – If True, convert underscores in flags to dashes.
- **explicit_bool** – Booleans can be specified on the command line as "–arg True" or "–arg False" rather than "–arg". Additionally, booleans can be specified by prefixes of True and False with any capitalization as well as 1 or 0.
- **config_files** – A list of paths to configuration files containing the command line arguments (e.g., '–arg1 a1 –arg2 a2'). Arguments passed in from the command line overwrite arguments from the configuration files. Arguments in configuration files that appear later in the list overwrite the arguments in previous configuration files.
- **kwargs** – Keyword arguments passed to the super class ArgumentParser.

batch_size: int = 500
Batch size.

c_puct: float = 10.0
Constant factor in MCTS.

data_path: str
Path to data CSV file.

max_atoms: int = 20
Maximum number of atoms in rationale.

min_atoms: int = 8
Minimum number of atoms in rationale.

process_args() → None
Perform additional argument processing and/or validation.

prop_delta: float = 0.5
Minimum score to count as positive.

property_id: int = 1
Index of the property of interest in the trained model.

rollout: int = 20
Number of rollout steps.

11.5 Hyperparameter Optimization Arguments

class chemprop.args.**HyperoptArgs**(**args*, ***kwargs*)

HyperoptArgs includes *TrainArgs* along with additional arguments used for optimizing Chemprop hyperparameters.

Initializes the Tap instance.

Parameters

- **args** – Arguments passed to the super class ArgumentParser.
- **underscores_to_dashes** – If True, convert underscores in flags to dashes.
- **explicit_bool** – Booleans can be specified on the command line as "–arg True" or "–arg False" rather than "–arg". Additionally, booleans can be specified by prefixes of True and False with any capitalization as well as 1 or 0.
- **config_files** – A list of paths to configuration files containing the command line arguments (e.g., '–arg1 a1 –arg2 a2'). Arguments passed in from the command line overwrite arguments from the configuration files. Arguments in configuration files that appear later in the list overwrite the arguments in previous configuration files.
- **kwargs** – Keyword arguments passed to the super class ArgumentParser.

config_save_path: str
Path to `.json` file where best hyperparameter settings will be written.

hyperopt_checkpoint_dir: str = None
Path to a directory where hyperopt completed trial data is stored. Hyperopt job will include these trials if restarted. Can also be used to run multiple instances in parallel if they share the same checkpoint directory.

log_dir: str = None
(Optional) Path to a directory where all results of the hyperparameter optimization will be written.

manual_trial_dirs: List[str] = None
Paths to save directories for manually trained models in the same search space as the hyperparameter search. Results will be considered as part of the trial history of the hyperparameter search.

num_iters: int = 20
Number of hyperparameter choices to try.

process_args() → None
Perform additional argument processing and/or validation.

startup_random_iters: int = 10
The initial number of trials that will be randomly specified before TPE algorithm is used to select the rest.

11.6 Scikit-Learn Train Arguments

class chemprop.args.**SklearnTrainArgs**(**args*, ***kwargs*)

SklearnTrainArgs includes *TrainArgs* along with additional arguments for training a scikit-learn model.

Initializes the Tap instance.

Parameters

- **args** – Arguments passed to the super class ArgumentParser.
- **underscores_to_dashes** – If True, convert underscores in flags to dashes.

- **explicit_bool** – Booleans can be specified on the command line as "–arg True" or "–arg False" rather than "–arg". Additionally, booleans can be specified by prefixes of True and False with any capitalization as well as 1 or 0.
- **config_files** – A list of paths to configuration files containing the command line arguments (e.g., '–arg1 a1 –arg2 a2'). Arguments passed in from the command line overwrite arguments from the configuration files. Arguments in configuration files that appear later in the list overwrite the arguments in previous configuration files.
- **kwargs** – Keyword arguments passed to the super class ArgumentParser.

class_weight: Literal['balanced'] = None
How to weight classes (None means no class balance).

impute_mode: Literal['single_task', 'median', 'mean', 'linear', 'frequent'] = None
How to impute missing data (None means no imputation).

model_type: Literal['random_forest', 'svm']
scikit-learn model to use.

num_bits: int = 2048
Number of bits in morgan fingerprint.

num_trees: int = 500
Number of random forest trees.

radius: int = 2
Morgan fingerprint radius.

single_task: bool = False
Whether to run each task separately (needed when dataset has null entries).

11.7 Scikit-Learn Predict Arguments

class chemprop.args.**SklearnPredictArgs**(**args*, *underscores_to_dashes: bool = False*, *explicit_bool: bool = False*, *config_files: Optional[List[str]] = None*, ***kwargs*)

SklearnPredictArgs contains arguments used for predicting with a trained scikit-learn model.

Initializes the Tap instance.

Parameters

- **args** – Arguments passed to the super class ArgumentParser.
- **underscores_to_dashes** – If True, convert underscores in flags to dashes.
- **explicit_bool** – Booleans can be specified on the command line as "–arg True" or "–arg False" rather than "–arg". Additionally, booleans can be specified by prefixes of True and False with any capitalization as well as 1 or 0.
- **config_files** – A list of paths to configuration files containing the command line arguments (e.g., '–arg1 a1 –arg2 a2'). Arguments passed in from the command line overwrite arguments from the configuration files. Arguments in configuration files that appear later in the list overwrite the arguments in previous configuration files.
- **kwargs** – Keyword arguments passed to the super class ArgumentParser.

checkpoint_dir: str = None
Path to directory containing model checkpoints (`.pkl` file)

checkpoint_path: str = None
Path to model checkpoint (`.pkl` file)

checkpoint_paths: List[str] = None
List of paths to model checkpoints (`.pkl` files)

number_of_molecules: int = 1
Number of molecules in each input to the model. This must equal the length of `smiles_columns` (if not `None`).

preds_path: str
Path to CSV file where predictions will be saved.

process_args() → None
Perform additional argument processing and/or validation.

smiles_columns: List[str] = None
List of names of the columns containing SMILES strings. By default, uses the first `number_of_molecules` columns.

test_path: str
Path to CSV file containing testing data for which predictions will be made.

11.8 Utility Functions

chemprop.args.**get_checkpoint_paths**(*checkpoint_path: Optional[str] = None*, *checkpoint_paths: Optional[List[str]] = None*, *checkpoint_dir: Optional[str] = None*, *ext: str = '.pt'*) → Optional[List[str]]
Gets a list of checkpoint paths either from a single checkpoint path or from a directory of checkpoints.

If `checkpoint_path` is provided, only collects that one checkpoint. If `checkpoint_paths` is provided, collects all of the provided checkpoints. If `checkpoint_dir` is provided, walks the directory and collects all checkpoints. A checkpoint is any file ending in the extension ext.

Parameters

- **checkpoint_path** – Path to a checkpoint.
- **checkpoint_paths** – List of paths to checkpoints.
- **checkpoint_dir** – Path to a directory containing checkpoints.
- **ext** – The extension which defines a checkpoint file.

Returns A list of paths to checkpoints or None if no checkpoint path(s)/dir are provided.

CHAPTER

TWELVE

NEURAL NETWORK UTILITY FUNCTIONS

chemprop.nn_utils.py contains utility funtions specific to neural networks.

class chemprop.nn_utils.**NoamLR**(*optimizer: torch.optim.optimizer.Optimizer*, *warmup_epochs: List[Union[float, int]]*, *total_epochs: List[int]*, *steps_per_epoch: int*, *init_lr: List[float]*, *max_lr: List[float]*, *final_lr: List[float]*)

Noam learning rate scheduler with piecewise linear increase and exponential decay.

The learning rate increases linearly from init_lr to max_lr over the course of the first warmup_steps (where `warmup_steps = warmup_epochs * steps_per_epoch`). Then the learning rate decreases exponentially from `max_lr` to `final_lr` over the course of the remaining `total_steps - warmup_steps` (where `total_steps = total_epochs * steps_per_epoch`). This is roughly based on the learning rate schedule from Attention is All You Need, section 5.3.

Parameters

- **optimizer** – A PyTorch optimizer.
- **warmup_epochs** – The number of epochs during which to linearly increase the learning rate.
- **total_epochs** – The total number of epochs.
- **steps_per_epoch** – The number of steps (batches) per epoch.
- **init_lr** – The initial learning rate.
- **max_lr** – The maximum learning rate (achieved after `warmup_epochs`).
- **final_lr** – The final learning rate (achieved after `total_epochs`).

get_lr() → List[float]

Gets a list of the current learning rates.

Returns A list of the current learning rates.

step(*current_step: Optional[int] = None*)

Updates the learning rate by taking a step.

Parameters **current_step** – Optionally specify what step to set the learning rate to. If None, `current_step = self.current_step + 1`.

chemprop.nn_utils.**compute_gnorm**(*model: torch.nn.modules.module.Module*) → float

Computes the norm of the gradients of a model.

Parameters **model** – A PyTorch model.

Returns The norm of the gradients of the model.

chemprop.nn_utils.**compute_pnorm**(*model: torch.nn.modules.module.Module*) → float

Computes the norm of the parameters of a model.

Parameters **model** – A PyTorch model.

Returns The norm of the parameters of the model.

chemprop.nn_utils.**get_activation_function**(*activation: str*) → torch.nn.modules.module.Module
Gets an activation function module given the name of the activation.

Supports:

- ReLU
- LeakyReLU
- PReLU
- tanh
- SELU
- ELU

Parameters **activation** – The name of the activation function.

Returns The activation function module.

chemprop.nn_utils.**index_select_ND**(*source: torch.Tensor*, *index: torch.Tensor*) → torch.Tensor
Selects the message features from source corresponding to the atom or bond indices in index.

Parameters

- **source** – A tensor of shape (num_bonds, hidden_size) containing message features.
- **index** – A tensor of shape (num_atoms/num_bonds, max_num_bonds) containing the atom or bond indices to select from source.

Returns A tensor of shape (num_atoms/num_bonds, max_num_bonds, hidden_size) containing the message features corresponding to the atoms/bonds specified in index.

chemprop.nn_utils.**initialize_weights**(*model: torch.nn.modules.module.Module*) → None
Initializes the weights of a model in place.

Parameters **model** – An PyTorch model.

chemprop.nn_utils.**param_count**(*model: torch.nn.modules.module.Module*) → int
Determines number of trainable parameters.

Parameters **model** – An PyTorch model.

Returns The number of trainable parameters in the model.

chemprop.nn_utils.**param_count_all**(*model: torch.nn.modules.module.Module*) → int
Determines number of trainable parameters.

Parameters **model** – An PyTorch model.

Returns The number of trainable parameters in the model.

CHAPTER

THIRTEEN

UTILITY FUNCTIONS

chemprop.utils.py contains general purpose utility functions.

chemprop.utils.**accuracy**(*targets: List[int]*, *preds: Union[List[float], List[List[float]]]*, *threshold: float = 0.5*) → float

Computes the accuracy of a binary prediction task using a given threshold for generating hard predictions.

Alternatively, computes accuracy for a multiclass prediction task by picking the largest probability.

Parameters

- **targets** – A list of binary targets.
- **preds** – A list of prediction probabilities.
- **threshold** – The threshold above which a prediction is a 1 and below which (inclusive) a prediction is a 0.

Returns The computed accuracy.

chemprop.utils.**bce**(*targets: List[int]*, *preds: List[float]*) → float

Computes the binary cross entropy loss.

Parameters

- **targets** – A list of binary targets.
- **preds** – A list of prediction probabilities.

Returns The computed binary cross entropy.

chemprop.utils.**build_lr_scheduler**(*optimizer: torch.optim.optimizer.Optimizer*, *args:* chemprop.args.TrainArgs, *total_epochs: Optional[List[int]] = None*) → torch.optim.lr_scheduler._LRScheduler

Builds a PyTorch learning rate scheduler.

Parameters

- **optimizer** – The Optimizer whose learning rate will be scheduled.
- **args** – A *TrainArgs* object containing learning rate arguments.
- **total_epochs** – The total number of epochs for which the model will be run.

Returns An initialized learning rate scheduler.

chemprop.utils.**build_optimizer**(*model: torch.nn.modules.module.Module*, *args:* chemprop.args.TrainArgs) → torch.optim.optimizer.Optimizer

Builds a PyTorch Optimizer.

Parameters

- **model** – The model to optimize.
- **args** – A *TrainArgs* object containing optimizer arguments.

Returns An initialized Optimizer.

chemprop.utils.**create_logger**(*name: str*, *save_dir: Optional[str] = None*, *quiet: bool = False*) → logging.Logger

Creates a logger with a stream handler and two file handlers.

If a logger with that name already exists, simply returns that logger. Otherwise, creates a new logger with a stream handler and two file handlers.

The stream handler prints to the screen depending on the value of `quiet`. One file handler (`verbose.log`) saves all logs, the other (`quiet.log`) only saves important info.

Parameters

- **name** – The name of the logger.
- **save_dir** – The directory in which to save the logs.
- **quiet** – Whether the stream handler should be quiet (i.e., print only important info).

Returns The logger.

chemprop.utils.**get_loss_func**(*args:* chemprop.args.TrainArgs) → torch.nn.modules.module.Module

Gets the loss function corresponding to a given dataset type.

Parameters **args** – Arguments containing the dataset type ("classification", "regression", or "multiclass").

Returns A PyTorch loss function.

chemprop.utils.**get_metric_func**(*metric: str*) → Callable[[Union[List[int], List[float]], List[float]], float]

Gets the metric function corresponding to a given metric name.

Supports:

- `auc`: Area under the receiver operating characteristic curve
- `prc-auc`: Area under the precision recall curve
- `rmse`: Root mean squared error
- `mse`: Mean squared error
- `mae`: Mean absolute error
- `r2`: Coefficient of determination R^2
- `accuracy`: Accuracy (using a threshold to binarize predictions)
- `cross_entropy`: Cross entropy
- `binary_cross_entropy`: Binary cross entropy

Parameters **metric** – Metric name.

Returns A metric function which takes as arguments a list of targets and a list of predictions and returns.

chemprop.utils.**load_args**(*path: str*) → *chemprop.args.TrainArgs*

Loads the arguments a model was trained with.

Parameters **path** – Path where model checkpoint is saved.

Returns The *TrainArgs* object that the model was trained with.

chemprop.utils.**load_checkpoint**(*path: str*, *device: Optional[torch.device] = None*, *logger: Optional[logging.Logger] = None*) → *chemprop.models.model.MoleculeModel*

Loads a model checkpoint.

Parameters

- **path** – Path where checkpoint is saved.
- **device** – Device where the model will be moved.
- **logger** – A logger for recording output.

Returns The loaded *MoleculeModel*.

chemprop.utils.**load_frzn_model**(*model: <module 'torch.nn' from '/home/docs/checkouts/readthedocs.org/user_builds/chemprop/conda/latest/lib/python3.8/site-packages/torch/nn/__init__.py'>*, *path: str*, *current_args: Optional[argparse.Namespace] = None*, *cuda: Optional[bool] = None*, *logger: Optional[logging.Logger] = None*) → *chemprop.models.model.MoleculeModel*

Loads a model checkpoint. :param path: Path where checkpoint is saved. :param current_args: The current arguments. Replaces the arguments loaded from the checkpoint if provided. :param cuda: Whether to move model to cuda. :param logger: A logger. :return: The loaded MoleculeModel.

chemprop.utils.**load_scalers**(*path: str*) → Tuple[*chemprop.data.scaler.StandardScaler*, *chemprop.data.scaler.StandardScaler*, *chemprop.data.scaler.StandardScaler*, *chemprop.data.scaler.StandardScaler*]

Loads the scalers a model was trained with.

Parameters **path** – Path where model checkpoint is saved.

Returns A tuple with the data *StandardScaler* and features *StandardScaler*.

chemprop.utils.**load_task_names**(*path: str*) → List[str]

Loads the task names a model was trained with.

Parameters **path** – Path where model checkpoint is saved.

Returns A list of the task names that the model was trained with.

chemprop.utils.**makedirs**(*path: str*, *isfile: bool = False*) → None

Creates a directory given a path to either a directory or file.

If a directory is provided, creates that directory. If a file is provided (i.e. `isfile == True`), creates the parent directory for that file.

Parameters

- **path** – Path to a directory or file.
- **isfile** – Whether the provided path is a directory or file.

chemprop.utils.**mse**(*targets: List[float]*, *preds: List[float]*) → float

Computes the mean squared error.

Parameters

- **targets** – A list of targets.
- **preds** – A list of predictions.

Returns The computed mse.

chemprop.utils.**overwrite_state_dict**(*loaded_param_name: str*, *model_param_name: str*, *loaded_state_dict: collections.OrderedDict*, *model_state_dict: collections.OrderedDict*, *logger: Optional[logging.Logger] = None*) → collections.OrderedDict

Overwrites a given parameter in the current model with the loaded model. :param loaded_param_name: name of parameter in checkpoint model. :param model_param_name: name of parameter in current model. :param loaded_state_dict: state_dict for checkpoint model. :param model_state_dict: state_dict for current model. :param logger: A logger. :return: The updated state_dict for the current model.

chemprop.utils.**prc_auc**(*targets: List[int]*, *preds: List[float]*) → float

Computes the area under the precision-recall curve.

Parameters

- **targets** – A list of binary targets.
- **preds** – A list of prediction probabilities.

Returns The computed prc-auc.

chemprop.utils.**rmse**(*targets: List[float]*, *preds: List[float]*) → float

Computes the root mean squared error.

Parameters

- **targets** – A list of targets.
- **preds** – A list of predictions.

Returns The computed rmse.

chemprop.utils.**save_checkpoint**(*path: str*, *model:* chemprop.models.model.MoleculeModel, *scaler: Optional[*chemprop.data.scaler.StandardScaler*] = None*, *features_scaler: Optional[*chemprop.data.scaler.StandardScaler*] = None*, *atom_descriptor_scaler: Optional[*chemprop.data.scaler.StandardScaler*] = None*, *bond_feature_scaler: Optional[*chemprop.data.scaler.StandardScaler*] = None*, *args: Optional[*chemprop.args.TrainArgs*] = None*) → None

Saves a model checkpoint.

Parameters

- **model** – A *MoleculeModel*.
- **scaler** – A *StandardScaler* fitted on the data.
- **features_scaler** – A *StandardScaler* fitted on the features.
- **atom_descriptor_scaler** – A *StandardScaler* fitted on the atom descriptors.
- **bond_feature_scaler** – A *StandardScaler* fitted on the bond_fetaures.
- **args** – The *TrainArgs* object containing the arguments the model was trained with.
- **path** – Path where checkpoint will be saved.

chemprop.utils.**save_smiles_splits**(*data_path: str*, *save_dir: str*, *task_names: Optional[List[str]] = None*, *features_path: Optional[List[str]] = None*, *train_data: Optional[*chemprop.data.data.MoleculeDataset*] = None*, *val_data: Optional[*chemprop.data.data.MoleculeDataset*] = None*, *test_data: Optional[*chemprop.data.data.MoleculeDataset*] = None*, *logger: Optional[logging.Logger] = None*, *smiles_columns: Optional[List[str]] = None*) → None

Saves a csv file with train/val/test splits of target data and additional features. Also saves indices of train/val/test split as a pickle file. Pickle file does not support repeated entries with the same SMILES or entries entered from

a path other than the main data path, such as a separate test path.

Parameters

- **data_path** – Path to data CSV file.
- **save_dir** – Path where pickle files will be saved.
- **task_names** – List of target names for the model as from the function get_task_names(). If not provided, will use datafile header entries.
- **features_path** – List of path(s) to files with additional molecule features.
- **train_data** – Train *MoleculeDataset*.
- **val_data** – Validation *MoleculeDataset*.
- **test_data** – Test *MoleculeDataset*.
- **smiles_columns** – The name of the column containing SMILES. By default, uses the first column.
- **logger** – A logger for recording output.

chemprop.utils.**timeit**(*logger_name: Optional[str] = None*) → Callable[[Callable], Callable]

Creates a decorator which wraps a function with a timer that prints the elapsed time.

Parameters **logger_name** – The name of the logger used to record output. If None, uses `print` instead.

Returns A decorator which wraps a function with a timer that prints the elapsed time.

chemprop.utils.**update_prediction_args**(*predict_args: chemprop.args.PredictArgs*, *train_args: chemprop.args.TrainArgs*, *missing_to_defaults: bool = True*, *validate_feature_sources: bool = True*) → None

Updates prediction arguments with training arguments loaded from a checkpoint file. If an argument is present in both, the prediction argument will be used.

Also raises errors for situations where the prediction arguments and training arguments are different but must match for proper function.

Parameters

- **predict_args** – The *PredictArgs* object containing the arguments to use for making predictions.
- **train_args** – The *TrainArgs* object containing the arguments used to train the model previously.
- **missing_to_defaults** – Whether to replace missing training arguments with the current defaults for :class: *~chemprop.args.TrainArgs*. This is used for backwards compatibility.
- **validate_feature_sources** – Indicates whether the feature sources (from path or generator) are checked for consistency between the training and prediction arguments. This is not necessary for fingerprint generation, where molecule features are not used.

CHAPTER

FOURTEEN

SCIKIT-LEARN MODELS

In addition to message passing neural networks, Chemprop also enables training and predicting with scikit-learn Random Forest and Support Vector Machine models applied to Morgan fingerprints.

14.1 Scikit-Learn Train

chemprop.sklearn_train.py contains functions for training `scikit-learn` models.

chemprop.sklearn_train.**impute_sklearn**(*model: Union[sklearn.ensemble._forest.RandomForestRegressor, sklearn.ensemble._forest.RandomForestClassifier, sklearn.svm._classes.SVR, sklearn.svm._classes.SVC]*, *train_data:* chemprop.data.data.MoleculeDataset, *args:* chemprop.args.SklearnTrainArgs, *logger: Optional[logging.Logger] = None*, *threshold: float = 0.5*) → List[float]

Trains a single-task scikit-learn model, meaning a separate model is trained for each task.

This is necessary if some tasks have None (unknown) values.

Parameters

- **model** – The scikit-learn model to train.
- **train_data** – The training data.
- **args** – A *SklearnTrainArgs* object containing arguments for training the scikit-learn model.
- **logger** – A logger to record output.
- **theshold** – Threshold for classification tasks.

Returns A list of list of target values.

chemprop.sklearn_train.**multi_task_sklearn**(*model: Union[sklearn.ensemble._forest.RandomForestRegressor, sklearn.ensemble._forest.RandomForestClassifier, sklearn.svm._classes.SVR, sklearn.svm._classes.SVC]*, *train_data:* chemprop.data.data.MoleculeDataset, *test_data:* chemprop.data.data.MoleculeDataset, *metrics: List[str]*, *args:* chemprop.args.SklearnTrainArgs, *logger: Optional[logging.Logger] = None*) → Dict[str, List[float]]

Trains a multi-task scikit-learn model, meaning one model is trained simultaneously on all tasks.

This is only possible if none of the tasks have None (unknown) values.

Parameters

- **model** – The scikit-learn model to train.
- **train_data** – The training data.
- **test_data** – The test data.
- **metrics** – A list of names of metric functions.
- **args** – A *SklearnTrainArgs* object containing arguments for training the scikit-learn model.
- **logger** – A logger to record output.

Returns A dictionary mapping each metric in `metrics` to a list of values for each task.

chemprop.sklearn_train.**predict**(*model: Union[sklearn.ensemble._forest.RandomForestRegressor, sklearn.ensemble._forest.RandomForestClassifier, sklearn.svm._classes.SVR, sklearn.svm._classes.SVC]*, *model_type: str*, *dataset_type: str*, *features: List[numpy.ndarray]*) → List[List[float]]

Predicts using a scikit-learn model.

Parameters

- **model** – The trained scikit-learn model to make predictions with.
- **model_type** – The type of model.
- **dataset_type** – The type of dataset.
- **features** – The data features used as input for the model.

Returns A list of lists of floats containing the predicted values.

chemprop.sklearn_train.**run_sklearn**(*args:* chemprop.args.SklearnTrainArgs, *data:* chemprop.data.data.MoleculeDataset, *logger: Optional[logging.Logger] = None*) → Dict[str, List[float]]

Loads data, trains a scikit-learn model, and returns test scores for the model checkpoint with the highest validation score.

Parameters

- **args** – A *SklearnTrainArgs* object containing arguments for loading data and training the scikit-learn model.
- **data** – A `MoleculeDataset` containing the data.
- **logger** – A logger to record output.

Returns A dictionary mapping each metric in `metrics` to a list of values for each task.

chemprop.sklearn_train.**single_task_sklearn**(*model: Union[sklearn.ensemble._forest.RandomForestRegressor, sklearn.ensemble._forest.RandomForestClassifier, sklearn.svm._classes.SVR, sklearn.svm._classes.SVC]*, *train_data:* chemprop.data.data.MoleculeDataset, *test_data:* chemprop.data.data.MoleculeDataset, *metrics: List[str]*, *args:* chemprop.args.SklearnTrainArgs, *logger: Optional[logging.Logger] = None*) → List[float]

Trains a single-task scikit-learn model, meaning a separate model is trained for each task.

This is necessary if some tasks have None (unknown) values.

Parameters

- **model** – The scikit-learn model to train.
- **train_data** – The training data.
- **test_data** – The test data.
- **metrics** – A list of names of metric functions.
- **args** – A *SklearnTrainArgs* object containing arguments for training the scikit-learn model.
- **logger** – A logger to record output.

Returns A dictionary mapping each metric in `metrics` to a list of values for each task.

chemprop.sklearn_train.**sklearn_train**() → None
Parses scikit-learn training arguments and trains a scikit-learn model.

This is the entry point for the command line command `sklearn_train`.

14.2 Scikit-Learn Predict

chemprop.sklearn_predict.py contains functions for training `scikit-learn` models.

chemprop.sklearn_predict.**predict_sklearn**(*args:* chemprop.args.SklearnPredictArgs) → None

Loads data and a trained scikit-learn model and uses the model to make predictions on the data.

Parameters **args** – A *SklearnPredictArgs* object containing arguments for loading data, loading a trained scikit-learn model, and making predictions with the model.

chemprop.sklearn_predict.**sklearn_predict**() → None
Parses scikit-learn predicting arguments and runs prediction using a trained scikit-learn model.

This is the entry point for the command line command `sklearn_predict`.

CHAPTER

FIFTEEN

USEFUL SCRIPTS

Additional useful scripts for working with property prediction datasets are contained in https://github.com/chemprop/chemprop/tree/master/scripts.

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W