CONTENTS:

1 Requirements 3

2 Installation 5
  2.1 Overview ................................................. 5
  2.2 Conda ................................................. 5
  2.3 Option 1: Installing from PyPi .......................... 5
  2.4 Option 2: Installing from source .................. 5
  2.5 Docker .................................................. 6

3 Tutorial 7
  3.1 Data .................................................. 7
  3.2 Training .............................................. 7
  3.3 Predicting .............................................. 10
  3.4 Web Interface ......................................... 11

4 Web Interface 13
  4.1 Overview ............................................. 13
  4.2 Flask ................................................. 14
  4.3 Gunicorn ............................................... 14

5 Data 15
  5.1 Data .................................................. 15
  5.2 Scaffold ............................................... 19
  5.3 Scaler ................................................ 20
  5.4 Utils .................................................. 21

6 Features 25
  6.1 Featurization .......................................... 25
  6.2 Features Generators ................................... 28
  6.3 Utils .................................................. 30

7 Models 31
  7.1 Model .................................................. 31
  7.2 MPN .................................................... 32

8 Training and Predicting 35
  8.1 Train .................................................. 35
  8.2 Run Training .......................................... 36
  8.3 Cross-Validation ....................................... 36
  8.4 Predict ............................................... 37
  8.5 Make Predictions ..................................... 37
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*.
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.
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.
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 a CSV file with a header row. For example:

```
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 adding computed molecule-level features can further improve performance on certain datasets. Features can be added to the model using the `--features_generator <generator>` flag for molecule-level features, or `--atom_descriptors <mode>` for atom-level features, or both.

#### 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](https://arxiv.org/pdf/1904.01561.pdf)
- **rdkit_2d_normalized** is the CDF-normalized version of the 200 rdkit descriptors.

#### 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.

#### Atomic 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, where the command line option `--atom_descriptors descriptor` concatenates the new features to
the embedded atomic features after the D-MPNN, or 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.

### 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>` Path where a CSV file containing the predictions will be saved.

For example:

```bash
chemprop_predict --test_path data/tox21.csv --checkpoint_dir tox21_checkpoints --preds_path tox21_preds.csv
```

or

```bash
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:

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

### 5.1 Data

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

```python
import torch

class MoleculeDataLoader:
    def __init__(self, dataset: 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

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

        property targets

        Returns the targets associated with each molecule.
```

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.
- **features** – A numpy array containing additional features (e.g., Morgan fingerprint).
- **features_generator** – A list of features generators to use.

`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`
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`
Gets the number of molecules in the `MoleculeDatapoint`.

**Returns**

- **The number of molecules.**

`reset_features_and_targets() → None`
Resets the features and targets to their raw values.

`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` with access to their attributes.

**Parameters**

- **data** – A list of `MoleculeDatapoints`.

`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_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.featureization.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 \texttt{batch_graph()}. This means that if the underlying set of \texttt{MoleculeDatapoints} 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.

\texttt{features()} \rightarrow \texttt{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.

\texttt{features_size()} \rightarrow \texttt{int}

Returns the size of the additional features vector associated with the molecules.

Returns The size of the additional features vector.

\texttt{mols()} \rightarrow \texttt{Union[List[rdkit.Chem.rdchem.Mol], List[List[rdkit.Chem.rdchem.Mol]]]}

Returns a list of the RDKit molecules associated with each MoleculeDatapoint.

Parameters \texttt{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 \texttt{flatten}.

\texttt{normalize_features()} \rightarrow \texttt{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

• \texttt{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.

• \texttt{replace_nan_token} – A token to use to replace NaN entries in the features.

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.

\texttt{normalize_targets()} \rightarrow \texttt{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.

\texttt{num_tasks()} \rightarrow \texttt{int}

Returns the number of prediction tasks.

Returns The number of tasks.
property number_of_molecules
    Gets the number of molecules in each MoleculeDatapoint.

    Returns The number of molecules.

reset_features_and_targets() → None
    Resets the features 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.

cached_graph() → bool
    Returns whether MolGraphs will be cached.

cached_mol() → bool
    Returns whether RDKit molecules will be cached.

construct_molecule_batch (data: List[chemprop.data.data.MoleculeDatapoint]) → chemprop.data.data.MoleculeDataset
    Constructs a MoleculeDataset from a list of MoleculeDatapoints.

    Additionally, precomputes the BatchMolGraph for the constructed MoleculeDataset.

    Parameters data – A list of MoleculeDatapoints.

    Returns A MoleculeDataset containing all the MoleculeDatapoints.

set_cache_graph (cache_graph: bool) → None
    Sets whether MolGraphs will be cached.

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], 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.


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 MoleculeDatasets 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]], 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.


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 Utilities

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

`chemprop.data.utils.filter_invalid_smiles(data: chemprop.data.data.MoleculeDataset) \rightarrow 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) \rightarrow 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.


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`.
- `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`.
- `atom_descriptors_path` - The path to the file containing the custom atom descriptors.
- `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.

```python
```

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.

```python
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.

```python
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** – 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`.

```python
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.

```python
chemprop.data.utils.preprocess_smiles_columns(smiles_columns: Optional[Union[str, List[Optional[str]]]] = None) ➞ List[Optional[str]]
```

Preprocesses the **smiles_columns** variable to ensure that it is a list.

**Parameters**

- **smiles_columns** – The names of the columns containing SMILES. By default, uses the first **number_of_molecules** columns.

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

```python
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.

```python
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.

```python
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.
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.

```python
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 MolGraphs from which to construct the BatchMolGraph.

**Methods**

- `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 bond index to all the incoming bond 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.

Returns the components of the BatchMolGraph.

The returned components are, in order:

- \(f\_\text{atoms}\)
- \(f\_\text{bonds}\)
- \(a2b\)
- \(b2a\)
- \(b2revb\)
- \(a\_\text{scope}\)
- \(b\_\text{scope}\)

**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).


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

A MolGraph computes the following attributes:

- \(n\_\text{atoms}\): The number of atoms in the molecule.
- \(n\_\text{bonds}\): The number of bonds in the molecule.
- \(f\_\text{atoms}\): A mapping from an atom index to a list of atom features.
- \(f\_\text{bonds}\): 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.

**Parameters** mol – A SMILES or an RDKit molecule.


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.

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() → int

Gets the dimensionality of the atom feature vector.

chemprop.features.featurization.get_bond_fdim(atom_messages: 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.

Returns The dimensionality of the bond feature vector.


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_descriptors_batch – A list of 2D numpy array containing additional atom descriptors to featurize the molecule.

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.set_extra_atom_fdim(extra)

Change the dimensionality of the atom feature vector.
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.

```python
def get_available_features_generators() -> List[str]
    Returns a list of names of available features generators.

def 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.
```

```python
def 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.
```

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.

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.

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.

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_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
- `.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.
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.

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

**Parameters**

- `args` – A `TrainArgs` object containing model arguments.
- `featurizer` – Whether the model should act as a featurizer, i.e., outputting the learned features from the last layer prior to prediction rather than outputting the actual property predictions.

```python
create_encoder(args: chemprop.args.TrainArgs) -> None
```

Creates the message passing encoder for the model.

**Parameters**

- `args` – A `TrainArgs` object containing model arguments.

```python
create_ffn(args: chemprop.args.TrainArgs) -> None
```

Creates the feed-forward layers for the model.

**Parameters**

- `args` – A `TrainArgs` object containing model arguments.

```python
```

Computes feature vectors of the input by running the model except for the last layer.

**Parameters**

- `batch` – A list of SMILES, a list of RDKit molecules, or a `BatchMolGraph`.
- `features_batch` – A list of numpy arrays containing additional features.
- `atom_descriptors_batch` – A list of numpy arrays containing additional atom descriptors.

**Returns** The feature vectors computed by the `MoleculeModel`. 

Runs the MoleculeModel on input.

Parameters

• **batch** – A list of SMILES, a list of RDKit molecules, or a BatchMolGraph.

• **features_batch** – A list of numpy arrays containing additional features.

• **atom_descriptors_batch** – A list of numpy arrays containing additional atom descriptors.

Returns The output of the MoleculeModel, which is either property predictions or molecule features if self.featurizer=True.

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.


Encodes a batch of molecules.

Parameters

• **batch** – A list of list of SMILES, a list of list of RDKit molecules, or a BatchMolGraph.

• **features_batch** – A list of numpy arrays containing additional features.

• **atom_descriptors_batch** – A list of numpy arrays containing additional atom descriptors.

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.

```python
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.
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.


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.


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.


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.

```python
```

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 around chemprop.train.predict.py which loads data, loads a trained model, makes predictions, and saves those predictions.

```python
def make_predictions(args: chemprop.args.PredictArgs, smiles: List[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.

**Returns** A list of lists of target predictions.
8.6 Evaluate

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

```python
```

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.

```python
```

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.
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, \ldots, 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, \ldots, 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.
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.


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`.


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

The size of the atom descriptors.

property atom_features_size

The size of the atom features.

batch_size: int = 50

Batch size.

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,

```python
    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
Whether to use CUDA (i.e., GPUs) or not.

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

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
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).

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

```python
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

**atom_messages**: `bool = False`

Centers messages on atoms instead of on bonds.

**bias**: `bool = False`

Whether to add bias to linear layers.

**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.

**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**

Index sets used for splitting data into train/validation/test during cross-validation

**data_path**: `str`

Path to data CSV file.
dataset_type: Literal[regression, classification, multiclass]
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.

extra_metrics: List[Literal[auc, prc-auc, rmse, mae, mse, r2, accuracy, cross_entropy]]
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
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.

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]
Metric to use during evaluation. It is also used with the validation set for early stopping. Defaults to “auc” for classification and “rmse” for regression.

property metrics
The list of metrics used for evaluation. Only the first is used for early stopping.

property minimize_score
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.

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

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

property num_tasks
   The number of tasks being trained on.

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.

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_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_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.

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

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] = 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.
property task_names
   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
   The size of the training data set.

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

property use_input_features
   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 \texttt{init\_lr} to \texttt{max\_lr}. Afterwards, learning rate decreases exponentially from \texttt{max\_lr} to \texttt{final\_lr}.

11.3 Predict Arguments

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

\texttt{PredictArgs} includes \texttt{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.

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

\texttt{property ensemble_size}
   The number of models in the ensemble.

\texttt{preds\_path: str}
   Path to CSV file where predictions will be saved.
process_args() \rightarrow 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() \rightarrow 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.

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

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

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).
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


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

```python
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.

**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.
```

Computes the molecule vectors output from the last layer of a MoleculeModel.

**Parameters**

- **model** – A MoleculeModel.
- **data** – A MoleculeDataset.
- **batch_size** – Batch size.
- **num_workers** – Number of parallel data loading workers.

**Returns** A list of 1D numpy arrays of length hidden_size containing the molecule vectors generated by the model for each molecule provided.

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.


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.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.

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.

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.

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`.


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.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.models.chemprop.models.model.MoleculeModel,
scaler: Optional[chemprop.data.scaler.StandardScaler] = None,

Saves a model checkpoint.

Parameters

model – A MoleculeModel.

scaler – A StandardScaler fitted on the data.

features_scaler – A StandardScaler fitted on the features.

args – The TrainArgs object containing the arguments the model was trained with.

path – Path where checkpoint will be saved.


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 same SMILES.

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.

chemprop.models.chemprop.models.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.
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.


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.

```python
```

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.

```python
```

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.

```python
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`. 

62 Chapter 14. Scikit-Learn Models
14.2 Scikit-Learn Predict

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

chemprop.sklearn_predict.predict_sklearn \((args: \text{chemprop.args.SklearnPredictArgs}) \rightarrow \text{None}\)

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

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

chemprop.sklearn_predict.sklearn_predict() \rightarrow \text{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 \texttt{sklearn_predict}.
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
C
chemprop.args, 52
chemprop.data.data, 15
chemprop.data.scaffold, 19
chemprop.data.scaler, 20
chemprop.data.utils, 21
chemprop.features.features_generators, 28
chemprop.features.featureization, 25
chemprop.features.utils, 30
chemprop.hyperparameter_optimization, 39
chemprop.interpret, 41
chemprop.models.model, 31
chemprop.models.mpn, 32
chemprop.nn_utils, 53
chemprop.sklearn_predict, 63
chemprop.sklearn_train, 61
chemprop.train.cross_validate, 36
chemprop.train.evaluate, 38
chemprop.train.make_predictions, 37
chemprop.train.predict, 37
chemprop.train.run_training, 36
chemprop.train.train, 35
chemprop.utils, 57
INDEX

A

accuracy() (in module chemprop.utils), 57
activation (chemprop.args.TrainArgs attribute), 45
aggregation (chemprop.args.TrainArgs attribute), 45
aggregation_norm (chemprop.args.TrainArgs attribute), 45
atom_descriptors (chemprop.args.CommonArgs attribute), 43
atom_descriptors() (chemprop.data.data.MoleculeDataset method), 16
atom_descriptors_path (chemprop.args.CommonArgs attribute), 43
atom_descriptors_size() (chemprop.args.CommonArgs property), 43
atom_descriptors_size() (chemprop.data.data.MoleculeDataset method), 16
atom_features() (in module chemprop.features.featureization), 26
atom_features() (chemprop.args.CommonArgs property), 43
atom_features_size() (chemprop.args.CommonArgs property), 43
atom_features_size() (chemprop.data.data.MoleculeDataset method), 16
atom_messages (chemprop.args.TrainArgs attribute), 45

B

batch_graph() (chemprop.data.data.MoleculeDataset method), 16
batch_size (chemprop.args.CommonArgs attribute), 43
batch_size (chemprop.args.InterpretArgs attribute), 49
BatchMolGraph (class in chemprop.features.featureization), 25
bce() (in module chemprop.utils), 57
bias (chemprop.args.TrainArgs attribute), 45
bond_features() (in module chemprop.features.featureization), 26
build_lr_scheduler() (in module chemprop.utils), 57
build_optimizer() (in module chemprop.utils), 57

c

c_puct (chemprop.args.InterpretArgs attribute), 49
cache_cutoff (chemprop.args.TrainArgs attribute), 45
cache_graph() (in module chemprop.data.data), 18
cache_mol() (in module chemprop.data.data), 18
checkpoint_dir (chemprop.args.CommonArgs attribute), 43
checkpoint_dir (chemprop.args.SklearnPredictArgs attribute), 51
checkpoint_path (chemprop.args.CommonArgs attribute), 43
checkpoint_path (chemprop.args.SklearnPredictArgs attribute), 51
checkpoint_paths (chemprop.args.CommonArgs attribute), 44
checkpoint_paths (chemprop.args.SklearnPredictArgs attribute), 51
chemprop.args module, 52
chemprop.data.data module, 15
chemprop.data.scaffold module, 19
chemprop.data.scaler module, 20
chemprop.data.utils module, 21
chemprop.features.featureization module, 28
chemprop.features.featureization module, 25
chemprop.features.utils module, 30
chemprop.hyperparameter_optimization module, 39
mcts_rollout() (in module chemprop.interpret), 42
MCTSNnode (class in chemprop.interpret), 41
metric (chemprop.args.TrainArgs attribute), 46
metrics (chemprop.args.TrainArgs property), 46
min_atoms (chemprop.args.InterpretArgs attribute), 49
minimize_score() (chemprop.args.TrainArgs property), 46
model_type (chemprop.args.SklearnTrainArgs attribute), 51
module
chemprop.args, 52
chemprop.data.data, 15
chemprop.data.scaffold, 19
chemprop.data.utils, 21
chemprop.features.features_generators, 28
chemprop.features.featureization, 25
chemprop.features.utils, 30
chemprop.train.train, 35
mols() (chemprop.data.data.MoleculeDataset property), 17
mol2graph() (in module chemprop.features.featureization), 27
MoleculeDataLoader (class in chemprop.data.data), 15
MoleculeDatapoint (class in chemprop.data.data), 15
MoleculeDataset (class in chemprop.data.data), 16
MoleculeModel (class in chemprop.models.model), 31
MoleculeSampler (class in chemprop.data.data), 18
MolGraph (class in chemprop.features.featureization), 26
mol() (chemprop.data.data.MoleculeDatapoint property), 16
normalize_features() (chemprop.data.data.MoleculeDataset method), 17
normalize_targets() (chemprop.data.data.MoleculeDataset method), 17
num_bits (chemprop.args.SklearnTrainArgs attribute), 51
num_folds (chemprop.args.SklearnTrainArgs attribute), 51
no_cache_mol (chemprop.args.CommonArgs attribute), 44
no_cuda (chemprop.args.CommonArgs attribute), 44
no_features_scaling (chemprop.args.CommonArgs attribute), 44
NoamLR (class in chemprop.nn_utils), 53
num_lrs() (chemprop.args.TrainArgs property), 50
num_tasks() (chemprop.args.TrainArgs property), 47
num_trees (chemprop.args.SklearnTrainArgs attribute), 51
num_workers (chemprop.args.CommonArgs attribute), 44
num_tasks() (chemprop.data.data.MoleculeDatapoint method), 16
num_workers (chemprop.args.CommonArgs attribute), 44
O
onek_encoding_unk() (in module chemprop.features.featureizers), 28
MPN (class in chemprop.models.mpn), 32
MPN_shared (chemprop.args.TrainArgs attribute), 46
MSE (class in chemprop.models.mpn), 32
multi_task_sklearn() (in module chemprop.sklearn_train), 61
multiclass_num_classes (chemprop.args.TrainArgs attribute), 47
N
no_cache_mol (chemprop.args.CommonArgs attribute), 44
no_cuda (chemprop.args.CommonArgs attribute), 44
no_features_scaling (chemprop.args.CommonArgs attribute), 44
NoamLR (class in chemprop.nn_utils), 53
normalize_features() (chemprop.data.data.MoleculeDataset method), 17
normalize_targets() (chemprop.data.data.MoleculeDataset method), 17
num_bits (chemprop.args.SklearnTrainArgs attribute), 51
num_folds (chemprop.args.SklearnTrainArgs attribute), 51
num_lrs() (chemprop.args.TrainArgs property), 50
num_tasks() (chemprop.args.TrainArgs property), 47
num_tasks() (chemprop.data.data.MoleculeDatapoint method), 16
num_tasks() (chemprop.data.data.MoleculeDataset method), 17
num_trees (chemprop.args.SklearnTrainArgs attribute), 51
num_workers (chemprop.args.CommonArgs attribute), 44
number_of_molecules (chemprop.args.CommonArgs attribute), 44
number_of_molecules (chemprop.args.SklearnPredictArgs attribute), 51
number_of_molecules() (chemprop.data.data.MoleculeDatapoint property), 16
number_of_molecules() (chemprop.data.data.MoleculeDataset property), 17
<table>
<thead>
<tr>
<th>Function/Class</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>param_count()</code></td>
<td>(in module chemprop.nn_utils)</td>
<td>55</td>
</tr>
<tr>
<td><code>prc_auc()</code></td>
<td>(in module chemprop.utils)</td>
<td>59</td>
</tr>
<tr>
<td><code>predict()</code></td>
<td>(in module chemprop.sklearn_train)</td>
<td>61</td>
</tr>
<tr>
<td><code>predict_sklearn()</code></td>
<td>(in module chemprop.sklearn_predict)</td>
<td>63</td>
</tr>
<tr>
<td><code>PredictArgs</code></td>
<td>(class in chemprop.args)</td>
<td>48</td>
</tr>
<tr>
<td><code>preds_path</code></td>
<td>(chemprop.args.PredictArgs attribute)</td>
<td>48</td>
</tr>
<tr>
<td><code>preprocess_smiles_columns()</code></td>
<td>(in module chemprop.data.utils)</td>
<td>23</td>
</tr>
<tr>
<td><code>process_args()</code></td>
<td>(chemprop.args.CommonArgs method)</td>
<td>44</td>
</tr>
<tr>
<td><code>process_args()</code></td>
<td>(chemprop.args.InterpretArgs method)</td>
<td>49</td>
</tr>
<tr>
<td><code>process_args()</code></td>
<td>(chemprop.args.PredictArgs method)</td>
<td>48</td>
</tr>
<tr>
<td><code>process_args()</code></td>
<td>(chemprop.args.SklearnPredictArgs method)</td>
<td>51</td>
</tr>
<tr>
<td><code>process_args()</code></td>
<td>(chemprop.args.TrainArgs method)</td>
<td>47</td>
</tr>
<tr>
<td><code>prop_delta</code></td>
<td>(chemprop.args.InterpretArgs attribute)</td>
<td>49</td>
</tr>
<tr>
<td><code>property_id</code></td>
<td>(chemprop.args.InterpretArgs attribute)</td>
<td>49</td>
</tr>
<tr>
<td><code>pytorch_seed</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>quiet</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>radius</code></td>
<td>(chemprop.args.SklearnTrainArgs attribute)</td>
<td>51</td>
</tr>
<tr>
<td><code>rdkit_2d_features_generator()</code></td>
<td>(in module chemprop.features.features_generators)</td>
<td>29</td>
</tr>
<tr>
<td><code>rdkit_2d_normalized_features_generator()</code></td>
<td>(in module chemprop.features.features_generators)</td>
<td>29</td>
</tr>
<tr>
<td><code>register_features_generator()</code></td>
<td>(in module chemprop.features.features_generators)</td>
<td>29</td>
</tr>
<tr>
<td><code>reset_features_and_targets()</code></td>
<td>(chemprop.data.data.MoleculeDatapoint method)</td>
<td>16</td>
</tr>
<tr>
<td><code>reset_features_and_targets()</code></td>
<td>(chemprop.data.data.MoleculeDataset method)</td>
<td>18</td>
</tr>
<tr>
<td><code>rmse()</code></td>
<td>(in module chemprop.utils)</td>
<td>59</td>
</tr>
<tr>
<td><code>rollout</code></td>
<td>(chemprop.args.InterpretArgs attribute)</td>
<td>49</td>
</tr>
<tr>
<td><code>run_sklearn()</code></td>
<td>(in module chemprop.sklearn_train)</td>
<td>62</td>
</tr>
<tr>
<td><code>run_training()</code></td>
<td>(in module chemprop.train.run_training)</td>
<td>36</td>
</tr>
<tr>
<td><code>save_checkpoint()</code></td>
<td>(in module chemprop.utils)</td>
<td>60</td>
</tr>
<tr>
<td><code>save_dir</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>save_features()</code></td>
<td>(in module chemprop.features.utils)</td>
<td>30</td>
</tr>
<tr>
<td><code>save_preds</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>save_smiles_splits</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>save_smiles_splits()</code></td>
<td>(in module chemprop.utils)</td>
<td>60</td>
</tr>
<tr>
<td><code>scaffold_split()</code></td>
<td>(in module chemprop.data.scaffold)</td>
<td>19</td>
</tr>
<tr>
<td><code>scaffold_to_smiles()</code></td>
<td>(in module chemprop.data.scaffold)</td>
<td>19</td>
</tr>
<tr>
<td><code>seed</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>separate_test_features_path</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>separate_test_path</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>separate_val_features_path</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>separate_val_path</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>set_cache_graph()</code></td>
<td>(in module chemprop.data.data)</td>
<td>18</td>
</tr>
<tr>
<td><code>set_cache_mol()</code></td>
<td>(in module chemprop.data.data)</td>
<td>18</td>
</tr>
<tr>
<td><code>set_extra_atom_fdim()</code></td>
<td>(in module chemprop.features.featureization)</td>
<td>27</td>
</tr>
<tr>
<td><code>set_features()</code></td>
<td>(chemprop.data.data.MoleculeDatapoint method)</td>
<td>27</td>
</tr>
<tr>
<td><code>set_features()</code></td>
<td>(chemprop.data.data.MoleculeDataset method)</td>
<td>27</td>
</tr>
<tr>
<td><code>set_targets()</code></td>
<td>(chemprop.data.data.MoleculeDatapoint method)</td>
<td>16</td>
</tr>
<tr>
<td><code>set_targets()</code></td>
<td>(chemprop.data.data.MoleculeDataset method)</td>
<td>18</td>
</tr>
<tr>
<td><code>show_individual_scores</code></td>
<td>(chemprop.args.TrainArgs attribute)</td>
<td>47</td>
</tr>
<tr>
<td><code>single_task</code></td>
<td>(chemprop.args.SklearnTrainArgs attribute)</td>
<td>51</td>
</tr>
<tr>
<td><code>single_task_sklearn()</code></td>
<td>(in module chemprop.sklearn_train)</td>
<td>62</td>
</tr>
<tr>
<td><code>sklearn_predict()</code></td>
<td>(in module chemprop.sklearn_predict)</td>
<td>63</td>
</tr>
<tr>
<td><code>sklearn_train()</code></td>
<td>(in module chemprop.sklearn_train)</td>
<td>62</td>
</tr>
<tr>
<td><code>SklearnPredictArgs</code></td>
<td>(class in chemprop.args)</td>
<td>51</td>
</tr>
<tr>
<td><code>SklearnTrainArgs</code></td>
<td>(class in chemprop.args)</td>
<td>50</td>
</tr>
<tr>
<td><code>smiles()</code></td>
<td>(chemprop.data.data.MoleculeDataset method)</td>
<td>18</td>
</tr>
<tr>
<td><code>smiles_columns</code></td>
<td>(chemprop.args.CommonArgs attribute)</td>
<td>44</td>
</tr>
</tbody>
</table>
smiles_columns (chemprop.args.SklearnPredictArgs attribute), 51
split_data() (in module chemprop.data.utils), 23
split_sizes (chemprop.args.TrainArgs attribute), 47
split_type (chemprop.args.TrainArgs attribute), 47
StandardScaler (class in chemprop.data.scaler), 20
step() (chemprop.nn_utils.NoamLR method), 53

T
target_columns (chemprop.args.TrainArgs attribute), 47
targets() (chemprop.data.data.MoleculeDataLoader property), 15
targets() (chemprop.data.data.MoleculeDataset method), 18
task_names() (chemprop.args.TrainArgs property), 47
test (chemprop.args.TrainArgs attribute), 48
test_fold_index (chemprop.args.TrainArgs attribute), 48
test_path (chemprop.args.PredictArgs attribute), 49
test_path (chemprop.args.SklearnPredictArgs attribute), 49
timeit() (in module chemprop.utils), 60
train() (in module chemprop.train.train), 35
train_data_size() (chemprop.args.TrainArgs property), 48
TrainArgs (class in chemprop.args), 45
transform() (chemprop.data.scaler.StandardScaler method), 20

U
undirected (chemprop.args.TrainArgs attribute), 48
use_input_features() (chemprop.args.TrainArgs property), 48

V
val_fold_index (chemprop.args.TrainArgs attribute), 48
validate_data() (in module chemprop.data.utils), 23
validate_dataset_type() (in module chemprop.data.utils), 23

W
warmup_epochs (chemprop.args.TrainArgs attribute), 48