ECNet Documentation

Release 3.3.1

Travis Kessler

Jan 28, 2020

Navigation

1 Installation **1**
- 1.1 Prerequisites . . . 1
- 1.2 Install via pip . . . 1
- 1.3 Install from source . . . 1

2 Quick Start **3**
- 2.1 Preface . . . 3
- 2.2 Model configuration file . . . 3
- 2.3 Using the Server object . . . 4

3 ECNet Tools **9**
- 3.1 Database creation . . . 9
- 3.2 ECNet .prj file usage . . . 10
- 3.3 Constructing parity plots . . . 10

4 Example Scripts **13**
- 4.1 SMILES String Validation . . . 13

CHAPTER 1

Installation

1.1 Prerequisites

- Have Python 3.7 installed
- Have the ability to install Python packages

1.2 Install via pip

If you are working in a Linux/Mac environment:

```
sudo pip install ecnet
```

Alternatively, in a Windows or virtualenv environment:

```
pip install ecnet
```

Note: if multiple Python releases are installed on your system (e.g. 2.7 and 3.7), you may need to execute the correct version of pip. For Python 3.7, change **"pip install ecnet"** to **"pip3 install ecnet"**.

To update your version of ECNet to the latest release version, use:

```
pip install --upgrade ecnet
```

1.3 Install from source

Download the ECNet repository, navigate to the download location on the command line/terminal, and execute:

```
python setup.py install
```

Additional package dependencies (ColorLogging, Ditto, ecabc, Keras, NumPy, PubChemPy, PyYaml, TensorFlow) will be installed during the ECNet installation process. If raw performance is your thing, consider building numerical packages like TensorFlow and NumPy from source.

CHAPTER 2

Quick Start

A full tutorial and additional databases are provided (tutorial, databases) to get you started. The following documentation is intended for users seeking additional information about ECNet's functionality.

2.1 Preface

ECNet operates using a **Server** object that interfaces with data utility classes, error calculation functions, and neural network creation classes. The Server object handles importing data and model creation for your project, and serves the data to models. Configurable variables for neural networks, such as learning rate, number of neurons per hidden layer, activation functions for hidden/input/output layers, and number of training epochs are found in a **.yml** configuration file.

2.2 Model configuration file

For training, we apply the Adam optimization algorithm to feed-forward neural networks. Here is the default model configuration, outlining default values we pass to model creation and training functions:

```
---
beta_1: 0.9
beta_2: 0.999
decay: 0.0
epochs: 2000
epsilon: 0.0000001
hidden_layers:
- - 32
  - relu
- - 32
  - relu
learning_rate: 0.001
output_activation: linear
```

2.3 Using the Server object

First, create a Python script to handle your task and copy an ECNet-formatted CSV database file to your working directory. The Server object will create a default configuration file if an existing one is not specified or found.

Your first steps are importing the Server object, initializing the Server and importing some data:

```
from ecnet import Server

# Initialize a Server
sv = Server()

# If `config.yml` does not already exist in your working directory, it will be␣
↪created with
#   default values; to specify another configuration file, use the model_config␣
↪argument
sv = Server(model_config='my_model_configuration.yml')

# You can utilize parallel processing (multiprocessing) for model training and␣
↪hyperparameter
#   tuning:
sv = Server(num_processes=4)
sv.num_processes = 8

# Import an ECNet-formatted CSV database
sv.load_data('my_data.csv')

# Learning, validation and test sets are defined in the database's ASSIGNMENT column;␣
↪to use
#   random set assignments, supply the `random` and `split` arguments:
sv.load_data(
    'my_data.csv',
    random=True,
    split=[0.7, 0.2, 0.1]
)
# 70% of data is in the learning set, 20% in the validation set and 10% in the test␣
↪set
```

ECNet utilizes console and file logging - by default, it will not log anything to either the console or a file. To enable logging functionality, we can use ECNet's logger:

```
from ecnet.utils.logging import logger

# Available levels are `debug`, `info`, `warn`, `error`, `crit`, `disable`

# Set the console log level to `info`:
logger.stream_level = 'info'

# Set the file log level to `info`:
logger.file_level = 'info'

# Specify a directory to save log files:
logger.log_dir = 'path\to\my\log\directory'
```

You can change all the model configuration variables from your Python script without having to edit and re-open your model configuration file:

```
# Configuration variables are found in the Server's '._vars' dictionary
sv._vars['learning_rate'] = 0.05
sv._vars['beta_2'] = 0.75
sv._vars['hidden_layers'] = [[32, 'relu'], [32, 'relu']]
sv._vars['epochs'] = 10000
```

Optimal input dimensionality, i.e. finding a balance between runtime and precision/accuracy, is often beneficial. To limit input dimensionality to a specified number of influential input parameters, ECNet utilizes random forest regression:

```
# Find the 15 most influential input parameters
sv.limit_inputs(15)

# Find the 15 most influential input parameters, and save them to an ECNet-formatted␣
↪database:
sv.limit_inputs(15, output_filename='my_limited_data.csv')
```

Optimal hyperparameters are essential for mapping inputs to outputs during neural network training. ECNet utilizes an artificial bee colony, ECabc, to optimize hyperparameters such as learning rate, beta, decay and epsilon values, and number of neurons per hidden layer:

```
# Tune hyperparameters for 50 iterations (search cycles) with 50 employer bees:
sv.tune_hyperparameters(50, 50)

# By default, all bees will use the same set assignments; to shuffle them:
sv.tune_hyperparameters(50, 50, shuffle='all', split=[0.7, 0.2, 0.1])
# To shuffle only the training sets (learning and validation), supply `train` instead

# By default, bees are evaluated on their performance across all sets; to specify a␣
↪set to
#   perform the evaluation:
sv.tune_hyperparameters(50, 50, eval_set='test')
# Available sets are `learn`, `valid`, `train`, `test`, None (all sets)

# The ABC will measure error using RMSE; to change the error function used:
sv.tune_hyperparameters(50, 50, eval_fn='mean_abs_error')
# Available functions are `rmse`, `mean_abs_error`, `med_abs_error`
```

ECNet is able to create an ensemble of neural networks (candidates chosen from pools) to predict a final value for the project. Projects can be saved and used at a later time.

```
# Create a project 'my_project' with 5 pools, 75 candidates per pool:
sv.create_project(
    'my_project',
    num_pools=5,
    num_candidates=75,
)

# Train neural networks using the number of epochs in your configuration file:
sv.train()

# To use periodic validation (training halts when validation set performance stops␣
↪improving),
#   supply the validate argument:
sv.train(validate=True)

# We can shuffle either 'train' (learning and validation) or 'all' sets with the␣
↪shuffle
```

(continues on next page)

(continued from previous page)

```
#   argument and a split:
sv.train(
    shuffle='train',
    split=[0.7, 0.2, 0.1]
)

# We can retrain pre-existing candidates:
sv.train(retrain=True)

# By default, best neural networks are selected from pools based on their performance␣
↪on all
#   sets; to specify a set used for evaluation:
sv.train(selection_set='test')
# Available sets are `learn`, `valid`, `train`, `test`, None (all sets)

# By default, candidates are evaluated by measuring their RMSE on the supplied set;␣
↪to specify
#   another error function:
sv.train(selection_fn='mean_abs_error')
# Available functions are `rmse`, `mean_abs_error`, `med_abs_error`

# Save your project
sv.save_project()

# You can save it with a name other than the one assigned
sv.save_project(filename='path/to/my/save.prj')

# When a project is saved, it will remove the folder structure it originated from; if␣
↪this is
#   unwanted:
sv.save_project(clean_up=False)

# A saved project contains all candidate neural networks, even if they have not been␣
↪selected;
#   to remove all non-chosen candidate neural networks:
sv.save_project(del_candidates=True)

# Predict values for the test set:
test_results = sv.use(dset='test')
# You can predict for 'learn', 'valid', 'train', 'test', or None (all) sets

# If you want to save these results to a CSV file, supply the output_filename argument
sv.use(dset='test', output_filename='my/test/results.csv')

# Calculates errors for the test set (any combination of these error functions can be␣
↪supplied as
#   arguments, and any dset listed above)
test_errors = sv.errors('rmse','r2','mean_abs_error','med_abs_error', dset='test')
```

Once you save a project, the .prj file can be used at a later time:

```
from ecnet.server import Server

# Specify a 'prj_file' argument to open a pre-existing project
sv = Server(prj_file='my_project.prj')

# Open an ECNet-formatted database with new data
```

(continues on next page)

(continued from previous page)

```
sv.load_data('new_data.csv')

# Save results to output file
#  - NOTE: no 'dset' argument for 'use_model' defaults to using all currently loaded␣
↪data
sv.use(output_filename='my/new/test/results.csv')
```

CHAPTER 3

ECNet Tools

3.1 Database creation

ECNet databases are comma-separated value (CSV) formatted files that provide information such as the ID of each data point, an optional explicit sort type, various strings and groups to identify data points, target values and input parameters. Row 1 is used to identify which columns are used for ID, explicit sorting assignment, various strings and groups, and target and input data, and row 2 contains the names of these strings/groups/targets/inputs. Additional rows are data points.

Our databases directory on GitHub contains databases for cetane number, cloud point, kinetic viscosity, pour point and yield sooting index, as well as a database template.

You can create an ECNet-formatted database with SMILES strings and (optionally) target values. Java JRE version 6 and above is required to create a database.

The database can then be constructed with:

```
from ecnet.tools.database import create_db

smiles_strings = ['CCC', 'CCCC', 'CCCCC']
targets = [0, 1, 2]

create_db(smiles_strings, 'my_database.csv', targets=targets)
```

Your database's DATAID column (essentially Bates numbers for each molecule) will increment starting at 0001. If a prefix is desired for these values, specify it with:

```
from ecnet.tools.database import create_db

smiles_strings = ['CCC', 'CCCC', 'CCCCC']
targets = [0, 1, 2]

create_db(smiles_strings, 'my_database.csv', targets=targets, id_prefix='MOL')
```

You may specify additional STRING columns:

```python
from ecnet.tools.database import create_db

smiles_strings = ['CCC', 'CCCC', 'CCCCC']
targets = [0, 1, 2]
extra_strings = {
    'Compound Name': ['Propane', 'n-Butane', 'Pentane'],
    'Literature Source': ['[1]', '[2]', '[3]']
}

create_db(
    smiles_strings,
    'my_database.csv',
    targets=targets,
    extra_strings=extra_strings
)
```

3.2 ECNet .prj file usage

Once an ECNet project has been created, the resulting .prj file can be used to predict properties for new molecules. Just supply SMILES strings, a pre-existing ECNet .prj file, and optionally a path to save the results to:

```python
from ecnet.tools.project import predict

smiles = ['CCC', 'CCCC']

# obtain results, do not save to file
results = predict(smiles, 'my_project.prj')

# obtain results, save to file
results = predict(smiles, 'my_project.prj', 'results.csv')
```

Java JRE 6.0+ is required for conversions.

3.3 Constructing parity plots

A common method for visualizing how well neural networks predict data is by utilizing a parity plot. A parity plot will show how much predictions deviate from experimental values by plotting them in conjunction with a 1:1 linear function (the closer a plot's data points are to this line, the better they perform).

To create a parity plot, let's import the ParityPlot object:

```python
from ecnet.tools.plotting import ParityPlot
```

And initialize it:

```python
my_plot = ParityPlot()

# The plot's title defaults to `Parity Plot`; let's change that:
my_plot = ParityPlot(title='Cetane Number Parity Plot')

# The plot's axes default to `Experimental Value` (x-axis) and `Predicted Value`
#    (y-axis); we can change those too:
my_plot = ParityPlot(
```

(continues on next page)

(continued from previous page)

```
    title='Cetane Number Parity Plot',
    x_label='Experimental CN',
    y_label='Predicted CN'
)

# The plot's font is Times New Roman by default; to use another font:
my_plot = ParityPlot(
    title='Cetane Number Parity Plot',
    x_label='Experimental CN',
    y_label='Predicted CN',
    font='Calibri'
)
```

Now that our plot is initialized, we can add data:

```
my_plot.add_series(x_vals, y_vals)
```

Say, for example, we obtained results from ECNet's Server object using the "use" method; let's plot predicted vs. experimental for the test set:

```
'''Let's assume you've trained your model using a Server, `sv`'''

# Obtain predictions for data in the test set:
predicted_data = sv.use(dset='test')
experimental_data = sv._sets.test_y

# Pass the test data's experimental values and its predicted values:
my_plot.add_series(experimental_data, predicted_data)

# We can also asign a name to the series, and change its color:
my_plot.add_series(
    experimental_data,
    predicted_data,
    name='Test Set',
    color='red'
)
```

Multiple data series can be added to your plot, allowing you to visualize different data sets together.

If we want to visualize how well given data points perform with respect to an error metric, we can add error bars to the plot. These error bars are placed on the positive and negative side of the 1:1 parity line:

```
'''Let's assume you've trained your model using a Server, `sv`'''

# Obtain the test set's RMSE:
errors = sv.errors('rmse', dset='test')

# Add the error bars:
my_plot.add_error_bars(errors['rmse'])

# We can show the value of the error by supplying:
my_plot.add_error_bars(errors['rmse'], label='Test Set RMSE')
```

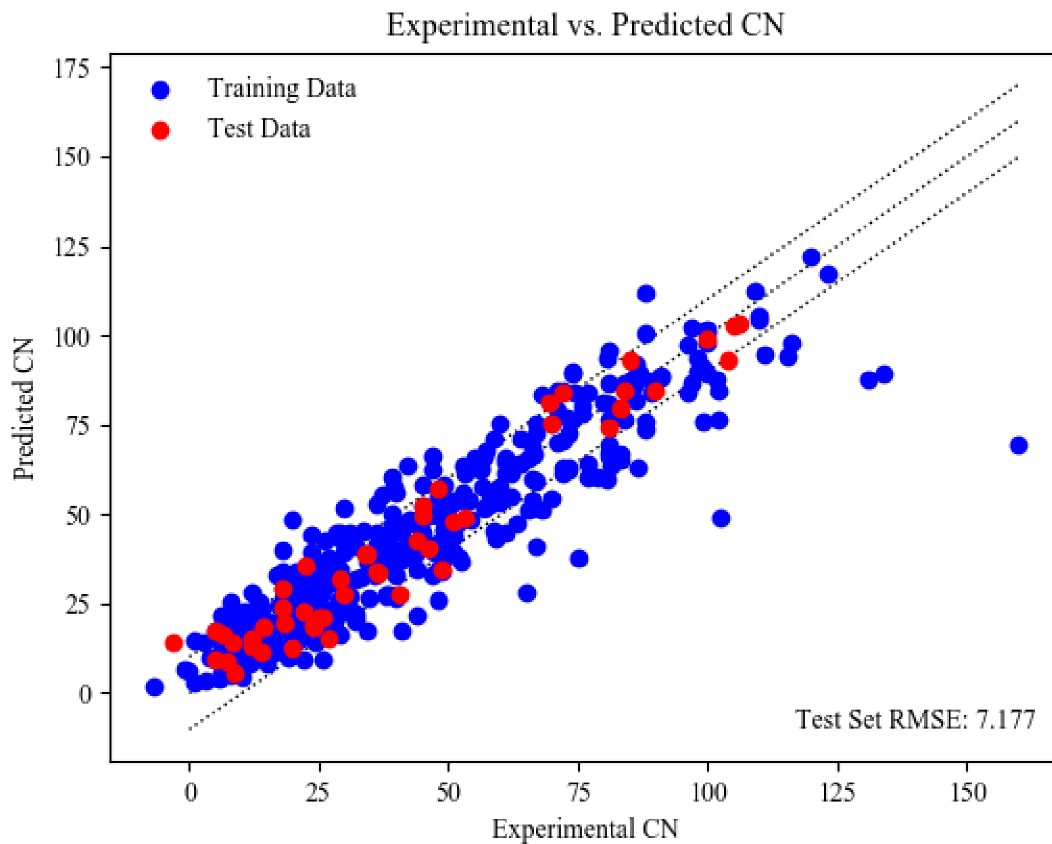
Once the plot is complete, it can be saved:

```
# Save the plot to `my_plot.png`:
my_plot.save('my_plot.png')
```

Or, we can view it without saving:

```
# View the plot in a pop-up window:
my_plot.show()
```

Here is what plotting cetane number training and test data looks like:



CHAPTER 4

Example Scripts

4.1 SMILES String Validation

SMILES strings are the basis for QSPR descriptor generation, and therefore play an immense role in what neural networks learn (and how they correlate QSPR descriptors to given fuel properties). It is paramount that SMILES strings for molecules are correct to ensure neural networks learn from correct molecule representations.

To validate SMILES strings for molecules stored in an ECNet-formatted database, we can use the script below to query PubChem using molecule names. The "validate_smiles" function accepts two arguments, the database you wish to validate and the filename of the resulting validated database. Note that QSPR descriptors in the resulting database do not reflect changes made to SMILES strings, and you will need to create a new database using our database construction tool to generate new descriptors.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# Developed in 2019 by Travis Kessler <travis.j.kessler@gmail.com>
#
# Example script for validating ECNet-formatted database SMILES strings
#

from ecnet.utils.data_utils import DataFrame
from ecnet.tools.database import get_smiles
from ecnet.utils.logging import logger


def validate_smiles(db_name, new_db):

    # load the database
    logger.log('info', 'Loading data from {}'.format(db_name))
    df = DataFrame(db_name)

    # check each molecule's SMILES, replace if incorrect
    for pt in df.data_points:
```

(continues on next page)

(continued from previous page)

```
        mol_name = getattr(pt, 'Compound Name')
        smiles = get_smiles(mol_name)
        if len(smiles) == 0:
            logger.log('warn', '{} not found on PubChem'.format(mol_name))
            continue
        else:
            if pt.SMILES not in smiles:
                logger.log(
                    'crit',
                    'Incorrect SMILES for {}:\n\tDatabase SMILES: {}'
                    '\n\tPubChem SMILES: {}'.format(
                        mol_name,
                        pt.SMILES,
                        smiles
                    ))
                pt.SMILES = smiles[0]
            else:
                logger.log('info', 'Correct SMILES for {}'.format(mol_name))

    # save the validated database
    logger.log('info', 'Saving validated data to {}'.format(new_db))
    df.save(new_db)
    return


if __name__ == '__main__':

    # initialize logging
    logger.stream_level = 'info'
    # un-comment this for file logging
    # logger.file_level = 'info'

    validate_smiles('unvalidated_db.csv', 'validated_db.csv')
```