mavenn Documentation

Release 1.0

Ammar Tareen and Justin B. Kinney

Mar 21, 2022

TABLE OF CONTENTS

MAVE-NN[1] enables the rapid quantitative modeling of genotype-phenotype (G-P) maps from the data produced by multiplex assays of variant effect (MAVEs). Such assays include deep mutational scanning (DMS) experiments on proteins, massively parallel reporter assays (MPRAs) on DNA or RNA regulatory sequences, and more. MAVE-NN conceptualizes G-P map inference as a problem in information compression; this problem is then solved by training a neural network using a TensorFlow backend. To learn more about this modeling strategy, please see our bioRxiv preprint.

MAVE-NN is written for Python 3 and is provided under an MIT open source license. The documentation provided here is meant to help users quickly get MAVE-NN working for their own research needs. Please do not hesitate to contact us with any questions or suggestions for improvements. For technical assistance or to report bugs, please contact Ammar Tareen (Email: tareen@cshl.edu, Twitter: @AmmarTareen1) . For more general correspondence, please contact Justin Kinney (Email: jkinney@cshl.edu, Twitter: @jbkinney).

[1] Tareen A, Kooshkbaghi M, Posfai A, Ireland WT, McCandlish DM, Kinney JB. MAVE-NN: learning genotype-phenotype maps from multiplex assays of variant effect Biorxiv (2020). https://doi.org/10.1101/2020.07.14.201475

CHAPTER

ONE

INSTALLATION INSTRUCTIONS

using the `pip` package manager by executing the following at the commandline:

```
$ pip install mavenn
```

Alternatively, you can clone MAVE-NN from GitHub by doing this at the command line:

```
$ cd appropriate_directory
$ git clone https://github.com/jbkinney/mavenn.git
```

where `appropriate_directory` is the absolute path to where you would like MAVE-NN to reside. Then add this to the top of any Python file in which you use MAVE-NN:

```
# Insert local path to MAVE-NN at beginning of Python's path
import sys
sys.path.insert(0, 'appropriate_directory/mavenn')

#Load mavenn
import mavenn
```

CHAPTER

TWO

MODELING TUTORIALS

MAVE-NN comes with a variety of pre-trained models that users can load and apply. Models similar to these can be trained using the following notebooks

2.1 Tutorial 1: Built-in demonstration scripts

MAVE-NN provides built-in demonstration scripts, or "demos", to help users quickly get started training and visualizing models. Demos are self-contained Python scripts that can be executed by calling `mavenn.run_demo`. To get a list of demo names, execute this function without passing any arguments:

[1]:
```
# Import MAVE-NN
import mavenn

# Get list of demos
mavenn.run_demo()
```

```
To run a demo, execute

        >>> mavenn.run_demo(name)

where 'name' is one of the following strings:

        1. "gb1_ge_evaluation"
        2. "mpsa_ge_training"
        3. "sortseq_mpa_visualization"

Python code for each demo is located in

        /Users/jkinney/github/mavenn/mavenn/examples/demos/
```

[1]:
```
['gb1_ge_evaluation', 'mpsa_ge_training', 'sortseq_mpa_visualization']
```

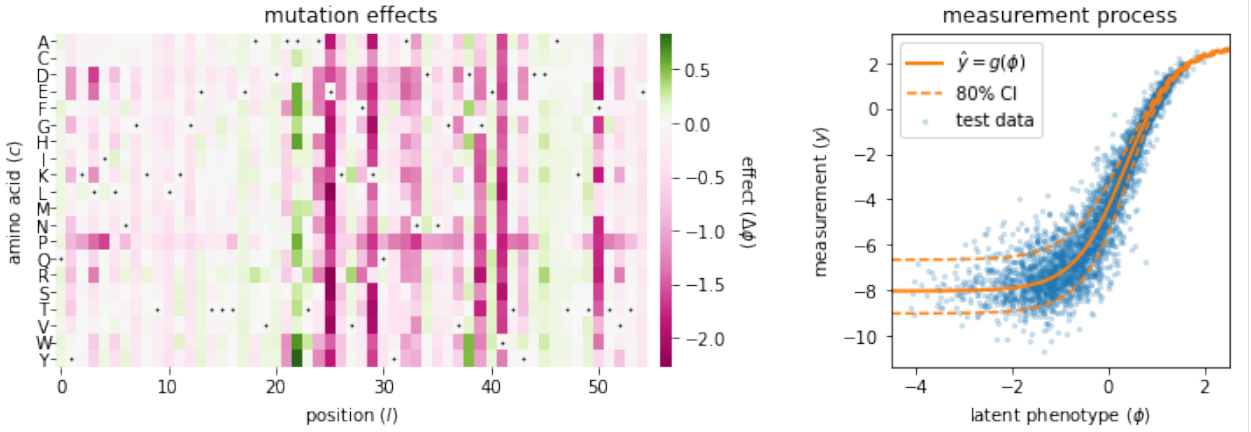
To see the Python code for any one of these demos, pass the keyword argument `print_code=True` to `mavenn.run_demo()`. Alternatively, navigate to the folder that is printed when executing `mavenn.run_demo()` on your machine and open the corresponding `*.py` file.

2.1.1 Evaluating a GE regression model

The `'gb1_ge_evaluation'` demo illustrates an additive G-P map and GE measurement process fit to data from a deep mutational scanning (DMS) experiment performed on protein GB1 by Olson et al., 2014.

[2]:
```
mavenn.run_demo('gb1_ge_evaluation', print_code=False)
```

```
Running /Users/jkinney/github/mavenn/mavenn/examples/demos/gb1_ge_evaluation.py...
Using mavenn at: /Users/jkinney/github/mavenn/mavenn
Model loaded from these files:
        /Users/jkinney/github/mavenn/mavenn/examples/models/gb1_ge_additive.pickle
        /Users/jkinney/github/mavenn/mavenn/examples/models/gb1_ge_additive.h5
```



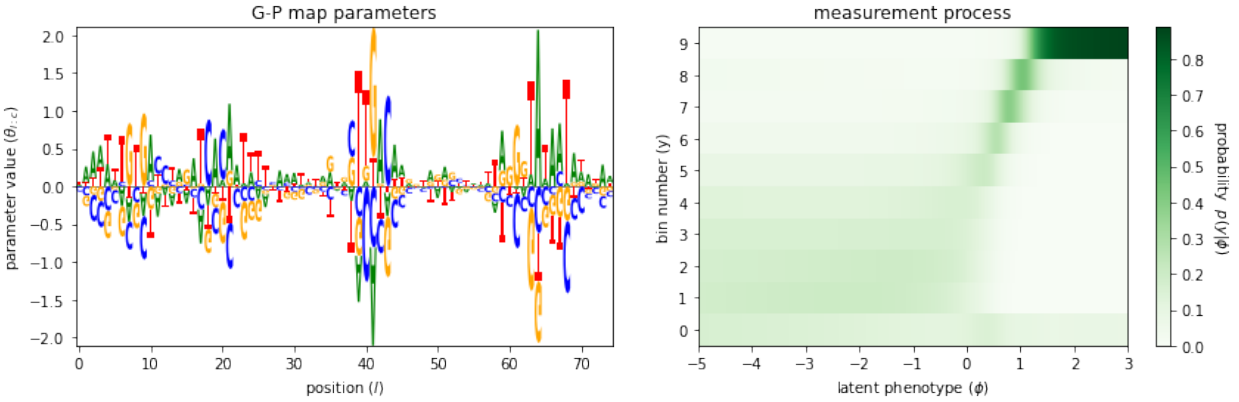

```
Done!
```

2.1.2 Visualizing an MPA regression model

The `'sortseq_mpa_visualization'` demo illustrates an additive G-P map, along with an MPA measurement process, fit to data from a sort-seq MPRA performed by Kinney et al., 2010.

[3]:
```
mavenn.run_demo('sortseq_mpa_visualization', print_code=False)
```

```
Running /Users/jkinney/github/mavenn/mavenn/examples/demos/sortseq_mpa_visualization.
↪py...
Using mavenn at: /Users/jkinney/github/mavenn/mavenn
Model loaded from these files:
        /Users/jkinney/github/mavenn/mavenn/examples/models/sortseq_full-wt_mpa_additive.
↪pickle
        /Users/jkinney/github/mavenn/mavenn/examples/models/sortseq_full-wt_mpa_additive.
↪h5
```



Done!

2.1.3 Training a GE regression model

The `'mpsa_ge_training'` demo uses GE regression to train a pairwise G-P map on data from a massively parallel splicing assay (MPSA) reported by Wong et al., 2018. This training process usually takes under a minute on a standard laptop.

[4]:
```
mavenn.run_demo('mpsa_ge_training', print_code=False)
```

```
Running /Users/jkinney/github/mavenn/mavenn/examples/demos/mpsa_ge_training.py...
Using mavenn at: /Users/jkinney/github/mavenn/mavenn
N = 24,405 observations set as training data.
Using 19.9% for validation.
Data shuffled.
Time to set data: 0.208 sec.

LSMR            Least-squares solution of  Ax = b

The matrix A has 19540 rows and 36 columns
damp = 0.00000000000000e+00

atol = 1.00e-06                 conlim = 1.00e+08

btol = 1.00e-06             maxiter =       36


   itn      x(1)       norm r    norm Ar  compatible   LS      norm A   cond A
     0  0.00000e+00  1.391e+02  4.673e+03    1.0e+00  2.4e-01
     1  2.09956e-04  1.273e+02  2.143e+03    9.2e-01  2.1e-01  8.1e+01  1.0e+00
     2  4.17536e-03  1.263e+02  1.369e+03    9.1e-01  5.1e-02  2.1e+02  1.9e+00
     3  3.65731e-03  1.251e+02  5.501e+01    9.0e-01  1.6e-03  2.8e+02  2.6e+00
     4  3.27390e-03  1.251e+02  6.185e+00    9.0e-01  1.7e-04  2.9e+02  3.2e+00
     5  3.20902e-03  1.251e+02  3.988e-01    9.0e-01  1.1e-05  3.0e+02  3.4e+00
     6  3.21716e-03  1.251e+02  1.126e-02    9.0e-01  3.0e-07  3.0e+02  3.4e+00

LSMR finished
The least-squares solution is good enough, given atol
istop =       2    normr = 1.3e+02
```

(continues on next page)

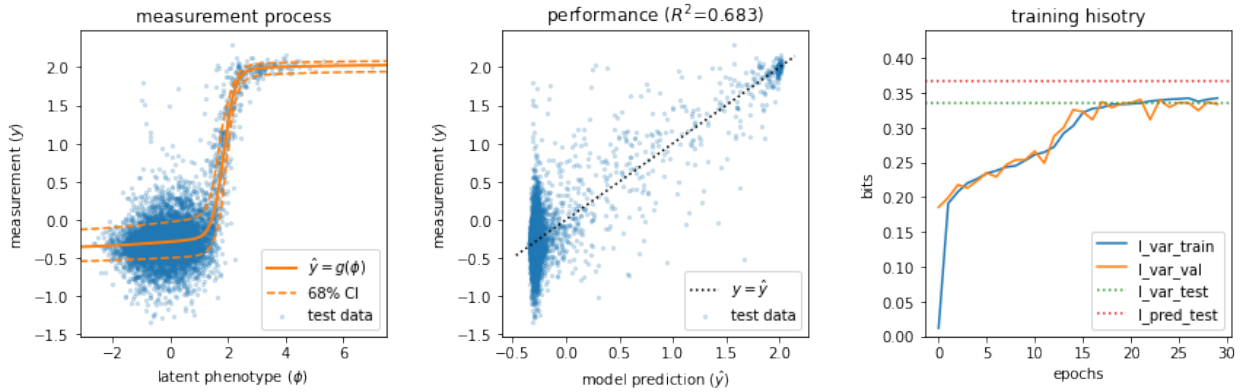
(continued from previous page)

```
    normA = 3.0e+02    normAr = 1.1e-02
itn   =       6    condA = 3.4e+00
    normx = 8.2e-01
     6  3.21716e-03   1.251e+02  1.126e-02
   9.0e-01  3.0e-07   3.0e+02  3.4e+00
Linear regression time: 0.0044 sec
Epoch 1/30
391/391 [==============================] - 1s 1ms/step - loss: 48.0562 - I_var: 0.0115 -␣
↪val_loss: 41.2422 - val_I_var: 0.1856
Epoch 2/30
391/391 [==============================] - 0s 686us/step - loss: 41.0110 - I_var: 0.1912␣
↪- val_loss: 40.7058 - val_I_var: 0.1993
Epoch 3/30
391/391 [==============================] - 0s 682us/step - loss: 40.3925 - I_var: 0.2076␣
↪- val_loss: 40.0481 - val_I_var: 0.2178
Epoch 4/30
391/391 [==============================] - 0s 684us/step - loss: 39.9577 - I_var: 0.2203␣
↪- val_loss: 40.1940 - val_I_var: 0.2132
Epoch 5/30
391/391 [==============================] - 0s 687us/step - loss: 39.7409 - I_var: 0.2264␣
↪- val_loss: 39.8678 - val_I_var: 0.2234
Epoch 6/30
391/391 [==============================] - 0s 690us/step - loss: 39.4834 - I_var: 0.2343␣
↪- val_loss: 39.4798 - val_I_var: 0.2351
Epoch 7/30
391/391 [==============================] - 0s 688us/step - loss: 39.3679 - I_var: 0.2381␣
↪- val_loss: 39.6919 - val_I_var: 0.2293
Epoch 8/30
391/391 [==============================] - 0s 688us/step - loss: 39.2042 - I_var: 0.2433␣
↪- val_loss: 39.1096 - val_I_var: 0.2462
Epoch 9/30
391/391 [==============================] - 0s 691us/step - loss: 39.1603 - I_var: 0.2452␣
↪- val_loss: 38.8807 - val_I_var: 0.2537
Epoch 10/30
391/391 [==============================] - 0s 701us/step - loss: 38.9189 - I_var: 0.2529␣
↪- val_loss: 38.8879 - val_I_var: 0.2537
Epoch 11/30
391/391 [==============================] - 0s 709us/step - loss: 38.6567 - I_var: 0.2612␣
↪- val_loss: 38.4904 - val_I_var: 0.2662
Epoch 12/30
391/391 [==============================] - 0s 686us/step - loss: 38.5589 - I_var: 0.2648␣
↪- val_loss: 39.1242 - val_I_var: 0.2490
Epoch 13/30
391/391 [==============================] - 0s 682us/step - loss: 38.3520 - I_var: 0.2723␣
↪- val_loss: 37.8326 - val_I_var: 0.2877
Epoch 14/30
391/391 [==============================] - 0s 688us/step - loss: 37.7298 - I_var: 0.2920␣
↪- val_loss: 37.4434 - val_I_var: 0.3001
Epoch 15/30
391/391 [==============================] - 0s 690us/step - loss: 37.3966 - I_var: 0.3027␣
↪- val_loss: 36.5743 - val_I_var: 0.3259
Epoch 16/30
```

(continues on next page)

(continued from previous page)

```
391/391 [==============================] - 0s 690us/step - loss: 36.7566 - I_var: 0.3217␣
↪- val_loss: 36.6817 - val_I_var: 0.3231
Epoch 17/30
391/391 [==============================] - 0s 707us/step - loss: 36.5558 - I_var: 0.3276␣
↪- val_loss: 37.0841 - val_I_var: 0.3117
Epoch 18/30
391/391 [==============================] - 0s 699us/step - loss: 36.4968 - I_var: 0.3290␣
↪- val_loss: 36.1766 - val_I_var: 0.3373
Epoch 19/30
391/391 [==============================] - 0s 691us/step - loss: 36.3217 - I_var: 0.3340␣
↪- val_loss: 36.4613 - val_I_var: 0.3290
Epoch 20/30
391/391 [==============================] - 0s 687us/step - loss: 36.3226 - I_var: 0.3337␣
↪- val_loss: 36.2868 - val_I_var: 0.3347
Epoch 21/30
391/391 [==============================] - 0s 702us/step - loss: 36.3051 - I_var: 0.3345␣
↪- val_loss: 36.2097 - val_I_var: 0.3359
Epoch 22/30
391/391 [==============================] - 0s 690us/step - loss: 36.2569 - I_var: 0.3356␣
↪- val_loss: 36.0617 - val_I_var: 0.3406
Epoch 23/30
391/391 [==============================] - 0s 688us/step - loss: 36.1654 - I_var: 0.3384␣
↪- val_loss: 37.0675 - val_I_var: 0.3118
Epoch 24/30
391/391 [==============================] - 0s 697us/step - loss: 36.1285 - I_var: 0.3395␣
↪- val_loss: 36.0666 - val_I_var: 0.3403
Epoch 25/30
391/391 [==============================] - 0s 694us/step - loss: 36.0856 - I_var: 0.3409␣
↪- val_loss: 36.4473 - val_I_var: 0.3296
Epoch 26/30
391/391 [==============================] - 0s 688us/step - loss: 36.0640 - I_var: 0.3416␣
↪- val_loss: 36.2272 - val_I_var: 0.3363
Epoch 27/30
391/391 [==============================] - 0s 699us/step - loss: 36.0285 - I_var: 0.3426␣
↪- val_loss: 36.2458 - val_I_var: 0.3356
Epoch 28/30
391/391 [==============================] - 0s 703us/step - loss: 36.2016 - I_var: 0.3377␣
↪- val_loss: 36.5967 - val_I_var: 0.3250
Epoch 29/30
391/391 [==============================] - 0s 689us/step - loss: 36.0952 - I_var: 0.3408␣
↪- val_loss: 36.1920 - val_I_var: 0.3369
Epoch 30/30
391/391 [==============================] - 0s 699us/step - loss: 36.0361 - I_var: 0.3427␣
↪- val_loss: 36.3018 - val_I_var: 0.3338
Training time: 9.0 seconds
I_var_test: 0.335 +- 0.024 bits
I_pred_test: 0.367 +- 0.016 bits
```



Done!

2.1.4 References

1. Kinney J, Murugan A, Callan C, Cox E. Using deep sequencing to characterize the biophysical mechanism of a transcriptional regulatory sequence. Proc Natl Acad Sci USA. 107:9158-9163 (2010).
2. Olson CA, Wu NC, Sun R. A comprehensive biophysical description of pairwise epistasis throughout an entire protein domain. Curr Biol 24:2643–2651 (2014).
3. Wong M, Kinney J, Krainer A. Quantitative activity profile and context dependence of all human 5' splice sites. Mol Cell 71:1012-1026.e3 (2018).

[ ]:

2.2 Tutorial 2: Protein DMS modeling using additive G-P maps

This tutorial covers perhaps the simplest application of MAVE-NN: the modeling of DMS data using an additive genotype-phenotype (G-P) map together with a global epistasis (GE) measurement process. The code below steps users through this process, and can be used to train models similar to the following built-in models, which are accessible using `mavenn.load_example_model()`:

- `'amyloid_additive_ge'`
- `'tdp43_additive_ge'`
- `'gb1_additive_ge'`

[1]:
```
# Standard imports
import numpy as np
import matplotlib.pyplot as plt

# Import MAVE-NN
import mavenn
```

2.2.1 Training

First we choose which dataset we wish to model, and we load it as a Pandas dataframe using `mavenn.load_example_dataset()`. We then compute the length of sequences in that dataset; we will need this quantity for defining the architecture of our model.

[2]:
```
# Choose dataset
dataset_name = 'amyloid'  # From Seuma et al., 2021
# dataset_name = 'tdp43'    # From Bolognesi et al., 2019
# dataset_name = 'gb1'      # From Olson et al., 2021. (Slowest)
print(f"Loading dataset '{dataset_name}' ")

# Load datset
data_df = mavenn.load_example_dataset(dataset_name)

# Get and report sequence length
L = len(data_df.loc[0,'x'])
print(f'Sequence length: {L:d} amino acids (+ stops)')

# Preview dataset
print('data_df:')
data_df
```

```
Loading dataset 'amyloid'
Sequence length: 42 amino acids (+ stops)
data_df:
```

[2]:

	set	dist	y	dy	x
0	training	1	-0.117352	0.387033	KAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
1	training	1	0.352500	0.062247	NAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
2	training	1	-2.818013	1.068137	TAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
3	training	1	0.121805	0.376764	SAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
4	training	1	-2.404340	0.278486	IAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
...	...	...	...	...	...
16061	training	2	-0.151502	0.389821	DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVKV
16062	training	2	-1.360708	0.370517	DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVLV
16063	training	2	-0.996816	0.346949	DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVMV
16064	training	2	-3.238403	0.429008	DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVTV
16065	training	2	-1.141457	0.365638	DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVVV

[16066 rows x 5 columns]

Next we use the built-in function `mavenn.split_dataset()` to split our dataset into two dataframes:

- `trainval_df`, which contains both the **training set** and **validation set* data.
- `test_df`, which contains only the **test set** data.

Note that `training_df`, previewed below, includes an extra column called `'validation'` that flags which sequences are reserved for the validation set.

[3]:
```
# Split dataset
trainval_df, test_df = mavenn.split_dataset(data_df)

# Preview trainval_df
print('trainval_df:')
trainval_df
```

```
Training set   :   14,481 observations (  90.13%)
Validation set :      826 observations (   5.14%)
Test set       :      759 observations (   4.72%)
-------------------------------------------------
Total dataset  :   16,066 observations ( 100.00%)

trainval_df:
```

[3]:

	validation	dist	y	dy	x
0	False	1	-0.117352	0.387033	KAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
1	False	1	0.352500	0.062247	NAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
2	False	1	-2.818013	1.068137	TAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
3	False	1	0.121805	0.376764	SAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
4	False	1	-2.404340	0.278486	IAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
...	...	...	...	...	...
15302	False	2	-0.151502	0.389821	DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVKV
15303	False	2	-1.360708	0.370517	DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVLV
15304	False	2	-0.996816	0.346949	DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVMV
15305	False	2	-3.238403	0.429008	DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVTV
15306	False	2	-1.141457	0.365638	DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVVV

[15307 rows x 5 columns]

Now we specify the architecture of the model we wish to train . To do this, we create an instance of the `mavenn.Model` class, called `model`, using the following keyword arguments:

- L=L specifies the sequence length.

- `alphabet='protein*'` specifies that the alphabet our sequences are built from consists of 21 characters representing the 20 amino acids plus a stop signal (which is represented by the character `'*'`). Other possible choices for `alphabet` are `'dna'`, `'rna'`, and `'protein'`.
- `gpmap_type='additive'` specifies that we wish to infer an additive G-P map.
- `regression_type='GE'` specifies that our model will have a global epistasis (GE) measurement process. We choose this because our MAVE measurements are continuous real numbers.
- `ge_noise_model_type='SkewedT'` specifies the use of a skewed-t noise model in the GE measurement process. The `'SkewedT'` noise model can accommodate asymmetric noise and is thus more flexible than the default `'Gaussian'` noise model.
- `ge_heteroskedasticity_order=2` specifies that the noise model parameters (the three parameters of the skewed-t distribution) are each modeled using quadratic functions of the predicted measurement $\hat{y}$. This will allow our model of experimental noise to vary with signal intensity.

We then set the training data by calling `model.set_data()`. The keyword argument `'validation_flags'` is used to specify which subset of the data in `trainval_df` will be used for validation (as opposed to stochastic gradient descent).

Next we train the model by calling `model.fit()`. In doing so we specify a number of hyperparameters including the learning rate, the number of epochs, the batch size, whether to use early stopping, and the early stopping patience. We also set `verbose=False` to limit the amount of user feedback.

Choosing hyperparameters is somewhat of an art, and the particular values used here were found by trial and error. In general users will have to try a number of different values for these and possibly other hyperparameters in order to find ones that work well. We recommend that users choose these hyperparameters in order to maximize the final value for `val_I_var`, the variational information of the trained model on the validation dataset.

[4]:
```
# Define model
model = mavenn.Model(L=L,
                     alphabet='protein*',
                     gpmap_type='additive',
                     regression_type='GE',
                     ge_noise_model_type='SkewedT',
                     ge_heteroskedasticity_order=2)

# Set training data
model.set_data(x=trainval_df['x'],
               y=trainval_df['y'],
               validation_flags=trainval_df['validation'])

# Train model
model.fit(learning_rate=1e-3,
          epochs=500,
          batch_size=64,
          early_stopping=True,
          early_stopping_patience=25,
          verbose=False);
```

```
N = 15,307 observations set as training data.
Using 5.4% for validation.
Data shuffled.
Time to set data: 0.297 sec.
```

```
0epoch [00:00, ?epoch/s]
```

```
Training time: 15.0 seconds
```

To assess the performance of our final trained model, we compute two different metrics on test data: **variational information** and **predictive information**. Variational information quantifies the performance of the full latent phenotype model (G-P map + measurement process), whereas predictive information quantifies the performance of just the G-P map. See Tareen et al. (2021) for an expanded discussion of these quantities.

Note that MAVE-NN also estimates the standard errors for these quantities.

[5]:
```
# Compute variational information on test data
I_var, dI_var =  model.I_variational(x=test_df['x'], y=test_df['y'])
print(f'test_I_var: {I_var:.3f} +- {dI_var:.3f} bits')

# Compute predictive information on test data
I_pred, dI_pred = model.I_predictive(x=test_df['x'], y=test_df['y'])
print(f'test_I_pred: {I_pred:.3f} +- {dI_pred:.3f} bits')
```

```
test_I_var: 1.111 +- 0.068 bits
test_I_pred: 1.196 +- 0.049 bits
```

To save the trained model we call `model.save()`. This records our model in **two separate files**: a pickle file that defines model architecture (extension `'.pickle'`), and an H5 file that records model parameters (extension `'.h5'`).

[6]:
```
# Save model to file
model_name = f'{dataset_name}_additive_ge'
model.save(model_name)
```

```
Model saved to these files:
        amyloid_additive_ge.pickle
        amyloid_additive_ge.h5
```

2.2.2 Visualization

We now discuss how to visualize the training history, performance, and parameters of a trained model. First we then load our model using `mavenn.load`:

[7]:
```
# Delete model if it is present in memory
try:
    del model
except:
    pass

# Load model from file
model = mavenn.load(model_name)
```

```
Model loaded from these files:
        amyloid_additive_ge.pickle
        amyloid_additive_ge.h5
```

The `model.history` attribute is a dict that contains values for multiple model performance metrics as a function of training epoch, each evaluated on both the training data and the validation data. `'loss'` and `'val_loss'` record loss values, while `'I_var'` and `'val_I_var'` record the variational information values. As described in Tareen et al. (2021), these metrics are closely related: variational information is an affine transformation of the per-datum log likelihood, whereas loss is equal to negative log likelihood plus regularization terms.

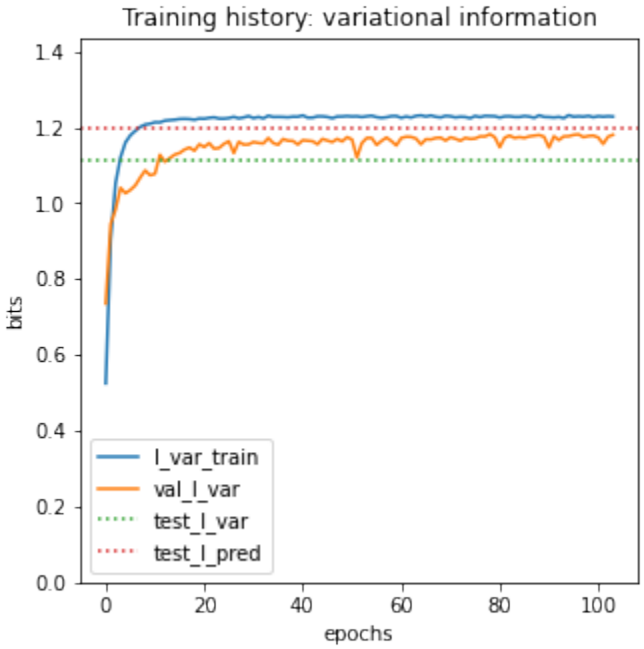
[8]:
```
# Show metrics recorded in model.history()
model.history.keys()
```

[8]:
```
dict_keys(['loss', 'I_var', 'val_loss', 'val_I_var'])
```

Plotting `'I_var'` and `'val_I_var'` versus epoch can often provide insight into the model training process.

[9]:
```
# Create figure and axes for plotting
fig, ax = plt.subplots(1,1,figsize=[5,5])

# Plot I_var_train, the variational information on training data as a function of epoch
ax.plot(model.history['I_var'],
        label=r'I_var_train')

# Plot I_var_val, the variational information on validation data as a function of epoch
ax.plot(model.history['val_I_var'],
        label=r'val_I_var')

# Show I_var_test, the variational information of the final model on test data
ax.axhline(I_var, color='C2', linestyle=':',
           label=r'test_I_var')

# Show I_pred_test, the predictive information of the final model on test data
ax.axhline(I_pred, color='C3', linestyle=':',
           label=r'test_I_pred')

# Style plot
ax.set_xlabel('epochs')
ax.set_ylabel('bits')
ax.set_title('Training history: variational information')
ax.set_ylim([0, 1.2*I_pred])
ax.legend()
```

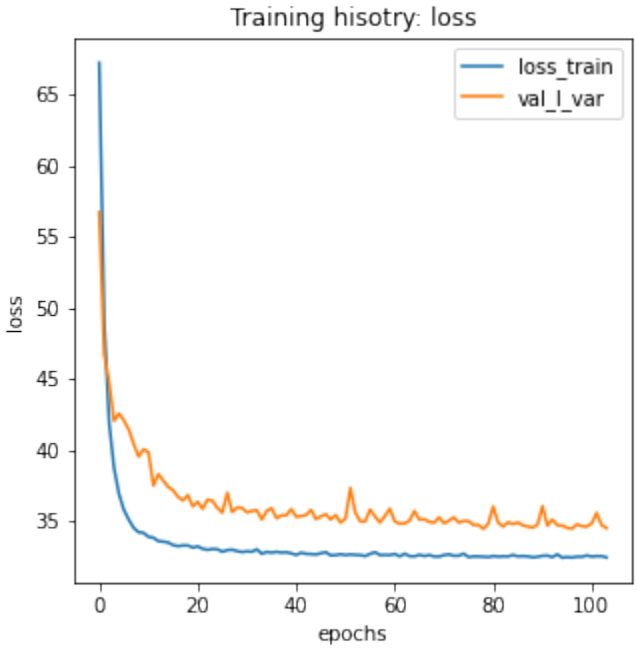
[9]:
```
<matplotlib.legend.Legend at 0x17fec1970>
```



Users can also plot 'loss' and 'var_loss' if they like, though the absolute values these quantities are be more difficult to interpret than `'I_var'` and `'val_I_var'`.

[10]:
```
# Create figure and axes for plotting
fig, ax = plt.subplots(1,1,figsize=[5,5])

# Plot loss_train, the loss computed on training data as a function of epoch
ax.plot(model.history['loss'],
        label=r'loss_train')

# Plot loss_val, the loss computed on validation data as a function of epoch
ax.plot(model.history['val_loss'],
        label=r'val_I_var')

# Style plot
ax.set_xlabel('epochs')
ax.set_ylabel('loss')
ax.set_title('Training hisotry: loss')
ax.legend()
```

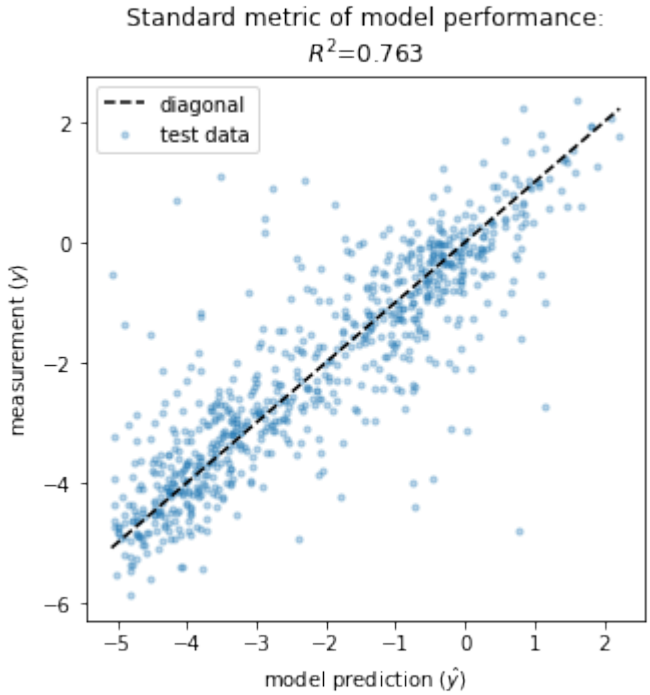
[10]:
```
<matplotlib.legend.Legend at 0x297279af0>
```



It is also useful to consider more traditional metrics of model performance. In the context of GE models, a natural choice is the squared Pearson correlation, R^2, between measurements y and model predictions $\hat{y}$:

[11]:
```
# Create figure and axes for plotting
fig, ax = plt.subplots(1,1,figsize=[5,5])

# Get test data y values
y_test = test_df['y']

# Compute yhat on test data
yhat_test = model.x_to_yhat(test_df['x'])

# Compute R^2 between yhat_test and y_test
Rsq = np.corrcoef(yhat_test.ravel(), test_df['y'])[0, 1]**2

# Plot y_test vs. yhat_test
ax.scatter(yhat_test, y_test, color='C0', s=10, alpha=.3,
           label='test data')

# Style plot
xlim = [min(yhat_test), max(yhat_test)]
ax.plot(xlim, xlim, '--', color='k', label='diagonal', zorder=100)
ax.set_xlabel('model prediction ($\hat{y}$)')
ax.set_ylabel('measurement ($y$)')
ax.set_title(f'Standard metric of model performance:\n$R^2$={Rsq:.3}');
ax.legend()
```

[11]:
```
<matplotlib.legend.Legend at 0x2972e10d0>
```



Next we visualize the GE measurement process inferred as part of our latent phenotype model. Recall from Tareen et al. (2021) that the measurement process consists of

- A nonlinearity $\hat{y} = g(\phi)$ that deterministically maps the latent phenotype ϕ to a prediction $\hat{y}$.
- A noise model $p(y|\hat{y})$ that stochastically maps predictions $\hat{y}$ to measurements y.

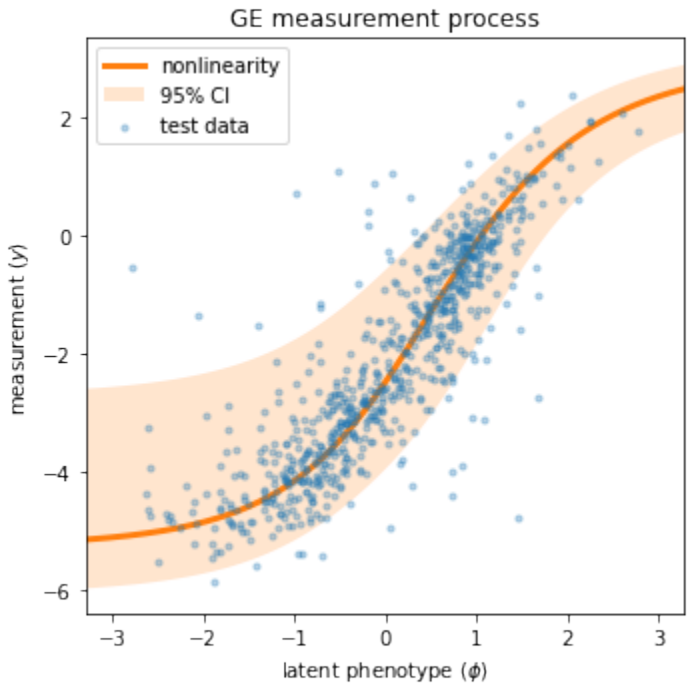
We can conveniently visualize both of these quantities in a single "global epistatsis plot":

[12]:
```
# Create figure and axes for plotting
fig, ax = plt.subplots(1,1,figsize=[5,5])

# Get test data y values
y_test = test_df['y']

# Compute  on test data
phi_test = model.x_to_phi(test_df['x'])

## Set phi lims and create a grid in phi space
phi_lim = [min(phi_test)-.5, max(phi_test)+.5]
phi_grid = np.linspace(phi_lim[0], phi_lim[1], 1000)

# Compute yhat each phi gridpoint
yhat_grid = model.phi_to_yhat(phi_grid)

# Compute 95% CI for each yhat
q = [0.025, 0.975]
yqs_grid = model.yhat_to_yq(yhat_grid, q=q)


# Plote 95% confidence interval
```

(continues on next page)

(continued from previous page)

```
ax.fill_between(phi_grid, yqs_grid[:, 0], yqs_grid[:, 1],
                alpha=0.2, color='C1', lw=0, label='95% CI')

# Plot GE nonlinearity
ax.plot(phi_grid, yhat_grid,
        linewidth=3, color='C1', label='nonlinearity')

# Plot scatter of  and y values,
ax.scatter(phi_test, y_test,
           color='C0', s=10, alpha=.3, label='test data',
           zorder=+100, rasterized=True)

# Style plot
ax.set_xlim(phi_lim)
ax.set_xlabel('latent phenotype ($\phi$)')
ax.set_ylabel('measurement ($y$)')
ax.set_title('GE measurement process')
ax.legend()
fig.tight_layout()
```



To retrieve the values of our model's G-P map parameters, we use the method `model.get_theta()`. This returns a dictionary:

[13]:
```
# Retrieve G-P map parameter dict and view dict keys
theta_dict = model.get_theta(gauge='consensus')
theta_dict.keys()
```

[13]:
```
dict_keys(['L', 'C', 'alphabet', 'theta_0', 'theta_lc', 'theta_lclc', 'theta_mlp',
↪'logomaker_df'])
```

It is important to appreciate that G-P maps usually have many non-identifiable directions in parameter space. These

are called **gauge freedoms**. Interpreting the values of model parameters requires that we first "pin down" these gauge freedoms by using a clearly specified convension. Specifying `gauge='consensus'` in `model.get_theta()` accomplishes this fixing all the $\theta_{l:c}$ parameters that contribute to the consensus sequence to zero. This convension allows all the other $\theta_{l:c}$ parameters in the additive model to be interpreted as single-mutation effects, $\Delta\phi$, away from the consensus sequence.

Finally, we use `mavenn.heatmap()` to visualize these additive parameters. This function takes a number of keyword arguments, which we summarize here. More information can be found in this function's docstring.

- `ax=ax`: specifies the axes on which to draw both the heatmap and the colorbar.
- `values=theta_dict['theta_lc']`: specifies the additive parameters in the form of a `np.array` of size `LxC`, where `C` is the alphabet size.
- `alphabet=theta_dict['alphabet']`: provides a list of characters corresponding to the columns of `values`
- `seq=model.x_stats['consensus_seq']`: causes `mavenn.heatmap()` to highlight the characters of a specific sequence of interest. In our case this is the consensus sequence, the additive parameters for which are all fixed to zero.
- `seq_kwargs={'c':'gray', 's':25}`: provides a keyword dictionary to pass to `ax.scatter()`; this specifies how the characters of the sequence of interest are to be graphically indicated.
- `cmap='coolwarm'`: specifies the colormap used to represent the values of the additive parameters.
- `cbar=True`: specifies that a colorbar be drawn.
- `cmap_size='2%'`: specifies the width of the colorbar relative to the enclosing ax object.
- `cmap_pad=.3`: specifies the spacing between the heatmap and the colorbar.
- `ccenter=0`: centers the colormap at zero.

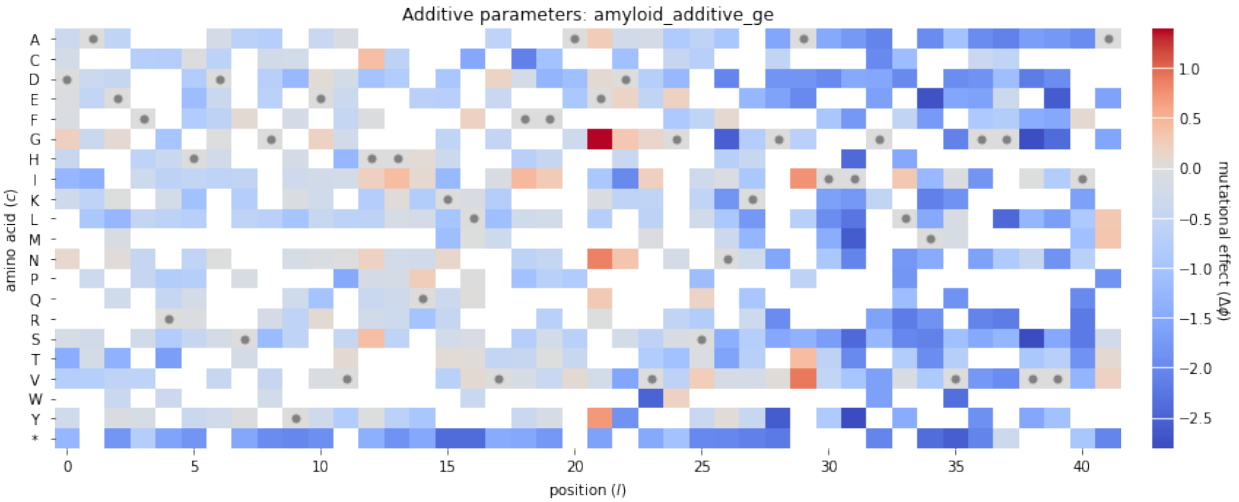
This function returns two objects: - `heatmap_ax` is the axes object on which the heatmap is drawn. - `cb` is the colorbar object; it's corresponding axes is given by `cb.ax`.

[14]:
```
# Create figure
fig, ax = plt.subplots(1,1, figsize=(12,5))

# Draw heatmap
heatmap_ax, cb = mavenn.heatmap(ax=ax,
                                values=theta_dict['theta_lc'],
                                alphabet=theta_dict['alphabet'],
                                seq=model.x_stats['consensus_seq'],
                                seq_kwargs={'c':'gray', 's':25},
                                cmap='coolwarm',
                                cbar=True,
                                cmap_size='2%',
                                cmap_pad=.3,
                                ccenter=0)
# Style heatmap (can be different between two dataset)
#heatmap_ax.set_xticks()
heatmap_ax.tick_params(axis='y', which='major', pad=10)
heatmap_ax.set_xlabel('position ($l$)')
heatmap_ax.set_ylabel('amino acid ($c$)')
heatmap_ax.set_title(f'Additive parameters: {model_name}')

# Style colorbar
cb.outline.set_visible(False)
```

(continues on next page)

(continued from previous page)

```
cb.ax.tick_params(direction='in', size=20, color='white')
cb.set_label('mutational effect ($\Delta \phi$)', labelpad=5, rotation=-90, ha='center',␣
↪va='center')

# Adjust figure and show
fig.tight_layout(w_pad=5)
```



Note that many of the squares in the heatmap are white. These correspond to additive parameters whose values are `NaN`. MAVE-NN sets the values of a feature effect to `NaN` when no variant in the training set exhibits that feature. Such `NaN` parameters are common as DMS libraries often do not contain a comprehensive set of single-amino-acid mutations.

2.2.3 References

1. Bolognesi B, Faure AJ, Seuma M, Schmiedel JM, Tartaglia GG, Lehner B. The mutational landscape of a prion-like domain. Nat Commun 10:4162 (2019).
2. Olson CA, Wu NC, Sun R. A comprehensive biophysical description of pairwise epistasis throughout an entire protein domain. Curr Biol 24:2643–2651 (2014).
3. Seuma M, Faure A, Badia M, Lehner B, Bolognesi B. The genetic landscape for amyloid beta fibril nucleation accurately discriminates familial Alzheimer's disease mutations. eLife 10:e63364 (2021).
4. Tareen A, Kooskhbaghi M, Posfai A, Ireland WT, McCandlish DM, Kinney JB. MAVE-NN: learning genotype-phenotype maps from multiplex assays of variant effect. bioRxiv doi:10.1101/2020.07.14.201475 (2020).

2.3 Tutorial 3: Splicing MPRA modeling using multiple built-in G-P maps

[1]:
```
# Standard imports
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
# Import MAVE-NN
import mavenn

# Import Logomaker for visualization
import logomaker
```

In this tutorial we show how to train multiple models with different G-P maps on the same dataset. To this end we use the built-in `'mpsa'` dataset, which contains data from the splicing MPRA of Wong et al. (2018). Next we show how to to compare the performance of these models, as in Figs. 5a-5d of Tareen et al. (2020). Finally, we demonstrate how to visualize the parameters of the `'pairwise'` G-P map trained on these data; similar visualizations are shown in Figs. 5e and 5f of Tareen et al. (2020).

2.3.1 Training multiple models

The models that we train each have a GE measurement process and one of four different types of G-P map: additive, neighbor, pairwise, or blackbox. The trained models are similar (though not identical) to the following built-in models, which can be loaded with `mavenn.load_example_model()`:

- `'mpsa_additive_ge'`
- `'mpsa_neighbor_ge'`
- `'mpsa_pairwise_ge'`
- `'mpsa_blackbox_ge'`

First we load, split, and preview the built-in `'mpsa'` dataset. We also compute the length of sequences in this dataset.

[2]:
```
# Load amyloid dataset
data_df = mavenn.load_example_dataset('mpsa')

# Get and report sequence length
L = len(data_df.loc[0,'x'])
print(f'Sequence length: {L:d} RNA nucleotides')

# Split dataset
trainval_df, test_df = mavenn.split_dataset(data_df)

# Preview trainval_df
print('\ntrainval_df:')
trainval_df
```

```
Sequence length: 9 RNA nucleotides
Training set   :   18,469 observations (  60.59%)
Validation set :    5,936 observations (  19.47%)
Test set       :    6,078 observations (  19.94%)
-------------------------------------------------
Total dataset  :   30,483 observations ( 100.00%)


trainval_df:
```

[2]:

	validation	tot_ct	ex_ct	y	x
0	False	28	2	0.023406	GGAGUGAUG
1	False	193	15	-0.074999	UUCGCGCCA

(continues on next page)

(continued from previous page)

```
2           False      27      0 -0.438475  UAAGCUUUU
3           False     130      2 -0.631467  AUGGUCGGG
4           False     552     19 -0.433012  AGGGCAGGA
...           ...     ...    ...       ...        ...
24400       False     167   1467  1.950100  GAGGUAAAU
24401       False     682     17 -0.570465  AUCGCUAGA
24402       False     190     17 -0.017078  CUGGUUGCA
24403       False     154     10 -0.140256  CGCGCACAA
24404       False     265      6 -0.571100  AUAGUCUAA

[24405 rows x 5 columns]
```

Next we instantiate and train our models. In order to train multiple different models in a consistent manner, we use dictionaries to specify default keyword arguments for `model.Model()` and `model.fit()`. Then for each type of G-P map we do the following: 1. Modify the hyperparameter dictionaries as desired. 2. Instantiate a model using these hyperparameters. 3. Set that model's training data. 4. Train the parameters of that model. 5. Evaluate that model's performance. 6. Save that model to disk.

[3]:
```
# Set default keyword arguments for model.Model()
default_model_kwargs = {
    'L':L,
    'alphabet':'rna',
    'regression_type':'GE',
    'ge_noise_model_type':'SkewedT',
    'ge_heteroskedasticity_order':2
}

# Set default keyword arguments for model.fit()
default_fit_kwargs = {
    'learning_rate':.001,
    'epochs':5,
    'batch_size':200,
    'early_stopping':True,
    'early_stopping_patience':30,
    'linear_initialization':False,
    'verbose':False
}

# Iterate over types of G-P maps
gpmap_types = ['additive','neighbor','pairwise','blackbox']
print(f'Training {len(gpmap_types)} models: {gpmap_types}')
for gpmap_type in gpmap_types:

    # Set model name
    model_name = f'mpsa_{gpmap_type}_ge'
    print('------------------------------')
    print(f"Training '{model_name}' model...\n")

    # Copy keyword arguments
    model_kwargs = default_model_kwargs.copy()
    fit_kwargs = default_fit_kwargs.copy()
```

(continues on next page)

(continued from previous page)

```
    # Modify keyword arguments based on G-P map being trained
    # Note: the need for different hyperparameters, such as batch_size
    # and learning_rate, was found by trial and error.
    if gpmap_type=='additive': pass;
    elif gpmap_type=='neighbor': pass;
    elif gpmap_type=='pairwise':
        fit_kwargs['batch_size'] = 50
    elif gpmap_type=='blackbox':
        model_kwargs['gpmap_kwargs'] = {'hidden_layer_sizes':[10]*5,
                                        'features':'pairwise'}
        fit_kwargs['learning_rate'] = 0.0005
        fit_kwargs['batch_size'] = 50
        fit_kwargs['early_stopping_patience'] = 10

    # Instantiate model using the keyword arguments in model_kwargs dict
    model = mavenn.Model(gpmap_type=gpmap_type, **model_kwargs)

    # Set training data
    model.set_data(x=trainval_df['x'],
                   y=trainval_df['y'],
                   validation_flags=trainval_df['validation'])

    # Train model using the keyword arguments in fig_kwargs dict
    model.fit(**fit_kwargs)

    # Compute variational information on test data
    I_var, dI_var =  model.I_variational(x=test_df['x'], y=test_df['y'])
    print(f'test_I_var: {I_var:.3f} +- {dI_var:.3f} bits')

    # Compute predictive information on test data
    I_pred, dI_pred = model.I_predictive(x=test_df['x'], y=test_df['y'])
    print(f'test_I_pred: {I_pred:.3f} +- {dI_pred:.3f} bits')

    # Save model to file
    model.save(model_name)

print('Done!')
```

```
Training 4 models: ['additive', 'neighbor', 'pairwise', 'blackbox']
----------------------------
Training 'mpsa_additive_ge' model...

N = 24,405 observations set as training data.
Using 24.3% for validation.
Data shuffled.
Time to set data: 0.223 sec.
```

```
0epoch [00:00, ?epoch/s]
```

```
Training time: 1.2 seconds
test_I_var: -0.140 +- 0.028 bits
test_I_pred: 0.059 +- 0.010 bits
Model saved to these files:
```

(continues on next page)

(continued from previous page)

```
        mpsa_additive_ge.pickle
        mpsa_additive_ge.h5
------------------------------
Training 'mpsa_neighbor_ge' model...

N = 24,405 observations set as training data.
Using 24.3% for validation.
Data shuffled.
Time to set data: 0.217 sec.
```

```
0epoch [00:00, ?epoch/s]
```

```
Training time: 1.3 seconds
test_I_var: -0.101 +- 0.024 bits
test_I_pred: 0.164 +- 0.009 bits
Model saved to these files:
        mpsa_neighbor_ge.pickle
        mpsa_neighbor_ge.h5
------------------------------
Training 'mpsa_pairwise_ge' model...

N = 24,405 observations set as training data.
Using 24.3% for validation.
Data shuffled.
Time to set data: 0.215 sec.
```

```
0epoch [00:00, ?epoch/s]
```

```
Training time: 2.1 seconds
test_I_var: 0.188 +- 0.025 bits
test_I_pred: 0.324 +- 0.013 bits
Model saved to these files:
        mpsa_pairwise_ge.pickle
        mpsa_pairwise_ge.h5
------------------------------
Training 'mpsa_blackbox_ge' model...

WARNING:tensorflow:From /Users/jkinney/miniforge3_arm64/lib/python3.9/site-packages/
↪tensorflow/python/ops/array_ops.py:5043: calling gather (from tensorflow.python.ops.
↪array_ops) with validate_indices is deprecated and will be removed in a future version.
Instructions for updating:
The `validate_indices` argument has no effect. Indices are always validated on CPU and␣
↪never validated on GPU.
N = 24,405 observations set as training data.
Using 24.3% for validation.
Data shuffled.
Time to set data: 0.216 sec.
```

```
0epoch [00:00, ?epoch/s]
```

```
Training time: 3.0 seconds
test_I_var: 0.320 +- 0.026 bits
test_I_pred: 0.403 +- 0.015 bits
Model saved to these files:
        mpsa_blackbox_ge.pickle
```

(continues on next page)

(continued from previous page)

```
        mpsa_blackbox_ge.h5
Done!
```

2.3.2 Visualizing model performance

To compare these models side-by-side, we first load them into a dictionary.

[4]:
```
# Iterate over types of G-P maps
gpmap_types = ['additive','neighbor','pairwise','blackbox']

# Create list of model names
model_names = [f'mpsa_{gpmap_type}_ge' for gpmap_type in gpmap_types]

# Load models into a dictionary indexed by model name
model_dict = {name:mavenn.load(name) for name in model_names}
```

```
Model loaded from these files:
        mpsa_additive_ge.pickle
        mpsa_additive_ge.h5
Model loaded from these files:
        mpsa_neighbor_ge.pickle
        mpsa_neighbor_ge.h5
Model loaded from these files:
        mpsa_pairwise_ge.pickle
        mpsa_pairwise_ge.h5
Model loaded from these files:
        mpsa_blackbox_ge.pickle
        mpsa_blackbox_ge.h5
```

To compare the performance of these models, we plot variational and predictive information in the form of a bar chart similar to that shown in Fig. 5a of Tareen et al., (2021).

[5]:
```
# Fill out dataframe containing values to plot
# This dataframe will then be used by seaborn's barplot() function
info_df = pd.DataFrame(columns=['name', 'gpmap', 'metric', 'I', 'dI'])
for gpmap_type in gpmap_types:

    # Get model
    name = f'mpsa_{gpmap_type}_ge'
    model = model_dict[name]

    # Compute variational information on test data
    I_var, dI_var =  model.I_variational(x=test_df['x'], y=test_df['y'])
    row = {'name':name,
           'gpmap':gpmap_type,
           'metric':'I_var',
           'I':I_var,
           'dI':dI_var}
    info_df = info_df.append(row, ignore_index=True)

    # Compute predictive information on test data
```

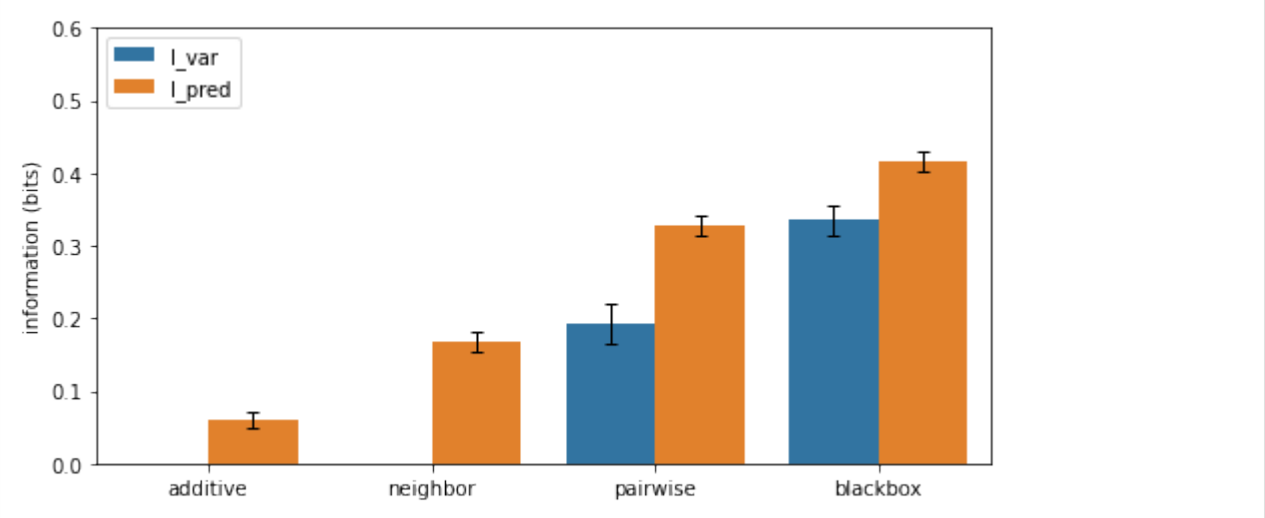
(continues on next page)

(continued from previous page)

```
    I_pred, dI_pred = model.I_predictive(x=test_df['x'], y=test_df['y'])
    row = {'name':name,
           'gpmap':gpmap_type,
           'metric':'I_pred',
           'I':I_pred,
           'dI':dI_pred}
    info_df = info_df.append(row, ignore_index=True)

# Print dataframe
print('Contents of info_df:', info_df, sep='\n')

# Create figure
fig, ax = plt.subplots(figsize=[8, 4])

# Plot bars
sns.barplot(ax=ax,
            data=info_df,
            hue='metric',
            x='gpmap',
            y='I')

# Plot errorbars
x = np.array([[x-.2,x+.2] for x in range(4)]).ravel()
ax.errorbar(x=x,
            y=info_df['I'].values,
            yerr=info_df['dI'].values,
            color='k', capsize=3, linestyle='none',
            elinewidth=1, capthick=1, solid_capstyle='round')

ax.set_ylabel('information (bits)')
ax.set_xlabel('')
ax.set_xlim([-.5, 3.5])
ax.set_ylim([0, 0.6])
ax.legend(loc='upper left')
```

```
Contents of info_df:
               name     gpmap  metric         I        dI
0  mpsa_additive_ge  additive   I_var -0.152251  0.024977
1  mpsa_additive_ge  additive  I_pred  0.058887  0.011109
2  mpsa_neighbor_ge  neighbor   I_var -0.092292  0.020540
3  mpsa_neighbor_ge  neighbor  I_pred  0.168971  0.013866
4  mpsa_pairwise_ge  pairwise   I_var  0.192587  0.027062
5  mpsa_pairwise_ge  pairwise  I_pred  0.327380  0.012934
6  mpsa_blackbox_ge  blackbox   I_var  0.335930  0.020831
7  mpsa_blackbox_ge  blackbox  I_pred  0.415347  0.013392
```

[5]:
```
<matplotlib.legend.Legend at 0x14e37b7c0>
```



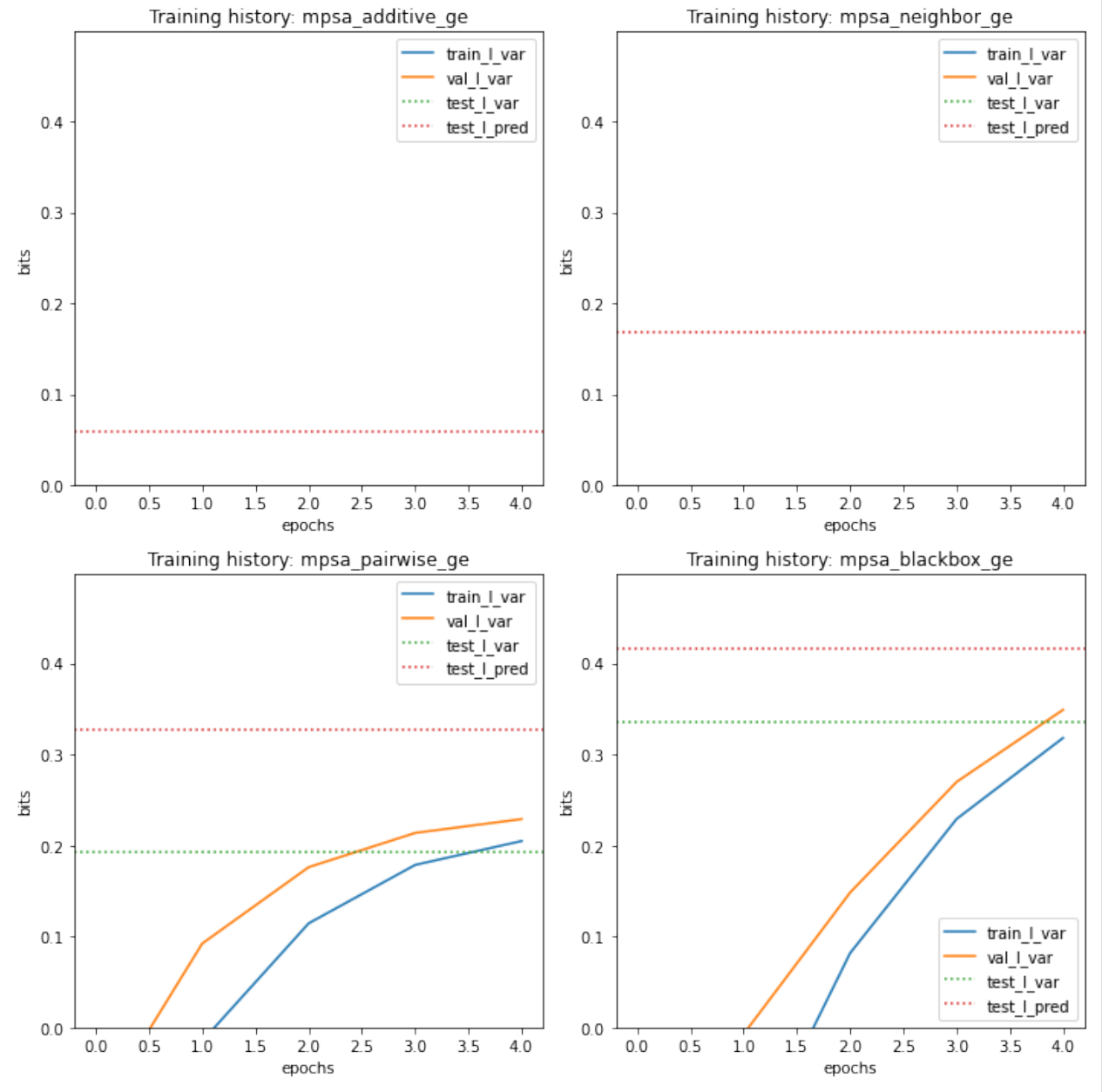
It can also be useful to observe the training history of each model in relation to performance metrics on the test set.

[6]:
```
# Create figure and axes for plotting
fig, axs = plt.subplots(2,2,figsize=[10,10])
axs = axs.ravel()

# Loop over models
for ax, name in zip(axs, model_names):

    # Get model
    model = model_dict[name]

    # Plot I_var_train, the variational information on training data as a function of␣
↪epoch
    ax.plot(model.history['I_var'],
            label=r'train_I_var')

    # Plot I_var_val, the variational information on validation data as a function of␣
↪epoch
    ax.plot(model.history['val_I_var'],
            label=r'val_I_var')

    # Get part of info_df referring to this model and index by metric
    ix = (info_df['name']==name)
    sub_df = info_df[ix].set_index('metric')

    # Show I_var_test, the variational information of the final model on test data
    ax.axhline(sub_df.loc['I_var','I'], color='C2', linestyle=':',
               label=r'test_I_var')

    # Show I_pred_test, the predictive information of the final model on test data
    ax.axhline(sub_df.loc['I_pred','I'], color='C3', linestyle=':',
               label=r'test_I_pred')

    # Style plot
    ax.set_xlabel('epochs')
```

(continues on next page)

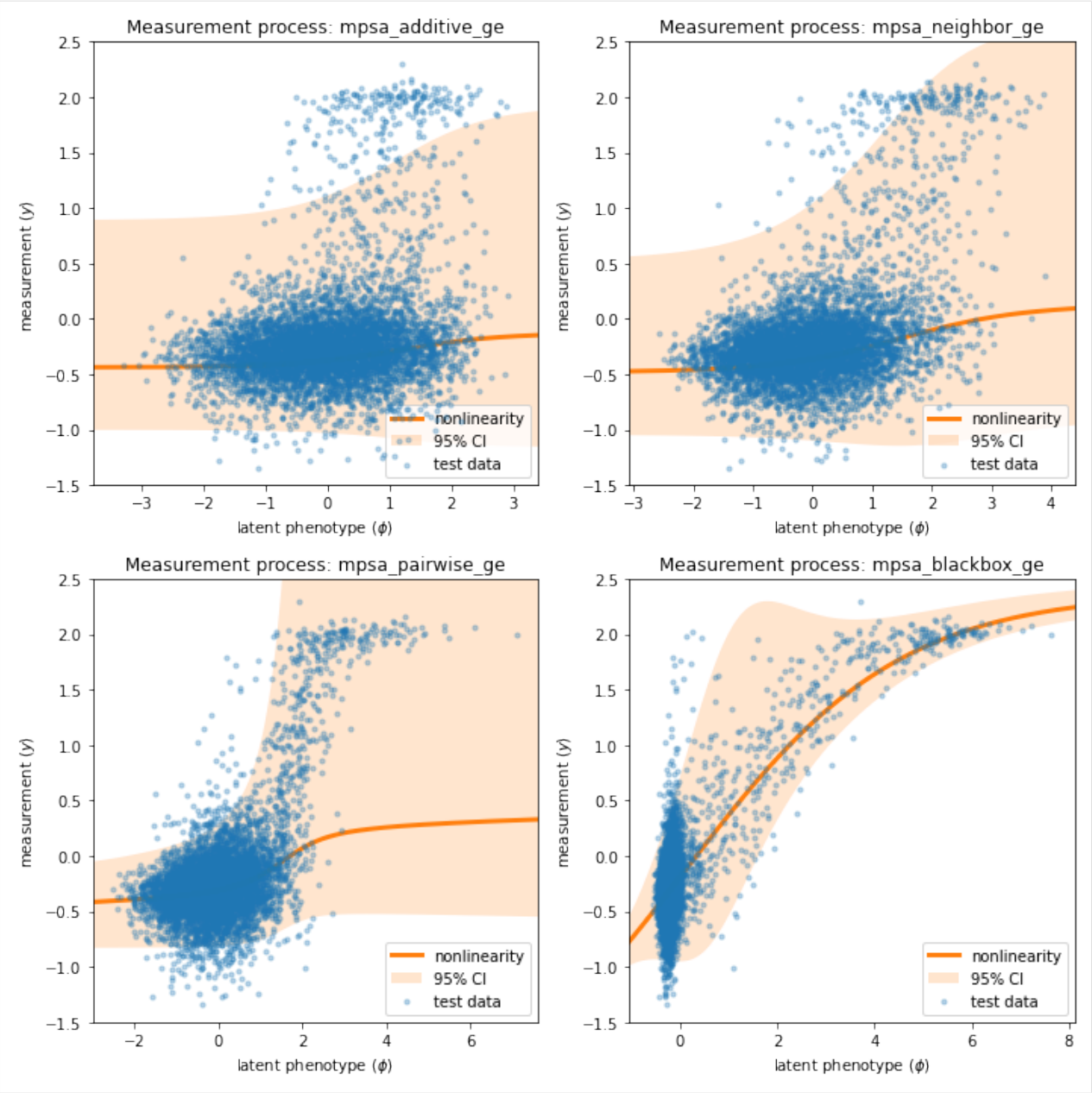
(continued from previous page)

```
    ax.set_ylabel('bits')
    ax.set_title(f'Training history: {name}')
    ax.set_ylim([0, 1.2*I_pred])
    ax.legend()

# Clean up figure
fig.tight_layout()
```



Next we visualize the GE measurement process inferred as part of our each latent phenotype model, comparing it to the test data.

[7]:
```
# Create figure and axes for plotting
fig, axs = plt.subplots(2,2,figsize=[10,10])
axs = axs.ravel()

# Loop over models
for ax, name in zip(axs, model_names):

    # Get model
    model = model_dict[name]

    # Get test data y values
    y_test = test_df['y']

    # Compute phi on test data
    phi_test = model.x_to_phi(test_df['x'])

    ## Set phi lims and create a grid in phi space
    phi_lim = [min(phi_test)-.5, max(phi_test)+.5]
    phi_grid = np.linspace(phi_lim[0], phi_lim[1], 1000)

    # Compute yhat for each phi gridpoint
    yhat_grid = model.phi_to_yhat(phi_grid)

    # Compute 95% CI for each yhat
    q = [0.025, 0.975]
    yqs_grid = model.yhat_to_yq(yhat_grid, q=q)

    # Plote 95% confidence interval
    ax.fill_between(phi_grid, yqs_grid[:, 0], yqs_grid[:, 1],
                    alpha=0.2, color='C1', lw=0, label='95% CI')

    # Plot GE nonlinearity
    ax.plot(phi_grid, yhat_grid,
            linewidth=3, color='C1', label='nonlinearity')

    # Plot scatter of phi and y values.
    ax.scatter(phi_test, y_test,
               color='C0', s=10, alpha=.3, label='test data', zorder=+100)

    # Style plot
    ax.set_xlim(phi_lim)
    ax.set_xlabel('latent phenotype ($\phi$)')
    ax.set_ylim([-1.5,2.5])
    ax.set_ylabel('measurement ($y$)')
    ax.set_title(f'Measurement process: {name}')
    ax.legend(loc='lower right')

fig.tight_layout()
```



Note that the measurement processes for all of the linear models (`'additive'`, `'neighbor'`, `'pairwise'`) are similar, while that of the `'blackbox'` model is quite different. This distinct behavior is due to the presence of nonlinearities in the `'blackbox'` G-P map that are not present in the other three.

2.3.3 Visualizaing pairwie model parameters

We now focus on visualizing the pairwise model. The mathematical formula for the pairwise G-P map is:

$$\phi_{\text{pairwise}}(x;\theta) = \theta_0 + \sum_l \sum_c \theta_{l:c} x_{l:c} + \sum_{l<l'} \sum_{c,c'} \theta_{l:c,l':c'} x_{l:c} x_{l':c'}.$$

To retrieve the values of each model's G-P map parameters, we use the method `model.get_theta()`, which returns a dictionary listing the various model parameter values. Because the sequence library spans nearly all possible 9nt 5' splice sites, rather than being clustered around a single wild-type sequence, we choose to insect the parameters using the "uniform" gauge.

[8]:
```
# Focus on pairwise model
model = model_dict['mpsa_pairwise_ge']

# Retrieve G-P map parameter dict and view dict keys
theta_dict = model.get_theta(gauge="uniform")
theta_dict.keys()
```

[8]:
```
dict_keys(['L', 'C', 'alphabet', 'theta_0', 'theta_lc', 'theta_lclc', 'theta_mlp',
↪'logomaker_df'])
```

Among the keys of the `dict` returned by `model.get_theta()`:

- `'theta_0'`: a single number representing the constant component, θ_0, of a linear model.
- `'theta_lc'`: an L x C matrix representing the additive parameters, $\theta_{l:c}$, of a linear model.
- `'logomaker_df'`: An L x C dataframe containing the additive parameters in a dataframe that facilitates visualization using `logomaker`.
- `'theta_lclc'`: an L x C x L x C tensor representing the pairwise parameters, $\theta_{l:c,l':c'}$, of a linear model; is nonzero only for neighbor and pairwise models.
- `'theta_mlp'`: a dictionary containing the parameters of the blackbox MLP model, if indeed this is the model that is fit.

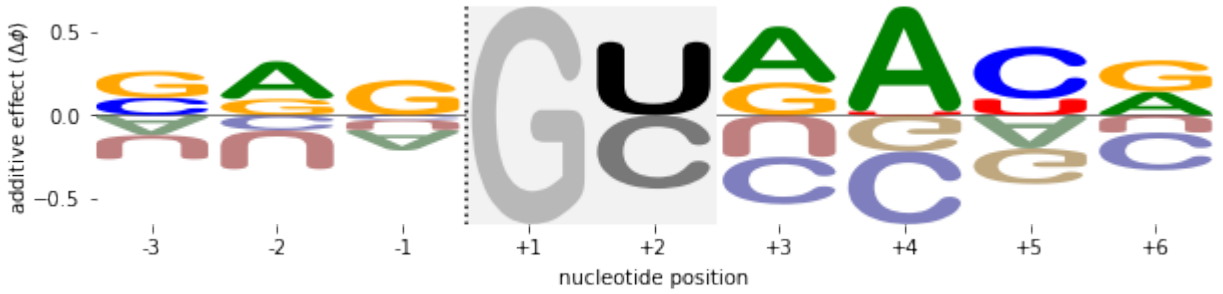
We first visualize the additive component of this model using a sequence logo, which we render using Logomaker (Tareen & Kinney, 2019). Note that the characters at positions +1 and +2 (corresponding to logo indices 3 and 4) are illustrated in a special manner due to library sequences having only a `'G'` at position +1 and a `'C'` or 'U' at position +2.

[9]:
```
# Get logo dataframe
logo_df = theta_dict['logomaker_df']

# Set NaN parameters to zero
logo_df.fillna(0, inplace=True)

# Create figure
fig, ax = plt.subplots(figsize=[10,2])

# Draw logo
logo = logomaker.Logo(df=logo_df,
                      ax=ax,
                      fade_below=.5,
                      shade_below=.5,
                      width=.9,
                      font_name='Arial Rounded MT Bold')
```

(continues on next page)

(continued from previous page)

```
ylim = ax.get_ylim()

# Highlight positions +1 and +2 to gray to indicate incomplete mutagenesis at these␣
↪positions
logo.highlight_position_range(pmin=3, pmax=4, color='w', alpha=1, zorder=10)
logo.highlight_position_range(pmin=3, pmax=4, color='gray', alpha=.1, zorder=11)

# Create a large `G` at position +1 to indicate that only this base was present in the␣
↪sequences under consideration
logo.style_single_glyph(p=3,
                        c='G',
                        flip=False,
                        floor=ylim[0],
                        ceiling=ylim[1],
                        color='gray',
                        zorder=30,
                        alpha=.5)

# Make 'C' and 'U' at position +2 black
logo.style_single_glyph(p=4, c='U', color='k', zorder=30, alpha=1)
logo.style_single_glyph(p=4, c='C', color='k', zorder=30, alpha=.5)

# Style logo
logo.style_spines(visible=False)
ax.axvline(2.5, linestyle=':', color='k', zorder=30)
ax.set_ylabel('additive effect ($\Delta \phi$)', labelpad=-1)
ax.set_xticks([0,1,2,3,4,5,6,7,8])
ax.set_xticklabels([f'{x:+d}' for x in range(-3,7) if x!=0])
ax.set_xlabel('nucleotide position', labelpad=5);
```



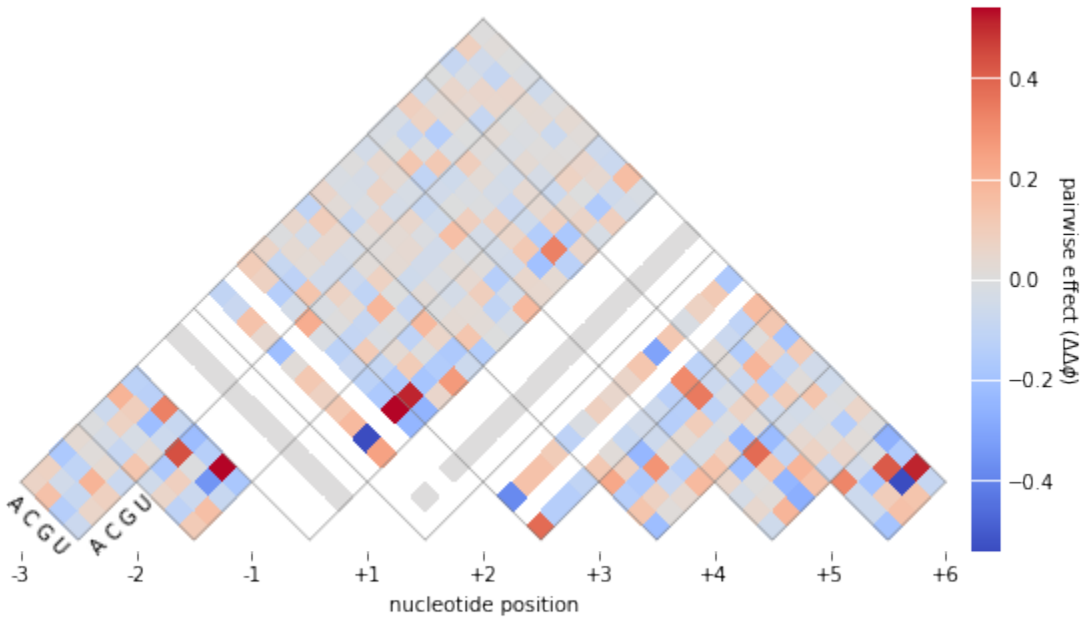
To visualize the pariwise parameters, we use the built-in function `mavenn.heatmap_pariwise()`.

[10]:
```
# Get pairwise parameters from theta_dict
theta_lclc = theta_dict['theta_lclc']

# Create fig and ax objects
fig, ax = plt.subplots(figsize=[10,5])

# Draw heatmap
ax, cb = mavenn.heatmap_pairwise(values=theta_lclc,
                                 alphabet='rna',
                                 ax=ax,
```

(continues on next page)

(continued from previous page)

```
                                    gpmap_type='pairwise',
                                    cmap_size='3%')

# Style heatmap
ax.set_xticks([0,1,2,3,4,5,6,7,8])
ax.set_xticklabels([f'{x:+d}' for x in range(-3,7) if x!=0])
ax.set_xlabel('nucleotide position', labelpad=5)

# Style colorbar
cb.set_label('pairwise effect ($\Delta \Delta \phi$)',
             labelpad=5, ha='center', va='center', rotation=-90)
cb.outline.set_visible(False)
cb.ax.tick_params(direction='in', size=20, color='white')
```



2.3.4 References

1. Wong MS, Kinney JB, Krainer AR. Quantitative activity profile and context dependence of all human 5' splice sites. Mol Cell 71:1012-1026.e3 (2018).
2. Tareen A, Kooshkbaghi M, Posfai A, Ireland WT, McCandlish DM, Kinney JB. MAVE-NN: learning genotype-phenotype maps from multiplex assays of variant effect. bioRxiv doi:10.1101/2020.07.14.201475 (2020).
3. Tareen A, Kinney JB. Logomaker: beautiful sequence logos in Python. Bioinformatics 36:2272–2274 (2019).

[ ]:

2.4 Tutorial 4: Protein DMS modeling using a biophysical G-P map

[1]:
```
# Standard imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Special imports
import mavenn
```

Here we show how to train and visualize a thermodynamic model describing the folding and IgG-binding of protein GB1 variants. This model was first proposed by Otwinowski (2018), who trained it on the DMS data of Olson et al. (2014). Here we repeat this exercise within the MAVE-NN framework, thus obtaining a model similar to the one featured in Figs. 6a and 6b of Tareen et al. (2021). The mathematical form of this G-P map is explianed in the supplemental material of Tareen et al. (2021); see in particular Fig. S4a.

2.4.1 Defining a custom G-P map

First we define a custom G-P map that represents our biophysical model. We do this by subclassing `CustomGPMapLayer` to get a custom G-P map class called `OtwinowskiGPMapLayer`. This subclassing procedure requires that we fill in the bodies of two specific methods. - `__init__()`: This constructor must first call the superclass constructor, which sets the attributes `L`, `C`, and `regularizer`. The the derived class constructor then defines all of the trainable parameters of the G-P map: `theta_f_0`, `theta_b_0`, `theta_f_lc`, and `theta_b_lc` in this case.

- `call()`: This is the meat of the custom G-P map. The input `x_lc` is a one-hot encoding of all sequences in a minimatch. It has size `[-1, L, C]`, where the first index runs over minibatch examples. The G-P map parameters are then used to compute and return a vector `phi` of latent phenotype values, one for each input sequence in the minibatch.

[2]:
```
# Standard TensorFlow imports
import tensorflow as tf
import tensorflow.keras.backend as K
from tensorflow.keras.initializers import Constant

# Import base class
from mavenn.src.layers.gpmap import GPMapLayer

# Define custom G-P map layer
class OtwinowskiGPMapLayer(GPMapLayer):
    """
    A G-P map representing the thermodynamic model described by
    Otwinowski (2018).
    """

    def __init__(self, *args, **kwargs):
        """Construct layer instance."""

        # Call superclass constructor
        # Sets self.L, self.C, and self.regularizer
        super().__init__(*args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
        # Initialize constant parameter for folding energy
        self.theta_f_0 = self.add_weight(name='theta_f_0',
                                         shape=(1,),
                                         trainable=True,
                                         regularizer=self.regularizer)

        # Initialize constant parameter for binding energy
        self.theta_b_0 = self.add_weight(name='theta_b_0',
                                         shape=(1,),
                                         trainable=True,
                                         regularizer=self.regularizer)

        # Initialize additive parameter for folding energy
        self.theta_f_lc = self.add_weight(name='theta_f_lc',
                                          shape=(1, self.L, self.C),
                                          trainable=True,
                                          regularizer=self.regularizer)

        # Initialize additive parameter for binding energy
        self.theta_b_lc = self.add_weight(name='theta_b_lc',
                                          shape=(1, self.L, self.C),
                                          trainable=True,
                                          regularizer=self.regularizer)

    def call(self, x_lc):
        """Compute phi given x."""

        # 1kT = 0.582 kcal/mol at room temperature
        kT = 0.582

        # Reshape input to samples x length x characters
        x_lc = tf.reshape(x_lc, [-1, self.L, self.C])

        # Compute Delta G for binding
        Delta_G_b = self.theta_b_0 + \
                    tf.reshape(K.sum(self.theta_b_lc * x_lc, axis=[1, 2]),
                               shape=[-1, 1])

        # Compute Delta G for folding
        Delta_G_f = self.theta_f_0 + \
                    tf.reshape(K.sum(self.theta_f_lc * x_lc, axis=[1, 2]),
                               shape=[-1, 1])

        # Compute and return fraction folded and bound
        Z = 1+K.exp(-Delta_G_f/kT)+K.exp(-(Delta_G_f+Delta_G_b)/kT)
        p_bf = (K.exp(-(Delta_G_f+Delta_G_b)/kT))/Z
        phi = p_bf #K.log(p_bf)/np.log(2)
        return phi
```

2.4.2 Training a model with a custom G-P map

Next we load the `'gb1'` dataset, compute sequence length, and split the data into a test set and a training+validation set.

[3]:
```
# Choose dataset
data_name = 'gb1'
print(f"Loading dataset '{data_name}' ")

# Load datset
data_df = mavenn.load_example_dataset(data_name)

# Get and report sequence length
L = len(data_df.loc[0,'x'])
print(f'Sequence length: {L:d} amino acids')

# Split dataset
trainval_df, test_df = mavenn.split_dataset(data_df)

# Preview trainval_df
print('trainval_df:')
trainval_df
```

```
Loading dataset 'gb1'
Sequence length: 55 amino acids
Training set   :  477,854 observations (  90.04%)
Validation set :   26,519 observations (   5.00%)
Test set       :   26,364 observations (   4.97%)
-------------------------------------------------
Total dataset  :  530,737 observations ( 100.00%)

trainval_df:
```

[3]:
```
        validation  dist  input_ct  selected_ct         y  \
0            False     2       173           33 -3.145154
1            False     2        18            8 -1.867676
2            False     2        66            2 -5.270800
3            False     2        72            1 -5.979498
4            False     2        69          168  0.481923
...            ...   ...       ...          ...       ...
504368       False     2       462          139 -2.515259
504369       False     2       317           84 -2.693165
504370       False     2       335           77 -2.896589
504371       False     2       148           28 -3.150861
504372       False     2        95           16 -3.287173

                                                        x
0       AAKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
1       ACKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
2       ADKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
3       AEKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
4       AFKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
...                                                   ...
504368  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
```

(continues on next page)

(continued from previous page)

```
504369  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
504370  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
504371  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
504372  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...

[504373 rows x 6 columns]
```

Next we create an instance of the `mavenn.Model` class. In addition to standard keyword arguments for GE regression, we pass keyword arguments specific to the use of our custom G-P map:

- `gpmap_type='custom'`: Alerts the `mavenn.Model()` constructor that we wish to use a custom G-P map.
- `custom_gpmap=OtwinowskiGPMapLayer`: Specifies the specific class to use for the custom G-P map layer.
- `gpmap_kwargs=gpmap_kwargs`: Provides a dictionary of arguments to be passed to the constructor of the custom G-P map.

[4]:
```
# Order the alphabet to match Otwinowski (2018)
alphabet = np.array(list('KRHEDNQTSCGAVLIMPYFW'))
C = len(alphabet)

# define custom gp_map parameters dictionary
gpmap_kwargs = {'L':L,
                'C':C,
                'theta_regularization': 0.0005}

# Create model instance
model = mavenn.Model(L=L,
                     alphabet=alphabet,
                     regression_type='GE',
                     ge_nonlinearity_type='nonlinear',
                     ge_nonlinearity_monotonic=False,
                     ge_noise_model_type='SkewedT',
                     ge_heteroskedasticity_order=2,
                     ge_nonlinearity_hidden_nodes=100,
                     eta_regularization=0.0001,
                     gpmap_type='custom',
                     normalize_phi=False,
                     custom_gpmap=OtwinowskiGPMapLayer,
                     gpmap_kwargs=gpmap_kwargs)
```

As in previous tutorials, we then set the training data using `model.set_data()` and then train the model using `model.fit()`.

[5]:
```
# Set False->True to train model
if False:

    # Set training data
    model.set_data(x=trainval_df['x'],
                   y=trainval_df['y'],
                   validation_flags=trainval_df['validation'])

    # Train model
    model.fit(learning_rate=.0005,
```

(continues on next page)

(continued from previous page)

```
            epochs=1000,
            batch_size=300,
            early_stopping=True,
            early_stopping_patience=50,
            linear_initialization=False,
            verbose=False);

    # Save model to file
    model_name = f'{data_name}_thermodynamic_model'
    model.save(model_name)
```

Next we evaluate the performance of the model on test data and save the model to disk.

2.4.3 Visualizing models with custom G-P maps

One can load the custom G-P map model and analyze its training history / performance in the same way as with built-in G-P map, e.g.:

[6]:
```
# Load model from file
model_name = f'{data_name}_thermodynamic_model'
model = mavenn.load(model_name)

# Compute variational information on test data
I_var, dI_var =  model.I_variational(x=test_df['x'], y=test_df['y'])
print(f'test_I_var: {I_var:.3f} +- {dI_var:.3f} bits')

# Compute predictive information on test data
I_pred, dI_pred = model.I_predictive(x=test_df['x'], y=test_df['y'])
print(f'test_I_pred: {I_pred:.3f} +- {dI_pred:.3f} bits')
```

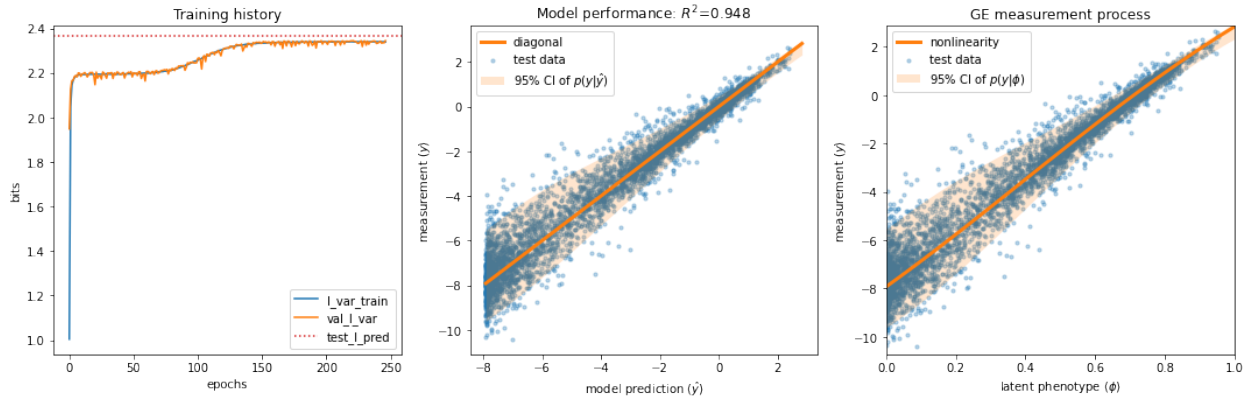
```
Model loaded from these files:
        gb1_thermodynamic_model.pickle
        gb1_thermodynamic_model.h5
test_I_var: 2.316 +- 0.013 bits
test_I_pred: 2.366 +- 0.006 bits
```

[7]:
```
# Get quantities to plot
y_test = test_df['y']
N_test = len(y_test)
yhat_test = model.x_to_yhat(test_df['x'])
phi_test = model.x_to_phi(test_df['x'])
phi_lim = [0, 1]
phi_grid = np.linspace(phi_lim[0], phi_lim[1], 1000)
yhat_grid = model.phi_to_yhat(phi_grid)
q = [0.025, 0.975]
yqs_grid = model.yhat_to_yq(yhat_grid, q=q)
ix = np.random.choice(a=N_test, size=5000, replace=False)
Rsq = np.corrcoef(yhat_test.ravel(), test_df['y'])[0, 1]**2

# Create figure and axes for plotting
```

(continues on next page)

(continued from previous page)

```
fig, axs = plt.subplots(1,3,figsize=[15,5])

# Plot panel 1: Training history
ax = axs[0]
ax.plot(model.history['I_var'],
        label=r'I_var_train')
ax.plot(model.history['val_I_var'],
        label=r'val_I_var')
ax.axhline(I_pred, color='C3', linestyle=':',
           label=r'test_I_pred')
ax.set_xlabel('epochs')
ax.set_ylabel('bits')
ax.set_title('Training history')
ax.legend()

## Panel 2: R^2 model performance
ax = axs[1]
ax.scatter(yhat_test[ix], y_test[ix], color='C0', s=10, alpha=.3,
           label='test data')
#xlim = [min(yhat_test), max(yhat_test)]
#ax.plot(xlim, xlim, '--', color='k', label='diagonal', zorder=100)
ax.fill_between(yhat_grid, yqs_grid[:, 0], yqs_grid[:, 1],
                alpha=0.2, color='C1', lw=0, label='95% CI of $p(y|\hat{y})$')
ax.plot(yhat_grid, yhat_grid,
        linewidth=3, color='C1', label='diagonal')
ax.set_xlabel('model prediction ($\hat{y}$)')
ax.set_ylabel('measurement ($y$)')
ax.set_title(f'Model performance: $R^2$={Rsq:.3}');
ax.legend()

## Panel 3: GE plot
ax = axs[2]
ax.scatter(phi_test[ix], y_test[ix],
           color='C0', s=10, alpha=.3, label='test data')
ax.fill_between(phi_grid, yqs_grid[:, 0], yqs_grid[:, 1],
                alpha=0.2, color='C1', lw=0, label='95% CI of $p(y|\phi)$')
ax.plot(phi_grid, yhat_grid,
        linewidth=3, color='C1', label='nonlinearity')
ax.set_ylim([min(y_test), max(y_test)])
ax.set_xlim(phi_lim)
ax.set_xlabel('latent phenotype ($\phi$)')
ax.set_ylabel('measurement ($y$)')
ax.set_title('GE measurement process')
ax.legend()
fig.tight_layout()
```



To retrieve the parameters of our custom G-P map, we again use the method `model.get_theta()`. This returns the dictionary provided by our custom G-P map via the method `get_params()`:

[8]:
```
# Retrieve G-P map parameter dict and view dict keys
theta_dict = model.layer_gpmap.get_params()
theta_dict.keys()
```

[8]:
```
dict_keys(['theta_f_0', 'theta_b_0', 'theta_f_lc', 'theta_b_lc'])
```

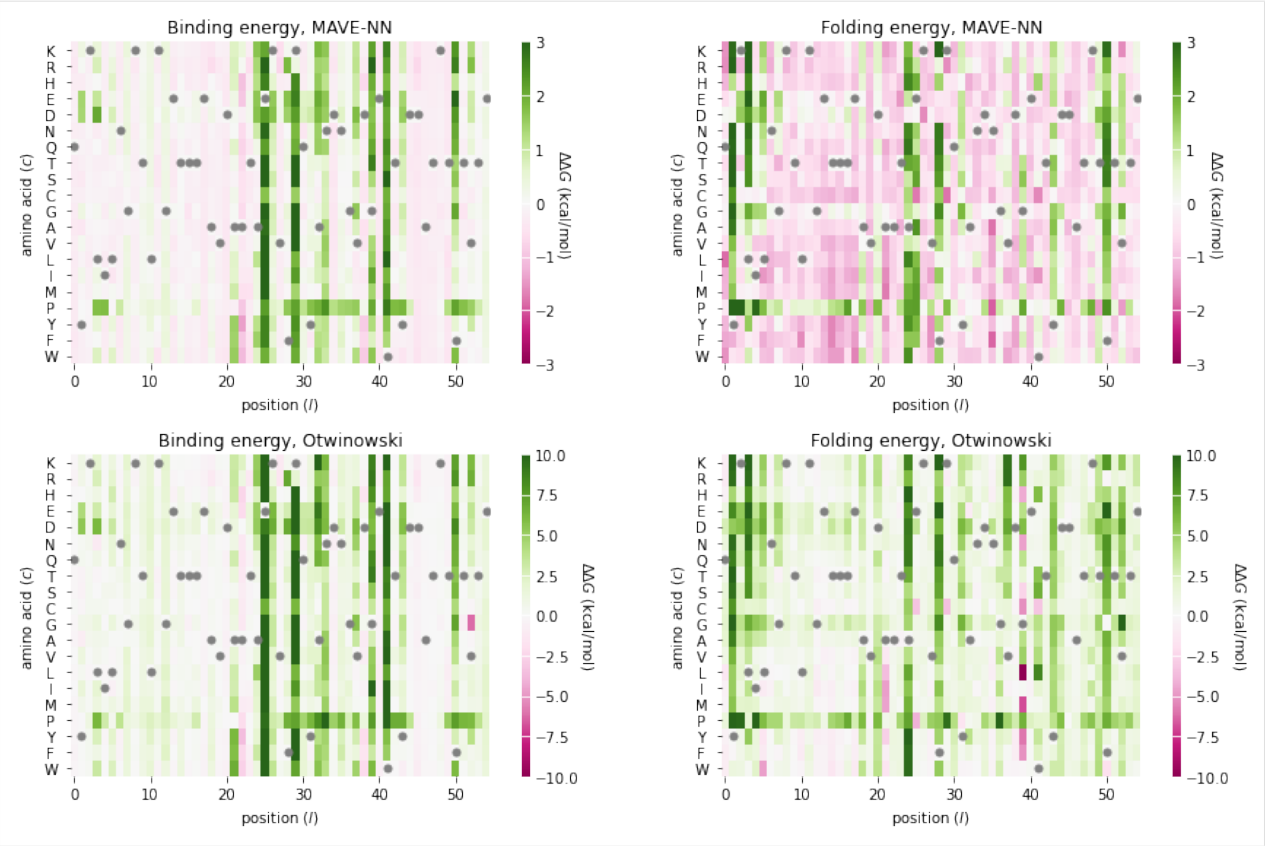
Next we visualize the additive parameters that determine both folding energy(`theta_b_lc`) and binding energy (`theta_r_lc`). Note that we visualize these as parameters as changes (`ddG_f` and `ddG_b`) with respect to the wild-type sequence. It is also worth comparing these $\Delta\Delta G$ values to those inferred by Otwinowski (2019).

[9]:
```
# Get the wild-type GB1 sequence
wt_seq = model.x_stats['consensus_seq']

# Convert this to a one-hot encoded matrix of size LxC
from mavenn.src.utils import _x_to_mat
x_lc_wt = _x_to_mat(wt_seq, model.alphabet)

# Subtract wild-type character value from parameters at each position
ddG_b_mat_mavenn = theta_dict['theta_b_lc'] - np.sum(x_lc_wt*theta_dict['theta_b_lc'],␣
↪axis=1)[:,np.newaxis]
ddG_f_mat_mavenn = theta_dict['theta_f_lc'] - np.sum(x_lc_wt*theta_dict['theta_f_lc'],␣
↪axis=1)[:,np.newaxis]

# Load Otwinowski parameters form file
dG_b_otwinowski_df = pd.read_csv('../../mavenn/examples/datasets/raw/otwinowski_gb_data.
↪csv.gz', index_col=[0]).T.reset_index(drop=True)[model.alphabet]
dG_f_otwinowski_df = pd.read_csv('../../mavenn/examples/datasets/raw/otwinowski_gf_data.
↪csv.gz', index_col=[0]).T.reset_index(drop=True)[model.alphabet]

# Compute ddG matrices for Otwinowski
ddG_b_mat_otwinowski = dG_b_otwinowski_df.values - \
                       np.sum(x_lc_wt*dG_b_otwinowski_df.values, axis=1)[:,np.newaxis]
ddG_f_mat_otwinowski = dG_f_otwinowski_df.values - \
                       np.sum(x_lc_wt*dG_f_otwinowski_df.values, axis=1)[:,np.newaxis]

# Set shared keyword arguments for heatmap
heatmap_kwargs = {
```

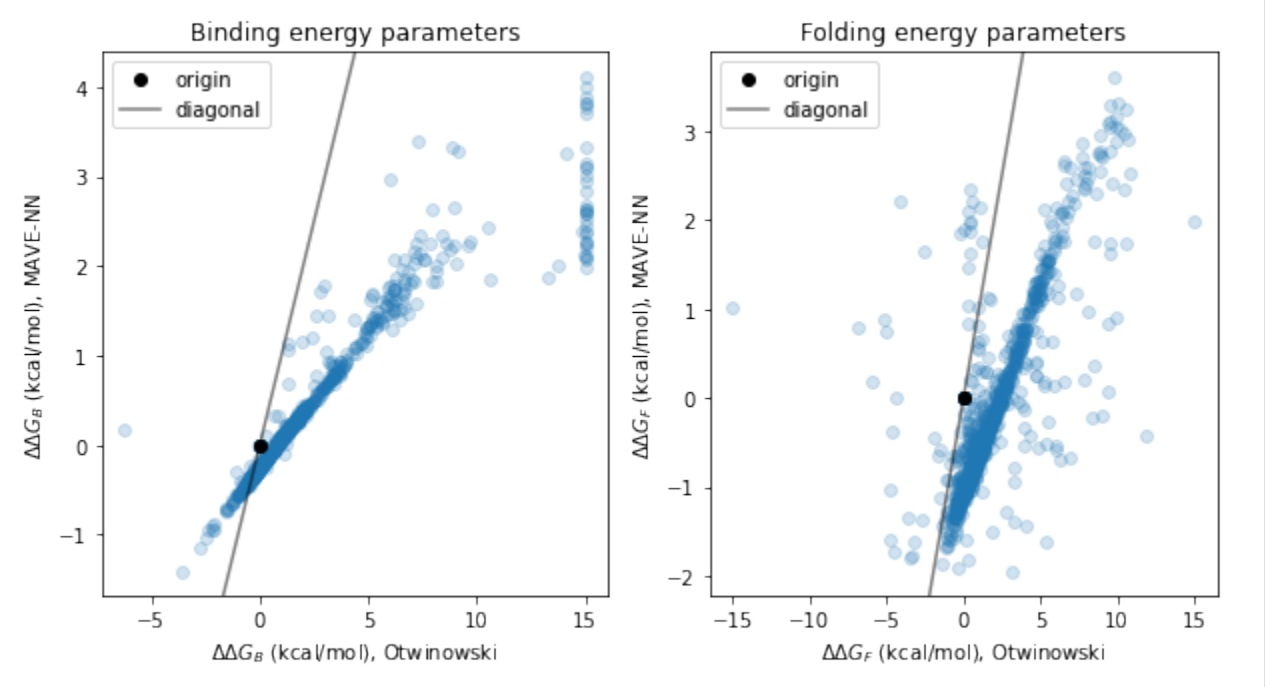
(continues on next page)

(continued from previous page)

```
    'alphabet':model.alphabet,
    'seq':wt_seq,
    'seq_kwargs':{'c':'gray', 's':25},
    'cmap':'PiYG',
    'cbar':True,
    'cmap_size':'2%',
    'cmap_pad':.3,
    'ccenter':0
}

# Set plotting routine
def draw(ax, ddG_mat, title, clim):
        # Draw binding energy heatmap
        heatmap_ax, cb = mavenn.heatmap(ax=ax,
                                        values=ddG_mat,
                                        clim=clim,
                                        **heatmap_kwargs)
        heatmap_ax.tick_params(axis='y', which='major', pad=10)
        heatmap_ax.set_xlabel('position ($l$)')
        heatmap_ax.set_ylabel('amino acid ($c$)')
        heatmap_ax.set_title(title)
        cb.outline.set_visible(False)
        cb.ax.tick_params(direction='in', size=20, color='white')
        cb.set_label('$\Delta \Delta G$ (kcal/mol)',
                     labelpad=5, rotation=-90, ha='center', va='center')

# Create figure and make plots
fig, axs = plt.subplots(2,2, figsize=(12,8))
draw(ax=axs[0,0],
     ddG_mat=ddG_b_mat_mavenn,
     title='Binding energy, MAVE-NN',
     clim=(-3, 3))
draw(ax=axs[0,1],
     ddG_mat=ddG_f_mat_mavenn,
     title='Folding energy, MAVE-NN',
     clim=(-3, 3))
draw(ax=axs[1,0],
     ddG_mat=ddG_b_mat_otwinowski,
     title='Binding energy, Otwinowski',
     clim=(-10, 10))
draw(ax=axs[1,1],
     ddG_mat=ddG_f_mat_otwinowski,
     title='Folding energy, Otwinowski',
     clim=(-10, 10))

# Adjust figure and show
fig.tight_layout(w_pad=5);
```



[10]:
```
# Set plotting routine
def draw(ax, x, y, ddG_var, title):
    ax.scatter(x, y, alpha=.2)
    xlim = ax.get_xlim()
    ax.autoscale(False)
    ax.plot(0,0,'ok', label='origin')
    ax.plot(xlim, xlim, '-k', alpha=.5, label='diagonal')
    ax.set_xlabel(f'{ddG_var} (kcal/mol), Otwinowski')
    ax.set_ylabel(f'{ddG_var} (kcal/mol), MAVE-NN')
    ax.set_title(title)
    ax.legend()

# Create figure and make plots
fig, axs = plt.subplots(1,2, figsize=(10,5))
draw(ax=axs[0],
     x=ddG_b_mat_otwinowski.ravel(),
     y=ddG_b_mat_mavenn.ravel(),
     ddG_var='$\Delta \Delta G_B$',
     title='Binding energy parameters')
draw(ax=axs[1],
     x=ddG_f_mat_otwinowski.ravel(),
     y=ddG_f_mat_mavenn.ravel(),
     ddG_var='$\Delta \Delta G_F$',
     title='Folding energy parameters')
```



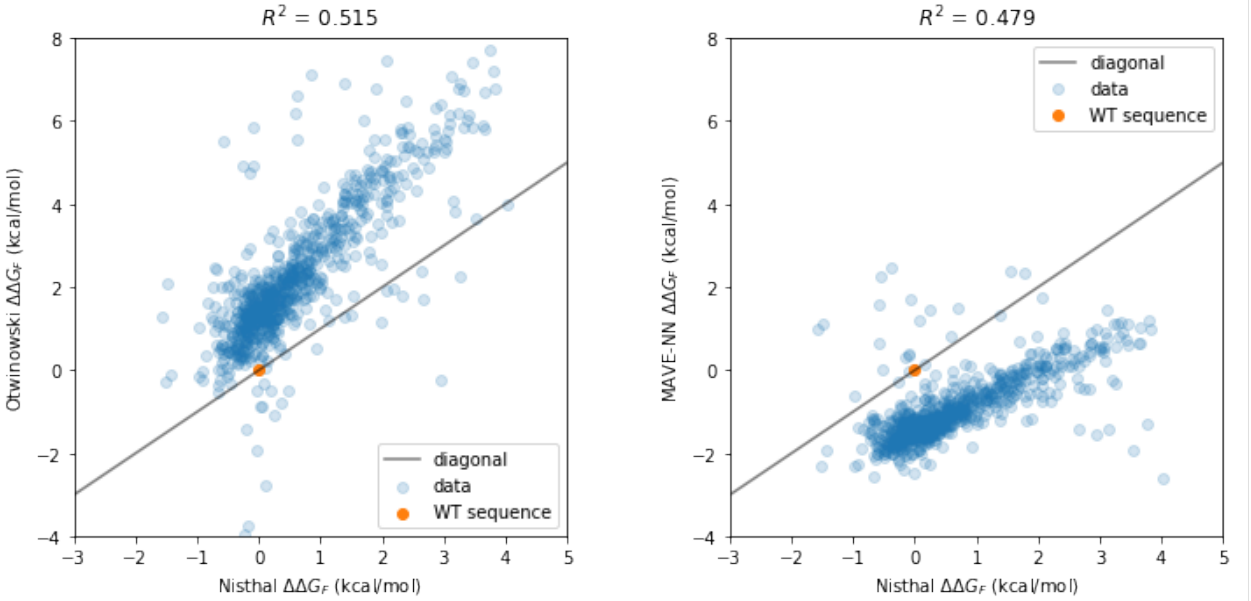
Finally, we compare our thermodynamic model's folding energy predictions to the $\Delta\Delta G_F$ measurements of Nisthal et al. (2019).

[11]:
```
# Load Nisthal data
nisthal_df = mavenn.load_example_dataset('nisthal')
nisthal_df.set_index('x', inplace=True)

# Get Nisthal folding energies relative to WT
dG_f_nisthal = nisthal_df['y']
dG_f_wt_nisthal = dG_f_nisthal[wt_seq]
ddG_f_nisthal = dG_f_nisthal - dG_f_wt_nisthal

# Get MAVE-NN folding energies relative to WT
x_nisthal = nisthal_df.index.values
x_nisthal_ohe = mavenn.src.utils.x_to_ohe(x=x_nisthal,
                                          alphabet=model.alphabet)
ddG_f_vec = ddG_f_mat_mavenn.ravel().reshape([1,-1])
ddG_f_mavenn = np.sum(ddG_f_vec*x_nisthal_ohe, axis=1)

# Get Otwinowski folding energies relative to WT
ddG_f_vec_otwinowski = ddG_f_mat_otwinowski.ravel().reshape([1,-1])
ddG_f_otwinowski = np.sum(ddG_f_vec_otwinowski*x_nisthal_ohe, axis=1)

# Define plotting routine
def draw(ax, y, model_name):
    Rsq = np.corrcoef(ddG_f_nisthal, y)[0, 1]**2
    ax.scatter(ddG_f_nisthal, y, alpha=.2, label='data')
    ax.scatter(0,0, label='WT sequence')
    xlim = [-3,5]
    ax.set_xlim(xlim)
    ax.set_ylim([-4,8])
```

(continues on next page)

(continued from previous page)

```
    ax.plot(xlim, xlim, color='k', alpha=.5, label='diagonal')
    ax.set_xlabel(f'Nisthal $\Delta \Delta G_F$ (kcal/mol)')
    ax.set_ylabel(f'{model_name} $\Delta \Delta G_F$ (kcal/mol)')
    ax.set_title(f'$R^2$ = {Rsq:.3f}')
    ax.legend()

# Make figure
fig, axs = plt.subplots(1,2,figsize=[10,5])
draw(ax=axs[0],
     y=ddG_f_otwinowski,
     model_name='Otwinowski')
draw(ax=axs[1],
     y=ddG_f_mavenn,
     model_name='MAVE-NN')

fig.tight_layout(w_pad=5)
```



2.4.4 References

1. Otwinowski J. Biophysical inference of epistasis and the effects of mutations on protein stability and function. Mol Biol Evol 35:2345–2354 (2018).
2. Olson CA, Wu NC, Sun R. A comprehensive biophysical description of pairwise epistasis throughout an entire protein domain. Curr Biol 24:2643–2651 (2014).
3. Tareen A, Posfai A, Ireland WT, McCandlish DM, Kinney JB. MAVE-NN: learning genotype-phenotype maps from multiplex assays of variant effect. bioRxiv doi:10.1101/2020.07.14.201475 (2020).
4. Nisthal A, Wang CY, Ary ML, Mayo SL. Protein stability engineering insights revealed by domain-wide comprehensive mutagenesis. Proc Natl Acad Sci 116:16367–16377 (2019).


```
[ ]:
```

2.5 Tutorial 5: The sort-seq *E. Coli lac* promoter binding analysis using a custom biophysical G-P maps

[1]:
```
# Standard imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Special imports
import mavenn
```

The sort-seq MPRA data of Kinney et al., 2010. The authors in Ref. [1] used fluorescence-activated cell sorting, followed by deep sequencing, to assay gene expression levels from variant lac promoters in E. coli. The data is available in MAVE-nn `load_example_dataset` function and it is called `'sortseq'`.

[2]:
```
# Choose dataset
data_name = 'sortseq'

print(f"Loading dataset '{data_name}' ")

# Load datset
data_df = mavenn.load_example_dataset(data_name)

# Get and report sequence length
L = len(data_df.loc[0, 'x'])
print(f'Sequence length: {L:d} amino acids')

# Preview dataset
data_df
```

```
Loading dataset 'sortseq'
Sequence length: 75 amino acids
```

[2]:

	set	ct_0	ct_1	ct_2	ct_3	ct_4	ct_5	ct_6	ct_7	ct_8	ct_9	\
0	training	0	1	0	0	0	0	0	0	0	0	
1	test	0	0	0	0	0	0	0	0	1	0	
2	test	0	0	0	0	0	0	1	0	0	0	
3	training	0	0	0	0	0	0	0	0	0	1	
4	training	0	0	0	0	0	0	0	0	0	1	
...	...	...	...	...	...	...	...	...	...	...	...	
50513	validation	0	0	0	1	0	0	0	0	0	0	
50514	test	0	0	0	0	0	0	0	0	1	0	
50515	training	0	0	0	1	0	0	0	0	0	0	
50516	training	1	0	0	0	0	0	0	0	0	0	
50517	training	1	0	0	0	0	0	0	0	0	0	

	x
0	AAAAAAAGTGAGTTAGCCAACTAATTAGGCACCGTACGCTTTATAG...
1	AAAAAATCTGAGTTAGCTTACTCATTAGGCACCCCAGGCTTGACAC...
2	AAAAAATCTGAGTTTGCTCACTCTATCGGCACCCCAGTCTTTACAC...
3	AAAAAATGAGAGTTAGTTCACTCATTCGGCACCACAGGCTTTACAA...
4	AAAAAATGGGTGTTAGCTCTATCATTAGGCACCCCCGGCTTTACAC...
...	...

(continues on next page)

(continued from previous page)

```
50513  TTTTGCAGAGTGTCAGCCCACTCATTACGCACCGCAGCCGTTACAC...
50514  TTTTTATGTGAGTTAGCTCACTCATTCGGCACCCTAGGCTTTACAC...
50515  TTTTTATGTGAGTTTGCTCACTCATGTGGCACCTAAGGCTTTACGC...
50516  TTTTTATGTGGGTTAGGTCGCGCATTAGGCACCGCAGGCTTTACCC...
50517  TTTTTATGTGTGTTTACTCTCTCATTAGGCACTCCACGCTTTACAC...

[50518 rows x 12 columns]
```

[3]:
```
# Split dataset
trainval_df, test_df = mavenn.split_dataset(data_df)

# Show dataset sizes
print(f'Train + val set size : {len(trainval_df):6,d} observations')
print(f'Test set size        : {len(test_df):6,d} observations')

# Preview trainval_df
trainval_df
```

```
Training set   :   30,516 observations (  60.41%)
Validation set :   10,067 observations (  19.93%)
Test set       :    9,935 observations (  19.67%)
-------------------------------------------------
Total dataset  :   50,518 observations ( 100.00%)

Train + val set size : 40,583 observations
Test set size        :  9,935 observations
```

[3]:
```
       validation  ct_0  ct_1  ct_2  ct_3  ct_4  ct_5  ct_6  ct_7  ct_8  ct_9  \
0           False     0     1     0     0     0     0     0     0     0     0
1           False     0     0     0     0     0     0     0     0     0     1
2           False     0     0     0     0     0     0     0     0     0     1
3           False     0     1     0     0     0     0     0     0     0     0
4            True     0     0     0     0     0     0     1     0     0     0
...           ...   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...
40578        True     0     1     0     0     0     0     0     0     0     0
40579        True     0     0     0     1     0     0     0     0     0     0
40580       False     0     0     0     1     0     0     0     0     0     0
40581       False     1     0     0     0     0     0     0     0     0     0
40582       False     1     0     0     0     0     0     0     0     0     0

                                                    x
0      AAAAAAAGTGAGTTAGCCAACTAATTAGGCACCGTACGCTTTATAG...
1      AAAAAATGAGAGTTAGTTCACTCATTCGGCACCACAGGCTTTACAA...
2      AAAAAATGGGTGTTAGCTCTATCATTAGGCACCCCCGGCTTTACAC...
3      AAAAAATGTCAGTTAGCTGACTCATTAGGCACCCCTGGCTTTACGT...
4      AAAAAATGTGAGAAAGCTCACTCCTTTGGCACCGCAGGCTTTACAC...
...                                                  ...
40578  TTTTGATGTGGGTTTGCTCTCTCTTCAGGCACCCCACGCTTTACGC...
40579  TTTTGCAGAGTGTCAGCCCACTCATTACGCACCGCAGCCGTTACAC...
40580  TTTTTATGTGAGTTTGCTCACTCATGTGGCACCTAAGGCTTTACGC...
40581  TTTTTATGTGGGTTAGGTCGCGCATTAGGCACCGCAGGCTTTACCC...
40582  TTTTTATGTGTGTTTACTCTCTCATTAGGCACTCCACGCTTTACAC...
```

(continues on next page)

(continued from previous page)

```
[40583 rows x 12 columns]
```

Training data are the count columns (10 columns) of the above dataset.

[4]:
```
# Get the length of the sequence
L = len(data_df['x'][0])
# Get the column index for the counts
y_cols = trainval_df.columns[1:-1]
# Get the number of count columns
len_y_cols = len(y_cols)
```

2.5.1 Training

A four-state thermodynamic model for transcriptional activation at *E. coli lac* promoter which proposed in Ref. [1] is trained here using MAVE-NN. Here, ΔG_R and ΔG_C are RNAP-DNA and CRP-DNA binding Gibbs free energies and CRP-RNAP interaction energy is represented by a scalar ΔG_I. The four-state thermodynamic model is summarized below:

microstates	Gibbs free energies	activity
free DNA	0	0
CRP-DNA binding	ΔG_C	0
RNAP-DNA binding	ΔG_R	1
CRP and RNAP both bounded to DNA and interact	$\Delta G_C + \Delta G_R + \Delta G_I$	1

The rate of transcription `tr_rate` has the following form:

$$tr_{\text{rate}} = t_{sat} \frac{\exp(-\Delta G_R) + \exp(-\Delta G_C - \Delta G_R - \Delta G_I)}{1 + \exp(-\Delta G_C) + \exp(-\Delta G_R) + \exp(-\Delta G_C - \Delta G_R - \Delta G_I)}$$

in which t_{sat} is the transcription rate resulting from full RNAP occupancy.

Here, the ΔG_C and ΔG_R are trainable matrices (weights of the neural network) and ΔG_I and t_{sat} are trainable scalars.

To fit the above thermodynamic models, we used the **Custom G-P maps** layer implemented in `'mavenn.src.layers.gpmap'`. For the detailed discussion on how to use the MAVE-NN custom G-P maps layer, checkout the thermodynamic model for IgG binding by GB1 tutorial.

[5]:
```
from mavenn.src.layers.gpmap import GPMapLayer

# Tensorflow imports
import tensorflow as tf
import tensorflow.keras.backend as K
from tensorflow.keras.initializers import Constant

class ThermodynamicLayer(GPMapLayer):
    """
    Represents a four-stage thermodynamic model
    containing the states:
    1. free DNA
    2. CPR-DNA binding
    3. RNAP-DNA binding
    4. CPR and RNAP both bounded to DNA and interact
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self,
                 tf_start,
                 tf_end,
                 rnap_start,
                 rnap_end,
                 *args, **kwargs):
        """Construct layer instance."""

        # Call superclass
        super().__init__(*args, **kwargs)

        # set attributes
        self.tf_start = tf_start            # transcription factor starting position
        self.tf_end = tf_end                 # transcription factor ending position
        self.L_tf = tf_end - tf_start        # length of transcription factor
        self.rnap_start = rnap_start         # RNAP starting position
        self.rnap_end = rnap_end             # RNAP ending position
        self.L_rnap = rnap_end - rnap_start  # length of RNAP

        # define bias/chemical potential weight for TF/CRP energy
        self.theta_tf_0 = self.add_weight(name='theta_tf_0',
                                          shape=(1,),
                                          initializer=Constant(1.),
                                          trainable=True,
                                          regularizer=self.regularizer)

        # define bias/chemical potential weight for rnap energy
        self.theta_rnap_0 = self.add_weight(name='theta_rnap_0',
                                            shape=(1,),
                                            initializer=Constant(1.),
                                            trainable=True,
                                            regularizer=self.regularizer)

        # initialize the theta_tf
        theta_tf_shape = (1, self.L_tf, self.C)
        theta_tf_init = np.random.randn(*theta_tf_shape)/np.sqrt(self.L_tf)

        # define the weights of the layer corresponds to theta_tf
        self.theta_tf = self.add_weight(name='theta_tf',
                                        shape=theta_tf_shape,
                                        initializer=Constant(theta_tf_init),
                                        trainable=True,
                                        regularizer=self.regularizer)

        # define theta_rnap parameters
        theta_rnap_shape = (1, self.L_rnap, self.C)
        theta_rnap_init = np.random.randn(*theta_rnap_shape)/np.sqrt(self.L_rnap)

        # define the weights of the layer corresponds to theta_rnap
        self.theta_rnap = self.add_weight(name='theta_rnap',
```

(continues on next page)

(continued from previous page)

```
                                        shape=theta_rnap_shape,
                                        initializer=Constant(theta_rnap_init),
                                        trainable=True,
                                        regularizer=self.regularizer)

        # define trainable real number G_I, representing interaction Gibbs energy
        self.theta_dG_I = self.add_weight(name='theta_dG_I',
                                  shape=(1,),
                                  initializer=Constant(-4),
                                  trainable=True,
                                  regularizer=self.regularizer)


    def call(self, x):
        """Process layer input and return output.

        x: (tensor)
            Input tensor that represents one-hot encoded
            sequence values.
        """

        # 1kT = 0.616 kcal/mol at body temperature
        kT = 0.616

        # extract locations of binding sites from entire lac-promoter sequence.
        # for transcription factor and rnap
        x_tf = x[:, self.C * self.tf_start:self.C * self.tf_end]
        x_rnap = x[:, self.C * self.rnap_start: self.C * self.rnap_end]

        # reshape according to tf and rnap lengths.
        x_tf = tf.reshape(x_tf, [-1, self.L_tf, self.C])
        x_rnap = tf.reshape(x_rnap, [-1, self.L_rnap, self.C])

        # compute delta G for crp binding
        G_C = self.theta_tf_0 + \
            tf.reshape(K.sum(self.theta_tf * x_tf, axis=[1, 2]),
                       shape=[-1, 1])

        # compute delta G for rnap binding
        G_R = self.theta_rnap_0 + \
            tf.reshape(K.sum(self.theta_rnap * x_rnap, axis=[1, 2]),
                       shape=[-1, 1])

        G_I = self.theta_dG_I

        # compute phi
        numerator_of_rate = K.exp(-G_R/kT) + K.exp(-(G_C+G_R+G_I)/kT)
        denom_of_rate = 1.0 + K.exp(-G_C/kT) + K.exp(-G_R/kT) + K.exp(-(G_C+G_R+G_I)/kT)
        phi = numerator_of_rate/denom_of_rate

        return phi
```

Training the Model

Here we train the model with MAVEN. Note that we initialize the Gibbs energies with random negative values to help the convergence of the training.

[6]:
```
# define custom gp_map parameters dictionary
gpmap_kwargs = {'tf_start': 1,  # starting position of the CRP
                'tf_end': 27,  # ending position of the CRP
                'rnap_start': 34,  # starting position of the RNAP
                'rnap_end': 75,   # ending position of the RNAP
                'L': L,
                'C': 4,
                'theta_regularization': 0.1}

# Create model
model = mavenn.Model(L=L,
                     Y=len_y_cols,
                     alphabet='dna',
                     regression_type='MPA',
                     gpmap_type='custom',
                     gpmap_kwargs=gpmap_kwargs,
                     custom_gpmap=ThermodynamicLayer);

# Set training data
model.set_data(x=trainval_df['x'],
               y=trainval_df[y_cols],
               validation_flags=trainval_df['validation'],
               shuffle=True);

# Fit model to data
model.fit(learning_rate=5e-4,
          epochs=2000,
          batch_size=100,
          early_stopping=True,
          early_stopping_patience=25,
          linear_initialization=False,
          verbose=False);
```

```
N = 40,583 observations set as training data.
Using 24.8% for validation.
Data shuffled.
Time to set data: 0.447 sec.
```

```
0epoch [00:00, ?epoch/s]
```

```
Training time: 71.6 seconds
```

[7]:
```
model.save('sortseq_thermodynamic_mpa')
```

```
Model saved to these files:
        sortseq_thermodynamic_mpa.pickle
        sortseq_thermodynamic_mpa.h5
```

[8]:
```
# Compute predictive information on test data
I_pred, dI_pred = model.I_predictive(x=test_df['x'], y=test_df[y_cols])
```

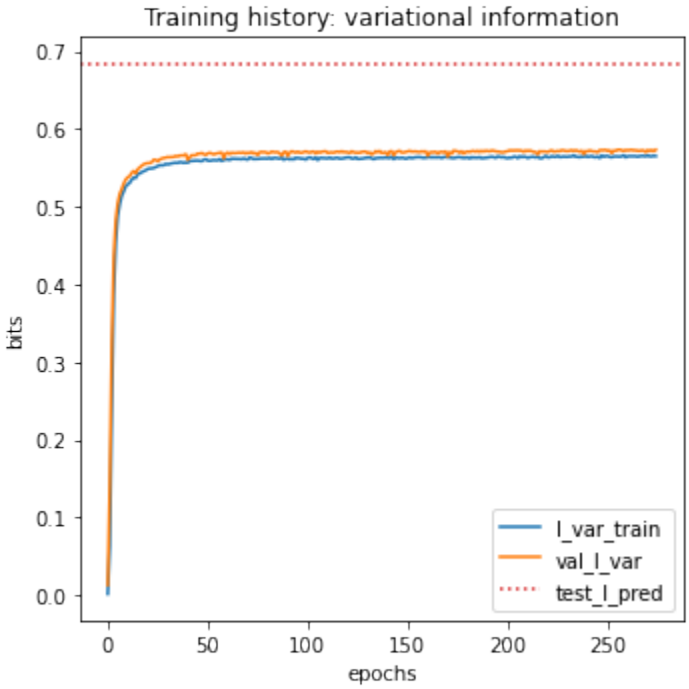
(continues on next page)

(continued from previous page)

```
print(f'test_I_pred: {I_pred:.3f} +- {dI_pred:.3f} bits')
```

```
test_I_pred: 0.685 +- 0.011 bits
```

[9]:
```
# Create figure and axes for plotting
fig, ax = plt.subplots(1, 1, figsize=[5, 5])

# Plot I_var_train, the variational information on training data as a function of epoch
ax.plot(model.history['I_var'], label=r'I_var_train')

# Plot I_var_val, the variational information on validation data as a function of epoch
ax.plot(model.history['val_I_var'], label=r'val_I_var')

# Show I_pred_test, the predictive information of the final model on test data
ax.axhline(I_pred, color='C3', linestyle=':', label=r'test_I_pred')

# Style plot
ax.set_xlabel('epochs')
ax.set_ylabel('bits')
ax.set_title('Training history: variational information')
ax.legend()

plt.tight_layout()
```



[10]:
```
# Get the trained model parameters
# Retrieve G-P map parameter dict and view dict keys
param_dict = model.layer_gpmap.get_params()
param_dict.keys()
```

[10]:
```
dict_keys(['theta_tf_0', 'theta_rnap_0', 'theta_tf', 'theta_rnap', 'theta_dG_I'])
```

Authors in Ref. [1], reported they inferred CRP-RNAP interaction energy to be $\Delta G_I = 3.26$ kcalmol. The MAVE-NN prediction is very similar to the reported value, while it is several order of magnitude faster than the method used in Ref. [1].

[11]:
```
delta_G_I = param_dict['theta_dG_I']  # Gibbs energy of Interaction (scalar)
print(f'CRP-RNAP interaction energy = {delta_G_I*0.62:.3f} k_cal/mol')
```

```
CRP-RNAP interaction energy = -1.549 k_cal/mol
```

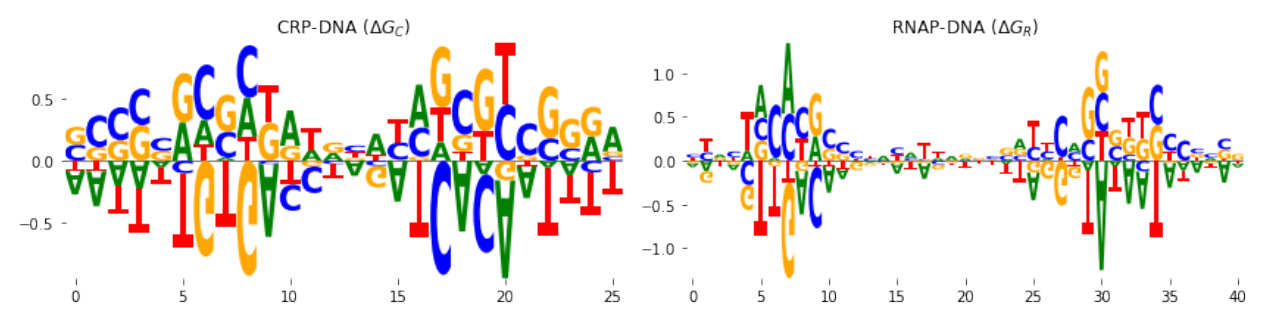
In addition, we can represent the CRP-DNA ΔG_C and RNAP-DNA ΔG_R binding energies in the weight matrix form. The weight matrices can be represented by sequence logos. To do that, we used the a Python package `Logomaker` which is also developed in our research group and is freely available here. To represent the weight matrices in logo

(a) we get the trained `crp_weights` and `rnap_weights` values,

(b) we convert them to the `pandas.DataFrame` with column names being the nucleotide strings.

The `pandas.DataFrame` can easily imported in `Logomaker`. See the documentation of `Logomaker` for detailed description of the parameters one can pass to make logos.

[12]:
```
# import logomaker
import logomaker

# Get the \Delta G_C trained values (theta_tf)
crp_weights = param_dict['theta_tf']
# Get the \Delta G_R trained values (theta_rnap)
rnap_weights = param_dict['theta_rnap']
# Convert them to pandas dataframe
crp_df = pd.DataFrame(crp_weights, columns=model.alphabet)
rnap_df = pd.DataFrame(rnap_weights, columns=model.alphabet)


# plot logos
fig, axs = plt.subplots(1, 2, figsize=[12, 3])

# sequence logo for the CRP-DNA binding energy matrix
logo = logomaker.Logo(crp_df,  ax=axs[0], center_values=True)
axs[0].set_title('CRP-DNA ($\Delta G_C$)')
logo.style_spines(visible=False)


# sequence logo for the RNAP-DNA binding energy matrix
logo = logomaker.Logo(rnap_df, ax=axs[1], center_values=True)
axs[1].set_title('RNAP-DNA ($\Delta G_R$)')
logo.style_spines(visible=False)

plt.tight_layout()
```



2.5.2 References

1. Kinney J, Murugan A, Callan C, Cox E (2010). Using deep sequencing to characterize the biophysical mechanism of a transcriptional regulatory sequence. Proc Natl Acad Sci USA. 107(20):9158-9163.

CHAPTER

THREE

BUILT-IN DATASETS

MAVE-NN provides multiple built-in datasets that users can easily load and use to train their own models

3.1 Overview of built-in datasets

[1]:
```
import mavenn
```

MAVE-NN comes with multiple built-in datasets for use in training or evaluating models. These datasets can be accessed by passing a datset name to `mavenn.load_example_dataset()`. To get a list of valid datset names, execute this command without any arguments:

[2]:
```
mavenn.load_example_dataset()
```

```
Please enter a dataset name. Valid choices are:
"amyloid"
"gb1"
"mpsa"
"mpsa_replicate"
"nisthal"
"sortseq"
"tdp43"
```

Datasets are returned in the form of `pandas` dataframes. Common fields include:

- `'x'`: Assayed sequences, all of which are the same length.
- `'y'`: Values of continuous measurements (used to train GE models).
- `'ct_y'`: Read counts observed in bin number `y`, where `y` is an integer ranging from `0` to `Y-1` (used to train MPA models).
- `'set'`: Indicates whether each observation was reserved for the `'training'`, `'validation'`, or `'test'` set when inferring the corresponding example models provided with MAVE-NN.

Other fields are sometimes provided as well, e.g. the raw input and output counts used to compute measurement values.

3.2 gb1 dataset

[1]:
```
# Standard imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Special imports
import mavenn
```

3.2.1 Summary

The DMS dataset from Olson et al., 2014. The authors used an RNA display selection experiment to assay the binding of over half a million protein GB1 variants to IgG. These variants included all 1-point and 2-point mutations within the 55 residue GB1 sequence. Only the 2-point variants are included in this dataset.

Name: `'gb1'`

Reference: Olson C, Wu N, Sun R. A comprehensive biophysical description of pairwise epistasis throughout an entire protein domain. Curr Biol 24(22):2643-2651 (2014).

[2]:
```
mavenn.load_example_dataset('gb1')
```

[2]:
```
             set  dist  input_ct  selected_ct         y  \
0       training     2       173           33 -3.145154
1       training     2        18            8 -1.867676
2       training     2        66            2 -5.270800
3       training     2        72            1 -5.979498
4       training     2        69          168  0.481923
...          ...   ...       ...          ...       ...
530732  training     2       462          139 -2.515259
530733  training     2       317           84 -2.693165
530734  training     2       335           77 -2.896589
530735  training     2       148           28 -3.150861
530736  training     2        95           16 -3.287173

                                                        x
0       AAKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
1       ACKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
2       ADKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
3       AEKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
4       AFKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
...                                                   ...
530732  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
530733  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
530734  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
530735  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
530736  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...

[530737 rows x 6 columns]
```

3.2.2 Preprocessing

First we load the double-mutation dataset published by Olson et al. (2021).

[3]:
```
# Dataset is is at this URL:
# url = 'https://ars.els-cdn.com/content/image/1-s2.0-S0960982214012688-mmc2.xlsx'

# We have downloaded this Excel file and reformatted it into a more parseable form
raw_data_file = '../../mavenn/examples/datasets/raw/gb1_raw.xlsx'

# Load data (takes a while)
double_mut_df = pd.read_excel(raw_data_file, sheet_name='double_mutants')
double_mut_df
```

[3]:
```
       Mut1 WT amino acid  Mut1 Position Mut1 Mutation Mut2 WT amino acid  \
0                       Q              2             A                  Y
1                       Q              2             A                  Y
2                       Q              2             A                  Y
3                       Q              2             A                  Y
4                       Q              2             A                  Y
...                   ...            ...           ...                ...
535912                  E             56             Y                  T
535913                  E             56             Y                  T
535914                  E             56             Y                  T
535915                  E             56             Y                  T
535916                  E             56             Y                  T

        Mut2 Position Mut2 Mutation  Input Count  Selection Count  \
0                   3             A          173               33
1                   3             C           18                8
2                   3             D           66                2
3                   3             E           72                1
4                   3             F           69              168
...               ...           ...          ...              ...
535912             55             R          462              139
535913             55             S          317               84
535914             55             V          335               77
535915             55             W          148               28
535916             55             Y           95               16

        Mut1 Fitness  Mut2 Fitness
0              1.518         0.579
1              1.518         0.616
2              1.518         0.010
3              1.518         0.009
4              1.518         1.054
...              ...           ...
535912         0.190         0.941
535913         0.190         0.840
535914         0.190         0.669
535915         0.190         0.798
535916         0.190         0.663

[535917 rows x 10 columns]
```

Next we reconstruct the wild-type GB1 sequence

[4]:
```
# Get unique WT pos-aa associations, sorted by position
wt_1_df = double_mut_df[['Mut1 Position', 'Mut1 WT amino acid']].copy()
wt_1_df.columns = ['pos','aa']
wt_2_df = double_mut_df[['Mut2 Position', 'Mut2 WT amino acid']].copy()
wt_2_df.columns = ['pos','aa']
wt_seq_df = pd.concat([wt_1_df, wt_2_df], axis=0).drop_duplicates().sort_values(by='pos
↪').reset_index(drop=True)

# Confirm that each position occurs at most once
assert np.all(wt_seq_df['pos'].value_counts()==1)

# Confirm that the set of unique positions is correct
L = len(wt_seq_df)
assert set(wt_seq_df['pos'].values) == set(range(2,L+2))

# Compute wt_seq and confirm its identity
wt_seq = ''.join(wt_seq_df['aa'])
known_wt_seq = 'QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDDATKTFTVTE'
assert wt_seq == known_wt_seq

# Print final wt sequence
print(f'WT sequence: {wt_seq}')
```

```
WT sequence: QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDDATKTFTVTE
```

Next we convert the list of mutations to an array `x` of variant sequences.

[5]:
```
# Introduce double mutations into WT sequence and append to a growing list
pos1s = double_mut_df['Mut1 Position'].values-2
pos2s = double_mut_df['Mut2 Position'].values-2
aa1s = double_mut_df['Mut1 Mutation'].values
aa2s = double_mut_df['Mut2 Mutation'].values
x_list = []
for pos1, aa1, pos2, aa2 in zip(pos1s, aa1s, pos2s, aa2s):
    mut_seq_list = list(wt_seq)
    mut_seq_list[pos1] = aa1
    mut_seq_list[pos2] = aa2
    mut_seq = ''.join(mut_seq_list)
    x_list.append(mut_seq)

# Convert list to an np.array and preview it
x = np.array(x_list)
x
```

[5]:
```
array(['AAKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDDATKTFTVTE',
       'ACKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDDATKTFTVTE',
       'ADKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDDATKTFTVTE', ...,
       'QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDDATKTFTVVY',
       'QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDDATKTFTVWY',
       'QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDDATKTFTVYY'],
      dtype='<U55')
```

We then compute the log2 enrichment `y` for each variant sequence in `x`.

[6]:
```
# Extract input and output counts
in_ct = double_mut_df['Input Count'].values
out_ct = double_mut_df['Selection Count'].values

# Read in wt data (on separate sheet) and compute baseline for y
wt_df = pd.read_excel(raw_data_file, sheet_name='wild_type')
wt_in_ct = wt_df['Input Count'][0]
wt_out_ct = wt_df['Selection Count'][0]

# Compute log2 enrichment values relative to WT
y = np.log2((out_ct+1)/(in_ct+1)) - np.log2((wt_out_ct+1)/(wt_in_ct+1))
y
```

[6]:
```
array([-3.14515399, -1.86767585, -5.27080003, ..., -2.89658854,
       -3.15086086, -3.287173  ])
```

Next we assign each sequence to the training, validation, or test set using a split of 90%:5%:5%.

[7]:
```
# Assign to training, validation, or test set
np.random.seed(0)
sets = np.random.choice(a=['training', 'validation', 'test'],
                        p=[.90,.05,.05],
                        size=len(x))
sets
```

[7]:
```
array(['training', 'training', 'training', ..., 'training', 'training',
       'training'], dtype='<U10')
```

Finally we assemble all relevant information into a dataframe and save.

[8]:
```
# Assemble into dataframe
final_df = pd.DataFrame({'set':sets, 'dist':2, 'input_ct':in_ct, 'selected_ct':out_ct, 'y
↪':y, 'x':x})

# Keep only sequences with input_ct >= 10
final_df = final_df[final_df['input_ct']>=10].reset_index(drop=True)

# Save to file (uncomment to execute)
# final_df.to_csv('gb1_data.csv.gz', index=False, compression='gzip')

# Preview dataframe
final_df
```

[8]:

	set	dist	input_ct	selected_ct	y	\
0	training	2	173	33	-3.145154	
1	training	2	18	8	-1.867676	
2	training	2	66	2	-5.270800	
3	training	2	72	1	-5.979498	
4	training	2	69	168	0.481923	
...	...	...	...	...	...	
530732	training	2	462	139	-2.515259	
530733	training	2	317	84	-2.693165	
530734	training	2	335	77	-2.896589	
530735	training	2	148	28	-3.150861	
530736	training	2	95	16	-3.287173	

(continues on next page)

(continued from previous page)

```
                                                        x
0       AAKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
1       ACKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
2       ADKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
3       AEKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
4       AFKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
...                                                   ...
530732  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
530733  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
530734  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
530735  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...
530736  QYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...

[530737 rows x 6 columns]
```

3.3 nisthal dataset

[1]:
```
# Standard imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Special imports
import mavenn
```

3.3.1 Summary

The DMS dataset from Nisthal et al. (2019). The authors used a high-throughput protein stability assay to measure folding energies for single-mutant variants of GB1. Column `'x'` list variant GB1 sequences (positions 2-56). Column `'y'` lists the Gibbs free energy of folding (i.e., ΔG_F) in units of kcal/mol; lower energy values correspond to increased protein stability. Sequences are not divided into training, validation, and test sets because this dataset is only used for validation in Tareen et al. (2021).

Name: `'nisthal'`

Reference: Nisthal A, Wang CY, Ary ML, Mayo SL. Protein stability engineering insights revealed by domain-wide comprehensive mutagenesis. Proc Natl Acad Sci USA 116:16367–16377 (2019).

[2]:
```
mavenn.load_example_dataset('nisthal')
```

[2]:
```
                                                    x  name       y
0    AYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02A  0.4704
1    DYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02D  0.5538
2    EYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02E -0.1299
3    FYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02F -0.3008
4    GYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02G  0.6680
..                                                 ...   ...     ...
913  TYTLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  K04T -0.4815
```

(continues on next page)

(continued from previous page)

```
914  TYVLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  K04V  0.2696
915  TYYLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  K04Y -0.8246
916  VYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02V -1.3090
917  YYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02Y -0.1476

[918 rows x 3 columns]
```

3.3.2 Preprocessing

First we load and preview the raw dataset published by Nisthal et al. (2019)

[3]:
```
raw_data_file = '../../mavenn/examples/datasets/raw/nisthal_raw.csv'
raw_df = pd.read_csv(raw_data_file)
raw_df
```

[3]:
```
                                              Sequence Description  Ligand  \
0      ATYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYD...        M01A     NaN
1      ATYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYD...        M01A     NaN
2      ATYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYD...        M01A     NaN
3      ATYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYD...        M01A     NaN
4      ATYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYD...        M01A     NaN
...                                                  ...         ...     ...
18856  YTYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYD...        M01Y     NaN
18857  YTYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYD...        M01Y     NaN
18858  YTYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYD...        M01Y     NaN
18859  YTYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYD...        M01Y     NaN
18860  YTYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYD...        M01Y     NaN

         Data       Units       Assay/Protocol
0         NaN    kcal/mol   ddG(deepseq)_Olson
1         NaN    kcal/mol    ddG_lit_fromOlson
2      -1.777  kcal/mol·M              m-value
3      -0.635    kcal/mol              FullMin
4      -0.510    kcal/mol  Rosetta SomeMin_ddG
...       ...         ...                  ...
18856   0.512    kcal/mol   SD of dG(H2O)_mean
18857   0.680    kcal/mol       ddG(mAvg)_mean
18858   2.691   M (Molar)                   Cm
18859   4.519    kcal/mol         dG(H2O)_mean
18860   4.630    kcal/mol        dG(mAvg)_mean

[18861 rows x 6 columns]
```

Next we do the following: - Select rows that have the value `'ddG(mAvg)_mean'` in the `'Assay/Protocol'` column. - Keep only the desired columns, and given them shorter names - Remove position 1 from variant sequences and drop duplicate sequences - Flip the sign of measured folding energies - Drop variants with ΔG values of exactly $+4$ kcal/mol, as these were not precisely measured. - Save the dataframe if desired

[5]:
```
# Select rows that have the value `'ddG(mAvg)_mean'` in the `'Assay/Protocol'` column.
data_df = raw_df[raw_df['Assay/Protocol']=='ddG(mAvg)_mean'].copy()
```

(continues on next page)

(continued from previous page)

```
# Keep only the desired columns, and given them shorter names
data_df.rename(columns={'Sequence':'x', 'Data': 'y', 'Description':'name'}, inplace=True)
cols_to_keep = ['x', 'name', 'y']
data_df = data_df[cols_to_keep]

# Remove position 1 from variant sequences and drop duplicate sequences
data_df['x'] = data_df['x'].str[1:]
data_df.drop_duplicates(subset='x', keep=False, inplace=True)

# Flip the sign of measured folding energies
data_df['y'] = -data_df['y']

# Drop variants with $\Delta G$ of exactly $+4$ kcal/mol, as these were not precisely␣
↪measured.
ix = data_df['y']==4
data_df = data_df[~ix]
data_df.reset_index(inplace=True, drop=True)

# Save to file (uncomment to execute)
# data_df.to_csv('nisthal_data.csv.gz', index=False, compression='gzip')
data_df
```

[5]:

```
                                                     x  name       y
0    AYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02A  0.4704
1    DYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02D  0.5538
2    EYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02E -0.1299
3    FYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02F -0.3008
4    GYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02G  0.6680
..                                                 ...   ...     ...
808  TYTLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  K04T -0.4815
809  TYVLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  K04V  0.2696
810  TYYLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  K04Y -0.8246
811  VYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02V -1.3090
812  YYKLILNGKTLKGETTTEAVDAATAEKVFKQYANDNGVDGEWTYDD...  T02Y -0.1476

[813 rows x 3 columns]
```

3.4 amyloid dataset

[1]:

```
# Standard imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Special imports
import mavenn
import os
import urllib
```

3.4.1 Summary

The deep mutational scanning (DMS) dataset of Seuma et al., 2021. The function of small protein called amyloid beta (A:math:*beta*) is suspected to play a significant role in Alzheimer's disease. By mutating each position in the protein, Seuma et al. produced more than 14,000 different versions of Aβ with single and double mutation. To globally quantify the impact of mutations, they used *in-vivo* selection assay using yeast cells and measured how quickly these mutants were able to aggregate. The quantification is summarized in the variable called nucleation score.

Names: `'amyloid'`

Reference: Seuma M, Faure A, Badia M, Lehner B, Bolognesi B. The genetic landscape for amyloid beta fibril nucleation accurately discriminates familial Alzheimer's disease mutations. eLife 10:e63364 (2021).

[2]:
```
mavenn.load_example_dataset('amyloid')
```

[2]:
```
            set  dist         y        dy  \
0      training     1 -0.117352  0.387033
1      training     1  0.352500  0.062247
2      training     1 -2.818013  1.068137
3      training     1  0.121805  0.376764
4      training     1 -2.404340  0.278486
...         ...   ...       ...       ...
16061  training     2 -0.151502  0.389821
16062  training     2 -1.360708  0.370517
16063  training     2 -0.996816  0.346949
16064  training     2 -3.238403  0.429008
16065  training     2 -1.141457  0.365638

                                            x
0      KAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
1      NAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
2      TAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
3      SAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
4      IAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
...                                           ...
16061  DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVKV
16062  DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVLV
16063  DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVMV
16064  DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVTV
16065  DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVVV

[16066 rows x 5 columns]
```

3.4.2 Preprocessing

The DMS dataset of single and double mutations in Aβ of Seuma et al., (2021) is publicly available in the excel format on the Gene Expression Omnibus server. It is formatted as follows:

- Single mutated sequences are in `1 aa change sheet`. For these sequences the `Pos` column lists the amino acid (aa) position which mutated, and `Mut` column is mutated aa residue.
- Double mutated sequences are in `2 aa change sheet`. For these sequences the `Pos1` and `Pos2` columns list the first and second aa positions which mutated. `Mut1` and `Mut2` columns are residues of mutation 1 and 2 in double mutant, respectively.

- Both single and double mutant consist of the nucleation scores across three replicates and the weighted average (`nscore`) of them based on their uncertainties (`sigma`).

[3]:
```
# Download datset
url = 'https://www.ncbi.nlm.nih.gov/geo/download/?acc=GSE151147&format=file&
↪file=GSE151147%5FMS%5FBL%5FBB%5Fprocessed%5Fdata%2Exlsx'
raw_data_file = 'Abeta_raw_data.xlsx'
urllib.request.urlretrieve(url, raw_data_file)

# Record wild-type sequence
wt_seq = 'DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA'

# Read single mutation sheet from raw data
single_mut_df = pd.read_excel(raw_data_file, sheet_name='1 aa change')

# Read double mutation sheet from raw data
double_mut_df = pd.read_excel(raw_data_file, sheet_name='2 aa changes')
```

[4]:
```
# Preview single-mutant data
single_mut_df.head()
```

[4]:
```
   Pos WT_AA Mut  Nham_nt  Nham_aa  Nmut_codons   STOP    mean_count  \
0    1     D   K        2        1            1  False    210.500000
1    1     D   N        2        1            1  False  28544.000000
2    1     D   T        2        1            1  False     97.000000
3    1     D   S        2        1            1  False    150.666667
4    1     D   I        2        1            1  False    334.333333

    nscore1    sigma1   nscore2    sigma2   nscore3    sigma3    nscore  \
0 -0.280176  0.482820  0.175372  0.647374       NaN       NaN -0.117352
1  0.388480  0.112041  0.306589  0.077314  0.785219  0.299795  0.352500
2       NaN       NaN -2.818013  1.068137       NaN       NaN -2.818013
3  0.003406  0.525670  0.180478  0.622756  0.448936  1.086370  0.121805
4 -2.364750  0.373224 -2.579152  0.482386 -2.074932  0.839842 -2.404340

      sigma
0  0.387033
1  0.062247
2  1.068137
3  0.376764
4  0.278486
```

[5]:
```
# Preview double-mutant data
double_mut_df.head()
```

[5]:
```
   Pos2 Mut2  Pos1 Mut1 WT_AA1 WT_AA2  Nham_nt  Nham_aa  Nmut_codons   STOP  \
0     2    E     1    E      D      A        2        2            2  False
1     2    E     1    G      D      A        2        2            2  False
2     2    E     1    N      D      A        2        2            2  False
3     2    E     1    V      D      A        2        2            2  False
4     2    E     1    Y      D      A        2        2            2  False

   mean_count   nscore1    sigma1   nscore2    sigma2   nscore3    sigma3  \
0   78.500000  0.160562  0.878728 -1.908344  0.999612       NaN       NaN
```

(continues on next page)

(continued from previous page)

```
1  139.500000 -0.461932  0.679144 -0.616485  0.715070       NaN       NaN
2  146.000000  0.143146  0.530710 -0.181673  0.855333       NaN       NaN
3  133.333333 -0.526572  0.551242 -1.427565  0.708833 -0.423844  1.053086
4   62.000000 -0.288245  0.876578       NaN       NaN       NaN       NaN

     nscore     sigma
0 -0.741292  0.659978
1 -0.535229  0.492438
2  0.052856  0.450957
3 -0.801619  0.402165
4 -0.288245  0.876578
```

To reformat `single_mut_df` and `double_mut_df` into the one provided with MAVE-NN, we first need to get the full sequence of amino acids corresponding to each mutation. Therefore, we used `Pos` and `Mut` columns to replace single aa in wild type sequence for each record for the single mutant. Then, we used `Pos1`, `Pos2`, `Mut1` and `Mut2` from the double mutants to replace two aa in the wild type sequence. The list of sequences with single and double mutants are called `single_mut_list` and `double_mut_list`, respectively. Those lists are then horizontally (column wise) stacked in `x` variable.

Next, we stack single- and double-mutant - nucleation scores `nscore` in `y` - score uncertainties `sigma` in `dy` - hamming distance in `dists`

Finally, we create a `set` column that randomly assigns each sequence to the training, test, or validation set (using a 90:05:05 split), then reorder the columns for clarity. The resulting dataframe is called final_df.

[6]:
```
# Introduce single mutations into wt sequence and append to a list
single_mut_list = []
for mut_pos, mut_char in zip(single_mut_df['Pos'].values,
                             single_mut_df['Mut'].values):
    mut_seq = list(wt_seq)
    mut_seq[mut_pos-1] = mut_char
    single_mut_list.append(''.join(mut_seq))

# Introduce double mutations into wt sequence and append to list
double_mut_list = []
for mut1_pos, mut1_char, mut2_pos, mut2_char in zip(double_mut_df['Pos1'].values,
                                                    double_mut_df['Mut1'].values,
                                                    double_mut_df['Pos2'].values,
                                                    double_mut_df['Mut2'].values):
    mut_seq = list(wt_seq)
    mut_seq[mut1_pos-1] = mut1_char
    mut_seq[mut2_pos-1] = mut2_char
    double_mut_list.append(''.join(mut_seq))

# Stack single-mutant and double-mutant sequences
x = np.hstack([single_mut_list,
               double_mut_list])

# Stack single-mutant and double-mutant nucleation scores
y = np.hstack([single_mut_df['nscore'].values,
               double_mut_df['nscore'].values])

# Stack single-mutant and double-mutant nucleation score uncertainties
```

(continues on next page)

(continued from previous page)

```
dy = np.hstack([single_mut_df['sigma'].values,
                double_mut_df['sigma'].values])

# List hamming distances
dists = np.hstack([1*np.ones(len(single_mut_df)),
                   2*np.ones(len(double_mut_df))]).astype(int)

# Assign each sequence to training, validation, or test set
np.random.seed(0)
sets = np.random.choice(a=['training', 'validation', 'test'],
                        p=[.9,.05,.05],
                        size=len(x))

# Assemble into dataframe
final_df = pd.DataFrame({'set':sets, 'dist':dists, 'y':y, 'dy':dy, 'x':x})

# # Save to file (uncomment to execute)
# final_df.to_csv('amyloid_data.csv.gz', index=False, compression='gzip')

# Preview dataframe
final_df
```

[6]:

```
            set  dist         y        dy  \
0      training     1 -0.117352  0.387033
1      training     1  0.352500  0.062247
2      training     1 -2.818013  1.068137
3      training     1  0.121805  0.376764
4      training     1 -2.404340  0.278486
...         ...   ...       ...       ...
16061  training     2 -0.151502  0.389821
16062  training     2 -1.360708  0.370517
16063  training     2 -0.996816  0.346949
16064  training     2 -3.238403  0.429008
16065  training     2 -1.141457  0.365638

                                            x
0      KAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
1      NAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
2      TAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
3      SAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
4      IAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVIA
...                                           ...
16061  DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVKV
16062  DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVLV
16063  DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVMV
16064  DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVTV
16065  DAEFRHDSGYEVHHQKLVFFAEDVGSNKGAIIGLMVGGVVVV

[16066 rows x 5 columns]
```

This final dataframe, final_df, has the same format as the `amyloid` dataset that comes with MAVE-NN.

3.5 tdp43 dataset

[1]:
```
# Standard imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Special imports
import mavenn
import os
import urllib
```

3.5.1 Summary

The deep mutagenesis dataset of Bolognesi et al., 2019. TAR DNA-binding protein 43 (TDP-43) is a heterogeneous nuclear ribonucleoprotein (hnRNP) in the cell nucleus which has a key role in regulating gene expression. Several neurodegenerative disorders have been associated with cytoplasmic aggregation of TDP-43, including amyotrophic lateral sclerosis (ALS), frontotemporal lobar degeneration (FTLD), Alzheimer's, Parkinson's, and Huntington's disease. Bolognesi et al., performed a comprehensive deep mutagenesis, using error-prone oligonucleotide synthesis to comprehensively mutate the prion-like domain (PRD) of TDP-43 and reported toxicity as a function of 1266 single and 56730 double mutations.

Names: `'tdp43'`

Reference: Benedetta B, Faure AJ, Seuma M, Schmiedel JM, Tartaglia GG, Lehner B. The mutational landscape of a prion-like domain. Nature Comm 10:4162 (2019).

[2]:
```
mavenn.load_example_dataset('tdp43')
```

[2]:
```
              set  dist         y        dy  \
0        training     1  0.032210  0.037438
1        training     1 -0.009898  0.038981
2        training     1 -0.010471  0.005176
3        training     1  0.030803  0.005341
4        training     1 -0.054716  0.035752
...           ...   ...       ...       ...
57991    training     2 -0.009706  0.035128
57992  validation     2 -0.030744  0.029436
57993  validation     2 -0.086802  0.033174
57994    training     2 -0.049587  0.029130
57995    training     2 -0.105390  0.031189

                                                       x
0      NNSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWG...
1      TNSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWG...
2      RNSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWG...
3      SNSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWG...
4      INSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWG...
...                                                  ...
57991  GNSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWG...
57992  GNSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWG...
57993  GNSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWG...
```

(continues on next page)

(continued from previous page)

```
57994  GNSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWG...
57995  GNSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWG...

[57996 rows x 5 columns]
```

3.5.2 Preprocessing

The deep mutagenesis dataset for single and double mutations in TDP-43 is publicly available (in excel format) in the **supplementary information/Supplementary Data 3** of the Bolognesi et al. published paper.

It is formatted as follows: - The wild type sequence absolute starting position is 290.

- Single mutated sequences are in the `1 AA change` sheet. For these sequences the `Pos_abs` column lists the absolute position of the amino acid (aa) which mutated with `Mut` column.
- Double mutated sequences are in `2 AA change` sheet. For these sequences the `Pos_abs1` and `Pos_abs2` columns list the first and second aa absolute positions which mutated. `Mut1` and `Mut2` columns are residues of mutation position 1 and 2 in double mutant, respectively.
- Both single and double mutants consist of the toxicity scores (measurements `y`) and corresponding uncertainties `dy`.
  - We will use the `toxicity` and `sigma` columns for single mutant sequences.
  - We will use the corrected relative toxicity `toxicity_cond` and the corresponding corrected uncertainty `sigma_cond` (see Methods section of the Reference paper).

[5]:
```
# Download datset
url = 'https://github.com/jbkinney/mavenn/blob/master/mavenn/examples/datasets/raw/tdp-
↪43_raw.xlsx?raw=true'
raw_data_file = 'tdp-43_raw.xlsx'
urllib.request.urlretrieve(url, raw_data_file)

# Record wild-type sequence
wt_seq =
↪'GNSRGGGAGLGNNQGSNMGGGMNFGAFSINPAMMAAAQAALQSSWGMMGMLASQQNQSGPSGNNQNQGNMQREPNQAFGSGNNS'

# Read single mutation sheet from raw data
single_mut_df = pd.read_excel(raw_data_file, sheet_name='1 AA change')

# Read double mutation sheet from raw data
double_mut_df = pd.read_excel(raw_data_file, sheet_name='2 AA change')

# Delete raw dataset
os.remove(raw_data_file)
```

[6]:
```
# Preview single-mutant data
single_mut_df.head()
```

[6]:

	Pos	WT_AA	Mut	Nmut_nt	Nmut_aa	Nmut_codons	STOP	mean_count	\
0	1	G	N	2	1	1	False	22.000000	
1	1	G	T	2	1	1	False	17.333333	
2	1	G	R	2	1	1	False	3888.666667	
3	1	G	S	2	1	1	False	3635.666667	

(continues on next page)

(continued from previous page)

```
4    1     G   I        2        1            1  False    21.666667

   is.reads0     sigma  toxicity  region  Pos_abs mut_code
0       True  0.037438  0.032210     290      290    G290N
1       True  0.038981 -0.009898     290      290    G290T
2       True  0.005176 -0.010471     290      290    G290R
3       True  0.005341  0.030803     290      290    G290S
4       True  0.035752 -0.054716     290      290    G290I
```

[7]:
```
# Preview double-mutant data
double_mut_df.head()
```

[7]:
```
   Nmut_nt  Nmut_aa  Nmut_codons  STOP  mean_count  is.reads0  Pos1  Pos2  \
0        2        2            2  True   16.333333       True     1     4
1        4        2            2  True   30.333333       True     1     4
2        2        2            2  True   43.333333       True     1     4
3        2        2            2  True   22.333333       True     1     4
4        2        2            2  True   29.333333       True     1     4

  WT_AA1 WT_AA2  ... sigma_cond toxicity1  toxicity2  toxicity_uncorr  \
0      G      R  ...   0.020867  0.001282  -0.174307        -0.139949
1      G      R  ...   0.017555  0.007680  -0.174307        -0.206614
2      G      R  ...   0.017882  0.044342  -0.174307        -0.123376
3      G      R  ...   0.018913 -0.010471  -0.174307        -0.136759
4      G      R  ...   0.021690  0.030803  -0.174307        -0.118746

   toxicity_cond  region  Pos_abs1  Pos_abs2  mut_code1  mut_code2
0      -0.169501     290       290       293      G290A      R293*
1      -0.193387     290       290       293      G290C      R293*
2      -0.142809     290       290       293      G290D      R293*
3      -0.165018     290       290       293      G290R      R293*
4      -0.153186     290       290       293      G290S      R293*

[5 rows x 25 columns]
```

To reformat `single_mut_df` and `double_mut_df` into the one provided with MAVE-NN, we first need to get the full sequence of amino acids corresponding to each mutation. Therefore, we used `Pos` and `Mut` columns to replace single aa in the wild type sequence for each record in the single mutant dataset. Then, we used `Pos_abs1`, `Pos_abs2`, `Mut1` and `Mut2` from the double mutants to replace two aa in the wild type sequence. The list of sequences with single and double mutants are called `single_mut_list` and `double_mut_list`, respectively. Those lists are then horizontally (column wise) stacked in the `x` variable.

Next, we stack single- and double-mutant - nucleation scores `toxicity` and `toxicity_cond` in `y` - score uncertainties `sigma` and `sigma_cond` in `dy` - hamming distances in `dist`

Finally, we create a `set` column that randomly assigns each sequence to the training, test, or validation set (using a 90:05:05 split), then reorder the columns for clarity. The resulting dataframe is called `final_df`.

[ ]:
```
# Introduce single mutations into wt sequence and append to a list
single_mut_list = []
for mut_pos, mut_char in zip(single_mut_df['Pos_abs'].values,
                             single_mut_df['Mut'].values):
    mut_seq = list(wt_seq)
```

(continues on next page)

(continued from previous page)

```
    mut_seq[mut_pos-290] = mut_char
    single_mut_list.append(''.join(mut_seq))

# Introduce double mutations into wt sequence and append to list
double_mut_list = []
for mut1_pos, mut1_char, mut2_pos, mut2_char in zip(double_mut_df['Pos_abs1'].values,
                                                    double_mut_df['Mut1'].values,
                                                    double_mut_df['Pos_abs2'].values,
                                                    double_mut_df['Mut2'].values):
    mut_seq = list(wt_seq)
    mut_seq[mut1_pos-290] = mut1_char
    mut_seq[mut2_pos-290] = mut2_char
    double_mut_list.append(''.join(mut_seq))

# Stack single-mutant and double-mutant sequences
x = np.hstack([single_mut_list,
               double_mut_list])

# Stack single-mutant and double-mutant nucleation scores
y = np.hstack([single_mut_df['toxicity'].values,
               double_mut_df['toxicity_cond'].values])

# Stack single-mutant and double-mutant nucleation score uncertainties
dy = np.hstack([single_mut_df['sigma'].values,
                double_mut_df['sigma_cond'].values])

# List hamming distances
dists = np.hstack([1*np.ones(len(single_mut_df)),
                   2*np.ones(len(double_mut_df))]).astype(int)

# Assign each sequence to training, validation, or test set
np.random.seed(0)
sets = np.random.choice(a=['training', 'validation', 'test'],
                        p=[.9,.05,.05],
                        size=len(x))

# Assemble into dataframe
final_df = pd.DataFrame({'set':sets, 'dist':dists, 'y':y, 'dy':dy, 'x':x})

# # Save to file (uncomment to execute)
final_df.to_csv('tdp43_data.csv.gz', index=False, compression='gzip')

# Preview dataframe
final_df
```

This final dataframe, `final_df`, has the same format as the `tdp43` dataset that comes with MAVE-NN.

3.6 mpsa datasets

[1]:
```
# Standard imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Special imports
import mavenn
import os
import urllib
```

3.6.1 Summary

The massively parallel splicing assay (MPSA) dataset of Wong et al., 2018. The authors used 3-exon minigenes to assay how inclusion of the middle exon varies with the sequence of that exon's 5' splice site. Nearly all 5' splice site variants of the form NNN/GYNNNN were measured, where the slash demarcates the exon/intron boundary. The authors performed experiments on multiple replicates of multiple libraries in three different minigene contexts: *IKBKAP* exons 19-21, *SMN1* exons 6-8, and *BRCA2* exons 17-19. The dataset `'mpsa'` is from library 1 replicate 1 in the *BRCA2* context, while `'mpsa_replicate'` is from library 2 replicate 1 in the same context.

In these dataframes, the `'tot_ct'` column reports the number of reads obtained for each splice site from total processed mRNA transcripts, the `'ex_ct'` column reports the number of reads obtained from processed mRNA transcripts containing the central exon, `'y'` is the $\log_{10}$ percent-spliced-in (PSI) value measured for each sequence, and `'x'` is the variant 5' splice site. Note that some sequences have $y > 2.0$, corresponding to PSI > 100, due to experimental noise.

Names: `'mpsa'`, `'mpsa_replicate'`

Reference: Wong MS, Kinney JB, Krainer AR. Quantitative activity profile and context dependence of all human 5' splice sites. Mol Cell. 71:1012-1026.e3 (2018).

[2]:
```
mavenn.load_example_dataset('mpsa')
```

[2]:

	set	tot_ct	ex_ct	y	x
0	training	28	2	0.023406	GGAGUGAUG
1	test	315	7	-0.587914	AGUGUGCAA
2	training	193	15	-0.074999	UUCGCGCCA
3	training	27	0	-0.438475	UAAGCUUUU
4	training	130	2	-0.631467	AUGGUCGGG
...	...	...	...	...	...
30478	training	190	17	-0.017078	CUGGUUGCA
30479	training	154	10	-0.140256	CGCGCACAA
30480	test	407	16	-0.371528	ACUGCUCAC
30481	training	265	6	-0.571100	AUAGUCUAA
30482	test	26	22	0.939047	GUGGUAACU

[30483 rows x 5 columns]

3.6.2 Preprocessing

The raw datasets for both `'mpsa'` and `'mpsa_replicate'` are available at github.com/jbkinney/15_splicing/.

[3]:
```
# Specify dataset
dataset = 'mpsa' # use to make 'mpsa' dataset
# dataset = 'mpsa_replicate' # uncomment to make 'mpsa_replicate' dataset instead
print(f'Creating dataset "{dataset}"...')

# Specify online directory
url_dir = 'https://github.com/jbkinney/15_splicing/raw/master/for_mavenn/'

# Specify raw data file and output file names
if dataset=='mpsa':
    raw_data_file = 'results.brca2_9nt_lib1_rep1.txt.gz'
elif dataset=='mpsa_replicate':
    raw_data_file = 'results.brca2_9nt_lib2_rep1.txt.gz'
else:
    assert False

# Download raw datasert
print(f'Retrieving data from {url_dir+raw_data_file}...')
urllib.request.urlretrieve(url_dir+raw_data_file, raw_data_file)

# Load raw datset
raw_df = pd.read_csv(raw_data_file, sep='\t', index_col=0, compression='gzip')

# Delete raw dataset
os.remove(raw_data_file)

# Preview raw dataset
print('Done!')
raw_df
```

```
Creating dataset "mpsa"...
Retrieving data from https://github.com/jbkinney/15_splicing/raw/master/for_mavenn/
↪results.brca2_9nt_lib1_rep1.txt.gz...
Done!
```

[3]:

	tot_ct	ex_ct	lib_ct	mis_ct	ss	bc
0	377	27	164	3	ACAGCGGGA	TTAGCTATCGGCTGACGTCT
1	332	5	97	1	AGCGTGTAT	CCACCCAACGCGCCGTCAGT
2	320	3286	46	1	CAGGTGAGA	TTGAGGTACACTGAACAGTC
3	312	2248	87	1	CAGGTTAGA	ACCGATCTGCCACGGCGACC
4	291	8	109	2	CAAGCCTTA	AGGGACCATCCAGTTCGCCT
...	...	...	...	...	...	...
944960	0	0	14	0	ACCGCGATG	TGAAATTGACCCGAGCCTGC
944961	0	0	14	1	AACGCCTCG	AACCAAAATACCTTGCGCTT
944962	0	0	14	0	TACGCATCG	TACTCAGCCAATGGCGAACA
944963	0	0	14	0	AAGGTCACG	CTATGCATCTACGCTTAATG
944964	0	0	2	0	CCAGCGCCG	AAAAAAAAAAAAGATTTGTT

[944965 rows x 6 columns]

Each raw dataset lists read counts for every cloned minigene plasmid. Note that each minigene contained one 5'ss

(column 'ss') and one associated barcode (column 'bc'). Each barcode is associated with a unique 5'ss, but each 5'ss will typically be associated with many different barcodes. In addition to 'tot_ct' and 'ex_ct', two other read count quantities are listed: 'lib_ct' is the number of reads supporting the indicated 5'ss-barcode association, while 'mis_ct' is the number of reads indicating association of the barcode with other 5'ss that are not shown. Filtering for 'lib_ct' >= 2 and 'lib_ct' >=4x 'mis_ct' was already performed by the authors, and we will ignore these columns in what follows.

To format this dataset for use in MAVE-NN, we do the following: - Drop the 'lib_ct' and 'mis_ct' columns. - Drop the 'bc' column and marginalize 'tot_ct' and 'ex_ct' by 5'ss sequence. - Rename the 'ss' column to 'x'. - Convert DNA 5'ss sequences to RNA sequences by replacing 'T' with 'U'. - Remove 5'ss with 'tot_ct' < 10 - Remove 5'ss with sequences that don't match 'NNNGYNNNN'. - Compute 'y', which lists estimated log10 PSI values. - Assign each 5'ss to the training, validation, or test set. - Shuffle the rows of the final dataframe and reorder the columns for clarity. - Save the dataframe as a .csv.gz file.

[4]:
```
# Remove unwanted columns
data_df = raw_df[['tot_ct','ex_ct','ss','bc']].copy()

# Marginalize by splice site
data_df = data_df.groupby('ss').sum().reset_index()

# Rename columns
data_df.rename(columns={'ss':'x'}, inplace=True)

# Make sequences RNA
data_df['x'] = [ss.replace('T','U') for ss in data_df['x']]

# Remove ss with minimum tot_ct
min_ct = 10
ix = data_df['tot_ct'] >= min_ct
data_df = data_df[ix].copy()
print(f'{(~ix).sum()} ss removed for having tot_ct < {min_ct}')

# Remove ss with invalid sequences
ix = np.array([((x[3]=='G') and (x[4] in {'C','U'})) for x in data_df['x']])
data_df = data_df[ix]
print(f'{(~ix).sum()} ss with invalid sequences removed')

# Get consensus i_cons and o_cons
pseudoct = 1.0
cons_seq = 'CAGGUAAGU'
tmp_df = data_df.set_index('x')
i_cons = tmp_df.loc[cons_seq,'tot_ct']
o_cons = tmp_df.loc[cons_seq,'ex_ct']
relative_psi_cons = (o_cons+pseudoct)/(i_cons+pseudoct)

# Compute y
i_n = data_df['tot_ct']
o_n = data_df['ex_ct']
relative_psi_n = (o_n+pseudoct)/(i_n+pseudoct)
psi_n = 100*relative_psi_n/relative_psi_cons
y_n = np.log10(psi_n)
data_df['y'] = y_n

# Assign data to training, validation, or test sets
```

(continues on next page)

(continued from previous page)

```
np.random.seed(0)
data_df['set'] = np.random.choice(a=['training', 'validation', 'test'],
                                  p=[.6,.2,.2],
                                  size=len(data_df))

# Shuffle data for extra safety
data_df = data_df.sample(frac=1).reset_index(drop=True)

# Order columns
final_df = data_df[['set', 'tot_ct', 'ex_ct', 'y', 'x']].copy()

# Save to file (uncomment to execute)
out_file = f'{dataset}_data.csv.gz'
final_df.to_csv(out_file, index=False, compression='gzip')
print(f'Dataset "{dataset}" saved to {out_file}')

# Preview final dataframe
print('Done!')
final_df
```

```
2309 ss removed for having tot_ct < 10
7 ss with invalid sequences removed
Dataset "mpsa" saved to mpsa_data.csv.gz
Done!
```

[4]:

	set	tot_ct	ex_ct	y	x
0	training	28	2	0.023406	GGAGUGAUG
1	test	315	7	-0.587914	AGUGUGCAA
2	training	193	15	-0.074999	UUCGCGCCA
3	training	27	0	-0.438475	UAAGCUUUU
4	training	130	2	-0.631467	AUGGUCGGG
...	...	...	...	...	...
30478	training	190	17	-0.017078	CUGGUUGCA
30479	training	154	10	-0.140256	CGCGCACAA
30480	test	407	16	-0.371528	ACUGCUCAC
30481	training	265	6	-0.571100	AUAGUCUAA
30482	test	26	22	0.939047	GUGGUAACU

[30483 rows x 5 columns]

3.7 sortseq dataset

[5]:

```
# Standard imports
import pandas as pd
import numpy as np

# Special imports
import mavenn
```

(continues on next page)

(continued from previous page)

```
import os
import urllib
```

3.7.1 Summary

The sort-seq MPRA data of Kinney et al., 2010. The authors used fluoresence-activated cell sorting, followed by deep sequencing, to assay gene expression levels from variant *lac* promoters in *E. coli*. The authors performed 6 different experiments, which varied in the region of the *lac* promoter that was mutagenized, the mutation rate used, the *E. coli* host strain, cellular growth conditions, and the number of bins into which cells were sorted. See Kinney et al., 2010 for more details.

In this dataframe, the `'x'` column lists (unique) variant sequences, columns `'ct_0'` through `'ct_9'` list the number of read counts for each sequence observed in each of the 10 respective FACS bins, and the `'set'` column indicates whether each sequence is assigned to the training set, the validation set, or the test set.

Names: `'sortseq'`

Associated datasets: `'sortseq_rnap-wt'`, `'sortseq_crp-wt'`, `'sortseq_full-500'`, `'sortseq_full-150'`, `'sortseq_full-0'`

Reference: Kinney J, Murugan A, Callan C, Cox E. Using deep sequencing to characterize the biophysical mechanism of a transcriptional regulatory sequence. Proc Natl Acad Sci USA. 107(20):9158-9163 (2010).

[2]:
```
mavenn.load_example_dataset('sortseq')
```

[2]:
```
              set  ct_0  ct_1  ct_2  ct_3  ct_4  ct_5  ct_6  ct_7  ct_8  ct_9  \
0            test     0     0     0     0     0     0     0     0     1     0
1        training     0     1     0     0     0     0     0     0     0     0
2        training     0     0     0     0     0     0     0     0     1     0
3            test     0     0     0     0     0     1     0     0     0     0
4            test     0     0     0     0     0     0     0     1     0     0
...           ...   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...
45773    training     0     0     0     1     0     0     0     0     0     0
45774  validation     2     0     0     0     0     0     0     0     0     0
45775    training     0     0     1     0     0     0     0     0     0     0
45776    training     2     0     0     0     0     0     0     0     0     0
45777  validation     0     0     0     0     2     0     0     0     0     0

                                                    x
0      GGCTTTACACTTTAAGCTGCCGCATCGTATGTTATGTGG
1      GGCTATACATTTTATGTTTCCGGGTCGTATTTTGTGTGG
2      GGCTTTACATTTTATGCTTCCTTCACGTATGTTGTGTCT
3      GGCATTACTCTTTGTGCTTCCGGCTCGTATGTTGTGTGG
4      GACTTTTCAATTTATGCTTTCAGTTGGTATGTTGTGTAG
...                                        ...
45773  GGCTTTTCACTTTATGCTTCTGGCTCGTATGTTGTGTGG
45774  GGTTTTACACTTTTTGCTTCCGGGCCAAATGTTGTGTGG
45775  GGCTCCACACATTATGCTTCCGGCTCGTCTGTTCGCTCG
45776  GGCTTTACACATTATGCTTCCGGCTCGTATGTTGTTTGG
45777  GGCTTTACACTTTATGCTTCCGGCACGTTTGTTGTGTGG

[45778 rows x 12 columns]
```

3.7.2 Preprocessing

The sort-seq MPRA dataset of Kinney et al., (2010) is available at https://github.com/jbkinney/09_sortseq/ in file `file_S2.txt.gz`. It is formatted as follows: the `'seq'` column lists (non-unique) variant 75 nt DNA sequences observed by high-throughput seuqencing, the `'experiment'` column lists which of the six reported experiments produced that sequence, and the `'bin'` column lists the FACS bin in which that sequence was observed. This dataframe is called `raw_df` in what follows.

[3]:
```
# Download datset
url = 'https://github.com/jbkinney/09_sortseq/raw/master/file_S2.txt.gz'
raw_data_file = 'file_S2.txt.gz'
urllib.request.urlretrieve(url, raw_data_file)

# Load raw dataset
raw_df = pd.read_csv('file_S2.txt.gz',
                     sep='\t',
                     header=None,
                     names=['experiment','bin','x'],
                     compression='gzip')

# Delete raw dataset
os.remove(raw_data_file)

# Preview raw_df
raw_df.head()
```

[3]:

	experiment	bin	x
0	crp-wt	B0	AATTAAGGGCAGTTAACTCACCCATTAGGCACCCCAGGCTTTACAC...
1	crp-wt	B0	AATTAATATGAGTTTGCTCACCCATTAGGCACCCCAGGCTTTACAC...
2	crp-wt	B0	AATTAATAAGAGTTCACTCACTCATACGGCACCCCAGGCTTTACAC...
3	crp-wt	B0	AATTTATGTGCTTTACCTCACTGATTTGGCACCCCAGGCTTTACAC...
4	crp-wt	B0	AATTAAGGTGAGTTCGCTCGCTCATGAGGCACCCCAGGCTTTACAC...

To reformat `'raw_df'` into the one provided with MAVE-NN, we first trim the dataframe to keep only rows corresponding to the `'full-wt'` experiment. We then rename each FACS bin `'BX'` to `'ct_X'` for X = 0, 1, ..., 9, and create a `'ct'` column filled with ones. The result is stored in a dataframe called `sub_df`.

Next we use the `pivot()` and `groupby()` functions in Pandas to obtain a dataframe in which the `'seq'` column lists only unique sequences, each of the 10 possible `'ct_X'` values in the original `'bin'` column now label a separate column, and the values in these new columns report the number of times each sequence was observed in each FACS bin. The result is stored in a dataframe called `pivot_df`.

Finally, we create a `'set'` column that randomly assigns each sequence to the training, test, or validation set (using a 60:20:20 split), then reorder the columns for clarity. The resulting dataframe is called `final_df`.

[4]:
```
# Keep only data from the full-wt experiment
ix = raw_df['experiment']=='full-wt'
sub_df = raw_df[ix].copy().reset_index(drop=True)[['bin','x']]

# Rename bins BX -> ct_X, where X = 0, 1, ..., 9
sub_df['bin'] = [f'ct_{s[1:]}' for s in sub_df['bin']]

# Add counts column
sub_df['ct'] = 1
```

(continues on next page)

(continued from previous page)

```
# Pivot dataframe
pivot_df = sub_df.pivot(index='x', values='ct', columns='bin').fillna(0).astype(int)
pivot_df.columns.name = None

# Groupby sequence
pivot_df = pivot_df.groupby('x').sum()

# Reindex dataframe
pivot_df = pivot_df.reset_index()

# Randomly assign sequences to training, validation, and test sets
final_df = pivot_df.copy()
np.random.seed(0)
final_df['set'] = np.random.choice(a=['training','test','validation'],
                                   p=[.6,.2,.2],
                                   size=len(final_df))

# Rearrange columns
new_cols = ['set'] + list(final_df.columns[1:-1]) + ['x']
final_df = final_df[new_cols]

# Save to file (uncomment to execute)
# final_df.to_csv('sortseq_data.csv.gz', index=False, compression='gzip')

# Preview final_df
final_df.head()
```

[4]:

	set	ct_0	ct_1	ct_2	ct_3	ct_4	ct_5	ct_6	ct_7	ct_8	ct_9	\
0	training	0	1	0	0	0	0	0	0	0	0	
1	test	0	0	0	0	0	0	0	0	1	0	
2	test	0	0	0	0	0	0	1	0	0	0	
3	training	0	0	0	0	0	0	0	0	0	1	
4	training	0	0	0	0	0	0	0	0	0	1	

	x
0	AAAAAAAGTGAGTTAGCCAACTAATTAGGCACCGTACGCTTTATAG...
1	AAAAAATCTGAGTTAGCTTACTCATTAGGCACCCCAGGCTTGACAC...
2	AAAAAATCTGAGTTTGCTCACTCTATCGGCACCCCAGTCTTTACAC...
3	AAAAAATGAGAGTTAGTTCACTCATTCGGCACCACAGGCTTTACAA...
4	AAAAAATGGGTGTTAGCTCTATCATTAGGCACCCCCGGCTTTACAC...

This final dataframe, `final_df`, has the same format as the `'sortseq'` dataset that comes with MAVE-NN.

CHAPTER

FOUR

UNDERLYING MATHEMATICS

4.1 Notation

Each MAVE dataset is represented a set of N observations, $(\vec{x}_n, y_n)_{n=1}^{N}$, where each observation consists of a sequence $\vec{x}_n$ and a measurement y_n. Here, y_n can be either a continuous real-valued number, or a categorical variable representing the bin in which the nth sequence was found. Note that, in this representation the same sequence $\vec{x}$ can be observed multiple times, potentially with different values for y due to experimental noise. Datasets with real-valued measurements y are analyzed using GE regression, while datasets with categorical y values are analyzed using MPA regression. Sets of measurements are denoted y_n, etc., and the mean and standard deviations of sets of numbers are denoted by the functions $(\cdot)$ and $(\cdot)$

In the `mavenn.Model` constructor, `L` controls sequence length, `alphabet` controls the set of characters, `Y` controls the number of y-bins (MPA regression only), and `regression_type` controls whether GE or MPA regression is used.

4.2 Genotype-phenotype (G-P) maps

We assume that all sequences have the same length L, and that at each of the L positions in each sequence there is one of C possible characters ($C = 4$ for DNA and RNA; $C = 20$ for protein). In general, MAVE-NN represents sequences using a vector of binary features $\vec{x}$. We adopt the following notation for the individual features in $\vec{x}$:

$$to$$

$$x_{l:c} =$$

$$\begin{cases} 1 & \text{if character } c \text{ occurs at position } l \\ 0 & \text{otherwise,} \end{cases} \tag{4.1}$$

=

$$\begin{cases} 1 & \text{if character } c \text{ occurs at position } l \\ 0 & \text{otherwise,} \end{cases}$$

where $l = 1, 2, \ldots, L$ indexes positions within the sequence, and c indexes the C distinct characters.

We assume that the latent phenotype is given by a function $\phi(\vec{x}; \vec{\theta})$ that depends on a vector of G-P map parameters $\vec{\theta}$. MAVE-NN currently supports four types of G-P maps, all of which can be inferred using either GE regression or MPA regression.

In the `mavenn.Model` constructor, `gpmap_type` controls which type of G-P map is used.

4.2.1 Additive G-P maps

Additive G-P maps assume that each position in $\vec{x}$ contributes independently to the latent phenotype ϕ, and is comptued using,

$$to$$

$$\phi(\vec{x}; \vec{\theta})_{\text{additive}} =$$

$$\theta_0 + \sum_{l=0}^{L-1} \sum_{c} \theta_{l:c} x_{l:c}. \quad (4.1)$$

=

$$\theta_0 + \sum_{l=0}^{L-1} \sum_{c} \theta_{l:c} x_{l:c}.$$

To use an additive G-P map, set `gpmap_type='additive'` in the `mavenn.Model` constructor.

4.2.2 Neighbor G-P maps

Neighbor G-P maps assume that each neighboring pair of positions in $\vec{x}$ contributes to the latent phenotype ϕ, and is comptued using,

$$to$$

$$\phi_{\text{neighbor}}(\vec{x}; \vec{\theta}) =$$

$$\theta_0 + \sum_{l=0}^{L-1} \sum_{c} \theta_{l:c} x_{l:c} + \sum_{l=0}^{L-2} \sum_{c,c'} \theta_{l:c,l+1:c'} x_{l:c} x_{l+1:c'},$$

=

$$\theta_0 + \sum_{l=0}^{L-1} \sum_{c} \theta_{l:c} x_{l:c} + \sum_{l=0}^{L-2} \sum_{c,c'} \theta_{l:c,l+1:c'} x_{l:c} x_{l+1:c'},$$

To use a neighbor G-P map, set `gpmap_type='neighbor'` in the `mavenn.Model` constructor.

4.2.3 Pairwise G-P maps

Pairwise G-P maps assume that every pair of positions in $\vec{x}$ contributes to the latent phenotype ϕ, and is comptued using,

$$to$$

$$\phi_{\text{pairwise}}(\vec{x}; \vec{\theta}) =$$

$$\theta_0 + \sum_{l=0}^{L-1} \sum_{c} \theta_{l:c} x_{l:c} + \sum_{l=0}^{L-2} \sum_{l'=l+1}^{L-1} \sum_{c,c'} \theta_{l:c,l':c'} x_{l:c} x_{l':c'}.$$

=

$$\theta_0 + \sum_{l=0}^{L-1} \sum_{c} \theta_{l:c} x_{l:c} + \sum_{l=0}^{L-2} \sum_{l'=l+1}^{L-1} \sum_{c,c'} \theta_{l:c,l':c'} x_{l:c} x_{l':c'}.$$

To use a pairwise G-P map, set `gpmap_type='pairwise'` in the `mavenn.Model` constructor.

4.2.4 Black box G-P maps

Black box G-P maps are represented as dense feed-forward neural networks, i.e. as multi-layer perceptrons (MLPs). To use a black box G-P map, set `gpmap_type='blackbox'` in the `mavenn.Model` constructor. Note that one can set the number of nodes used in each layer of the ML. For instance, to create an MLP with three hidden layers of size 100, 30, and 10, set `gpmap_kwargs={'hidden_layer_sizes':[100,30,10]}` in the `mavenn.Model` constructor.

4.3 MPA measurement processes

In MPA regression, MAVE-NN directly models the measurement process $p(y|\phi)$. At present, MAVE-NN only supports MPA regression for discrete values of y, which are indexed using nonnegative integers. MAVE-NN takes two forms of input for MPA regression. One is a set of (non-unique) sequence-measurement pairs $(\vec{x}_n, y_n)_{n=0}^{N-1}$, where N is the total number of independent measurements and $y_n \in \{0, 1, \ldots, Y-1\}$, where Y is the total number of bins. The other is a set of (unique) sequence-count-vector pairs $(\vec{x}_m, \vec{c}_m)_{m=0}^{M-1}$, where M is the total number of unique sequences in the data set, and $\vec{c}_m = (c_{m0}, c_{m1}, \ldots, c_{m(Y-1)})$ lists, for each value of y, the number of times c_{my} that the sequence $\vec{x}_m$ was observed in bin y. MPA measurement processes are computed internally using the latter representation via

$$p(y|\phi) = \frac{w_y(\phi)}{\sum_{y'} w_{y'}(\phi)} \tag{4.-1}$$

$$w_y(\phi) = \exp\left[a_y + \sum_{k=0}^{K-1} b_{yk} \tanh(c_{yk}\phi + d_{yk})\right] \tag{4.0}$$

where K is the number of hidden nodes per value of y. The trainable parameters of this measurement process are thus $\vec{\eta} = a_y, b_{yk}, c_{yk}, d_{yk}$.

4.4 GE nonlinearities

GE models assume that each measurement y of a sequence $\vec{x}$ is a nonlinear function $g(\phi)$ plus some noise. In MAVE-NN, this nonlinearity is represented as a sum of $\tanh$ sigmoids, i.e.,

$$g(\phi; \vec{\alpha}) = a + \sum_{k=0}^{K-1} b_k \tanh(c_k\phi + d_k)$$

where K specifies the number of "hidden nodes" contributing to the sum, and $\vec{\alpha} = a, b_k, c_k, d_k$ are trainable parameters. By default, MAVE-NN constrains $g(\phi; \vec{\alpha})$ to be monotonic in ϕ by requiring all $b_k \geq 0$ and $c_k \geq 0$, but this constraint can be relaxed.

In the `mavenn.Model` constructor, `ge_nonlinearity_hidden_nodes` controls the value for K, while `ge_nonlinearity_monotonic` controls the monotonicity constraint on $g(\phi; \vec{\alpha})$.

4.5 GE noise models

MAVE-NN supports three types of GE noise models: Gaussian, Cauchy, and skewed-t. These are specified using the `ge_noise_model` keyword argument in the `mavenn.Model` constructor.

4.5.1 Gaussian nose models

The Gaussian noise model, corresponding to `ge_noise_model='Gaussian'`, is given by

$$to$$

$$p_{\text{gauss}}(y|\hat{y};s) = \frac{1}{\sqrt{2\pi s^2}} \exp\left[-\frac{(y-\hat{y})^2}{2s^2}\right], \quad (4.2)$$

where s denotes the standard deviation. Importantly, MAVE-NN allows this noise model to be heteroskedastic by representing s as an exponentiated polynomial in $\hat{y}$, i.e.,

$$s(\hat{y}) = \exp\left[\sum_{k=0}^{K} a_k \hat{y}^k\right],$$

where K is the order of the polynomial and a_k are trainable parameters. The user has the option to set K, and using $K = 0$ renders this noise model homoskedastic. Quantiles are computed using $y_q = \hat{y} + s\sqrt{2}, \text{erf}^{-1}(2q-1)$ for user-specified values of :math:`q $\in [0, 1]$.

4.5.2 Cauchy noise models

The Cauchy noise model, corresponding to `ge_noise_model='Cauchy'`, is given by

$$to$$

$$p_{\text{cauchy}}(y|\hat{y};s) = \left[\pi s\left(1 + \frac{(y-\hat{y})^2}{s^2}\right)\right]^{-1} \quad (4.4)$$

where the scale parameter s is an exponentiated K'th order polynomial in $\hat{y}$. Quantiles are computed using $y_q = \hat{y} + s\tan[\pi(q-\frac{1}{2})]$.

4.5.3 Skewed-t noise models

The skewed-t noise model, corresponding to `ge_noise_model='SkewedT'`, is of the form described by Jones & Faddy (2003), and is given by

$$p_{\text{skewt}}(y|\hat{y}; s, a, b) = s^{-1} f(t; a, b),$$

where

$$t = t^* + \frac{y - \hat{y}}{s}. \quad t^* = \frac{(a-b)\sqrt{a+b}}{\sqrt{2a+1}\sqrt{2b+1}},$$

and

$$to$$

$$f(t; a, b) =$$

$$\frac{2^{1-a-b}}{\sqrt{a+b}} \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \left[1 + \frac{t}{\sqrt{a+b+t^2}}\right]^{a+\frac{1}{2}} \left[1 - \frac{t}{\sqrt{a+b+t^2}}\right]^{b+\frac{1}{2}} . \quad (4.7)$$

=

$$\frac{2^{1-a-b}}{\sqrt{a+b}} \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \left[1 + \frac{t}{\sqrt{a+b+t^2}}\right]^{a+\frac{1}{2}} \left[1 - \frac{t}{\sqrt{a+b+t^2}}\right]^{b+\frac{1}{2}} .$$

Note that the t statistic here is an affine function of y chosen so that the distribution's mode (corresponding to t^*) is positioned at $\hat{y}$. The three parameters of this model, s, a, b, are each represented using K-th order exponentiated polynomial with trainable coefficients. Quantiles are computed using

$$y_q = \hat{y} + (t_q - t^*)s,$$

where

$$t_q = \frac{(2x_q - 1)\sqrt{a+b}}{\sqrt{1-(2x_q-1)^2}}, \quad x_q = I_q^{-1}(a,b),$$

and I^{-1} denotes the inverse of the regularized incomplete Beta function $I_x(a,b)$.

4.6 Gauge modes and diffemorphic modes

These G-P maps also have non-identifiable degrees of freedom that must be "fixed", i.e. pinned down, before the values of individual parameters can be meaningfully interpreted. These degrees of freedom come in two flavors: gauge modes and diffeomorphic modes. Gauge modes are changes to $\vec{\theta}$ that do not alter the values of the latent phenotype ϕ. Diffeomorphic modes (Kinney & Atwal, 2014; Atwal & Kinney, 2016) are changes to $\vec{\theta}$ that do alter ϕ, but do so in ways that can be undone by transformations of the measurement process $p(y|\phi)$ along corresponding "dual modes". As shown in (Kinney & Atwal, 2014), the diffeomorphic modes of linear G-P maps like those considered here will in general correspond to affine transformations of ϕ (though there are exceptions at special values of $\vec{\theta}$). MAVE-NN fixes both gauge modes and diffeomorphic modes before providing parameter values or other access to model internals to the user, e.g. when `model.get_theta()` is called.

4.6.1 Fixing diffeomorphic modes

Diffeomorphic modes of G-P maps are fit by transforming G-P map parameters $\vec{\theta}$ via

$$\theta_0 \to \theta_0 - a$$

and

$$to$$

$$\vec{\theta} \to \vec{\theta}/b \quad (4.10)$$

where $a = (\phi_n)$ and $b = (\phi_n)$ are the mean and standard deviation of ϕ values computed on the training data. This produces a corresponding change in latent phenotype values $\phi \to (\phi - a)/b$. To avoid altering likelihood contributions

$p(y|x)$, MAVE-NN further transforms the measurement process $p(y|\phi)$ along dual modes in order to compensate. In GE regression this is done by adjusting the GE nonlinearity while keeping the noise model $p(y|\hat{y})$ fixed,

$$to$$

$$g(\phi) \to g(a + b\phi), \quad (4.11)$$

while in MPA regression MAVE-NN adjusts the full measurement process,

$$to$$

$$p(y|\phi) \to p(y|a + b\phi). \quad (4.12)$$

4.6.2 Fixing the gauge

Gauge modes are fit using a "hierarchical gauge", in which lower-order terms in $\phi(\vec{x}; \vec{\theta})$ account for the highest possible fraction of variance in ϕ. Such gauges further require a probability distribution on sequence space, which is used to compute these variances. MAVE-NN assumes that such distributions factorize by position, and can thus be represented by a probability matrix with elements $p_{l:c}$, representing the probability of character c at position l. MAVE-NN then transforms G-P map parameters via,

$$to$$

$$\theta_0 \to$$

$$\theta_0 + \sum_l \sum_{c'} \theta_{l:c'}\, p_{l:c'} + \sum_l \sum_{l'>l} \sum_{c,c'} \theta_{l:c,l':c'}\, p_{l:c}\, p_{l':c'},$$

$$\theta_0 + \sum_l \sum_{c'} \theta_{l:c'}\, p_{l:c'} + \sum_l \sum_{l'>l} \sum_{c,c'} \theta_{l:c,l':c'}\, p_{l:c}\, p_{l':c'},$$

$$to$$

$$\theta_{l:c} \rightarrow$$

$$\theta_{l:c} - \sum_{c'} \theta_{l:c'}\, p_{l:c}$$

$$+ \sum_{l'>l} \sum_{c'} \theta_{l:c,l':c'}\, p_{l':c'} + \sum_{l'<l} \sum_{c'} \theta_{l':c',l:c}\, p_{l':c'}$$

$$- \sum_{l'>l} \sum_{c',c''} \theta_{l:c',l':c''}\, p_{l:c'}\, p_{l':c''}, - \sum_{l'<l} \sum_{c',c''} \theta_{l':c'',l:c'}\, p_{l:c'}\, p_{l':c''},$$

$$\theta_{l:c} - \sum_{c'} \theta_{l:c'}\, p_{l:c}$$
$$+ \sum_{l'>l} \sum_{c'} \theta_{l:c,l':c'}\, p_{l':c'} + \sum_{l'<l} \sum_{c'} \theta_{l':c',l:c}\, p_{l':c'}$$
$$- \sum_{l'>l} \sum_{c',c''} \theta_{l:c',l':c''}\, p_{l:c'}\, p_{l':c''}, - \sum_{l'<l} \sum_{c',c''} \theta_{l':c'',l:c'}\, p_{l:c'}\, p_{l':c''},$$

and

$$to$$

$$\theta_{l:c,l':c'} \rightarrow$$

$$\theta_{l:c,l':c'} - \sum_{c''} \theta_{l:c'',l':c'}\, p_{l:c''} - \sum_{c''} \theta_{l:c,l':c''}\, p_{l':c''} + \sum_{c'',c'''} \theta_{l:c'',l':c'''}\, p_{l:c''}\, p_{l':c'''}.$$

$$\theta_{l:c,l':c'} - \sum_{c''} \theta_{l:c'',l':c'}\, p_{l:c''} - \sum_{c''} \theta_{l:c,l':c''}\, p_{l':c''} + \sum_{c'',c'''} \theta_{l:c'',l':c'''}\, p_{l:c''}\, p_{l':c'''}.$$

The gauge-fixing transformations for the both the additive and neighbor models follow the same rules, but with $\theta_{l:c,l':c'} = 0$ for all l, l' (additive model) or whenever $l' \neq l + 1$ (neighbor model).

4.7 Metrics

4.7.1 Loss function

Let $\vec{\theta}$ denote the parameters of G-P map, and $\vec{\eta}$ denote the parameters of the measurement process. MAVE-NN optimizations these parameters using stochastic gradient descent (SGD) on a loss function given by

$$\mathcal{L} = \mathcal{L}_{+}\mathcal{L}_{\text{reg}}$$

where $\mathcal{L}$ is the negative log likelihood of the model, given by

$$to$$

$$\mathcal{L}_{[}\vec{\theta}, \vec{\eta}] = -\sum_{n=0}^{N-1} \log\left[p(y_n|\phi_n; \vec{\eta})\right], \ \phi_n = \phi(\vec{x}_n; \vec{\theta}), \quad (4.11)$$

and $\mathcal{L}_{\text{reg}}$ provides for regularization of the model parameters.

In the context of GE regression, we can write $\vec{\eta} = (\vec{\alpha}, \vec{\beta})$ where $\vec{\alpha}$ represents the parameters of the GE nonlinearity $g(\phi)$, and $\vec{\beta}$ denotes the parameters of the noise model $p(y|\hat{y})$. The likelihood contributions from each observation n then becomes $p(y_n|\phi_n; \vec{\eta}) = p(y_n|\hat{y}_n; \vec{\beta})$ where $\hat{y}_n = g(\phi_n; \vec{\alpha})$. In the context of MPA regression with a dataset of the form $(\vec{x}_m, \vec{c}_m)_{m=1}^{M}$, the loss function simplifies to

$$to$$

$$\mathcal{L}_{[}\vec{\theta}, \vec{\eta}] =$$

$$\sum_{m=0}^{M-1} \sum_{y=0}^{Y-1} c_{my} \log[p(y|\phi_m; \vec{\eta})], \quad (4.12)$$

=

$$\sum_{m=0}^{M-1}\sum_{y=0}^{Y-1} c_{my}\log[p(y|\phi_m;\vec{\eta})],$$

where $\phi_m = \phi(x_m;\vec{\theta})$. For the regularization term, MAVE-NN uses an L_2 penalty of the form

$$\mathcal{L}_{\text{reg}}[\vec{\theta},\vec{\eta}] = \lambda_\theta|\vec{\theta}|^2 + \lambda_\eta|\vec{\eta}|^2,$$

where λ_θ and λ_η respectively control the strength of regularization for the G-P map and measurement process parameters.

4.7.2 Variational information

It is sometimes useful to consider a quantity we call the "variational information", $I_[y;\phi]$, which is an affine transformation of $\mathcal{L}$:

$$I_[y;\phi] = H[y] - \frac{\log_2(e)}{N}\mathcal{L}.$$

Here $H[y]$ is the entropy of the data y_n, which is estimated using the k'th nearest neighbor (kNN) estimator from the NPEET package (Ver Steeg, 2014). Noting that this quantity can also be written as $I_[y;\phi] = H[y] - (Q_n)$ where $Q_n = -\log_2 p(y_n|\phi_n)$, we estimate the associated uncertainty using

$$\delta I_[y;\phi] = \sqrt{\delta H[y]^2 + (Q_n)/N}.$$

Variational information quantifies the mutual information between ϕ and y assuming that the inferred measurement process $p(y|\phi)$ is correct. We therefore propose $I_\pm\delta I$ as a diagnostic for model fitting and performance, as it is directly comparable to the predictive information of the model, $I_[y;\phi]$, and can be computed during model training without any costly entropy estimation steps.

It is worth noting that $I_[y;\phi]$ provides an approximate lower bound to $I_[y;\phi]$. In the $N_\to\infty$ limit,

$$
\begin{aligned}
& to \\
I_[y;\phi] &= \\
H[y] - H[y|\phi] &= \\
H[y] + \log_2 p_(y|\phi) &= \\
H[y] + \log_2 p_(y|\phi) {}_+ \log_2 \frac{p_(y|\phi)}{p_(y|\phi)} &= \\
H[y] - \frac{\log_2(e)}{N\mathcal{L}_+ D_{KL}(p_|p_)} &= \\
I_[y;\phi] + D_{\mathrm{KL}}(p_|p_) &\geq \\
I_[y;\phi], & \quad (4.15)
\end{aligned}
$$

$H[y] - H[y|\phi]$
$H[y] + \log_2 p_(y|\phi)$
$H[y] + \log_2 p_(y|\phi) {}_+ \log_2 \frac{p_(y|\phi)}{p_(y|\phi)}$
$H[y] - \frac{\log_2(e)}{N\mathcal{L}_+ D_{KL}(p_|p_)}$
$I_[y;\phi] + D_{\mathrm{KL}}(p_|p_)$
$I_[y;\phi]$,

where D_{KL} denotes the Kullback-Leibler divergence and $\cdot$ indicates averaging over $p_(y,\phi)$. When N is finite this lower bound is only approximate, of course, but the uncertainties in these information quantities are often quite small due to the large size of typical MAVE datasets. Moreover the difference between these quantities,

,

$I_[y;\phi] - I_[y;\phi] \approx D_{\mathrm{KL}}(p_|p_)$

quantifies how much p deviates from p, and can thus be used as a diagnostic for how accurate the assumed form of the measurement process is.

4.7.3 Predictive information

The predictive information $I_{[}y;\phi]$, when computed on data not used for training, provides a measure of how strongly a G-P map predicts experimental measurements irrespective of the corresponding measurement process. To estimate this quantity we use k'th nearest neighbor (kNN) estimators of entropy and mutual information adapted from the NPEET Python package (VanSteeg, 2014). Here, the user has the option of adjusting k, which controls a variance/bias tradeoff. When y is discrete (MPA regression), $I_{[}y;\phi]$ is computed using the classic kNN entropy estimator (Vasicek, 1976; Kraskov et al., 2004) via the decomposition $I_{[}y;\phi] = H[\phi] - \sum_y p(y) H_y[\phi]$, where $H_y[\phi]$ denotes the entropy of $p(\phi|y)$ for fixed y. When y is continuous (GE regression), $I_{[}y;\phi]$ is estimated using the kNN-based KSG algorithm (Kraskov et al, 2004). This approach optionally supports a local nonuniformity correction of (Gao et al., 2014), which is important when y and ϕ exhibit strong dependencies, but which also requires substantially more time to compute. We emphasize that both estimates of $I_{[}y;\phi]$ make no use of the measurement process $p(y|\phi)$ inferred by MAVE-NN.

4.7.4 Uncertainties in kNN estimates

MAVE-NN quantifies uncertainties in $H[y]$ and $I[y;\phi]$ using multiple random samples of half the data. Let $\mathcal{D}_{100\%}$ denote a full dataset, and let $\mathcal{D}_{50\%,r}$ denote a 50% subsample (indexed by r) of this dataset. Given an estimator $E(\cdot)$ of either entropy or mutual information, as well as the number of subsamples R to use, the uncertainty in $E(\mathcal{D}_{100\%})$ is estimated as

$$\delta E(\mathcal{D}_{100\%}) = \frac{1}{\sqrt{2}} \left[E(\mathcal{D}_{50\%,r})_{r=0}^{R-1} \right].$$

By default MAVE-NN uses $R = 25$. We note that computing such uncertainty estimates substantially increase computation time, as $E(\cdot)$ needs to be evaluated $R + 1$ times instead of just once. We also note that boostrap resampling is not advisable in this context, as it systematically underestimates $H[y]$ and overestimates $I[y;z]$ (data not shown).

4.8 References

1. Jones M, Faddy M (2003). A skew extension of the t-distribution, with applications. J Roy Stat Soc B Met. 65(1):159-174.
2. Kinney J, Atwal G (2014). Parametric Inference in the Large Data Limit Using Maximally Informative Models. Neural Comput 26(4):637-653.
3. Atwal G, Kinney J (2016). Learning Quantitative Sequence–Function Relationships from Massively Parallel Experiments. J Stat Phys. 162(5):1203-1243.
4. Ver Steeg G (2014). Non-Parametric Entropy Estimation Toolbox (NPEET). https://github.com/gregversteeg/NPEET
5. Kraskov A, Stögbauer H, Grassberger P (2004). Estimating mutual information. Phys Rev E. 69(6):066138.
6. Gao S, Steeg G, Galstyan A (2015). Efficient Estimation of Mutual Information for Strongly Dependent Variables. PMLR 38:277-286

CHAPTER

FIVE

API REFERENCE

5.1 Tests

A suite of automated tests are provided to ensure proper software installation and execution.

mavenn.**run_tests**()
Run all MAVE-NN functional tests.

5.2 Examples

A variety of real-world datasets, pre-trained models, analysis demos, and tutorials can be accessed using the following functions.

mavenn.**load_example_dataset**(*name=None*)
Load example dataset provided with MAVE-NN.

Parameters

name: (str) Name of example dataset. If `None`, a list of valid dataset names will be printed.

Returns

data_df: (pd.DataFrame) Dataframe containing the example datase.

mavenn.**load_example_model**(*name=None*)
Load an example model already inferred by MAVE-NN.

Parameters

name: (str, None) Name of model to load. If `None`, a list of valid model names will be printed.

Returns

model: (mavenn.Model) A pre-trained Model object.

mavenn.**run_demo**(*name=None*, *print_code=False*, *print_names=True*)
Perform demonstration of MAVE-NN.

Parameters

name: (str, None) Name of demo to run. If `None`, a list of valid demo names will be returned.

print_code: (bool) If `True`, the text of the demo file will be printed along with the output from running this file. If `False`, only the demo output will be shown.

print_names: (bool) If True and `name=None`, the names of all demos will be printed.

Returns

demo_names: (list, None) List of demo names, returned if user passes `names=None`. Otherwise None.

`mavenn.list_tutorials()`
Reveal local directory where MAVE-NN tutorials are stored, as well as the names of available tutorial notebook files.

5.3 Load

MAVE-NN allows users to save and load trained models.

`mavenn.load(`*filename*, *verbose=True*`)`
Load a previously saved model.

Saved models are represented by two files having the same root and two different extensions, `.pickle` and `.h5`. The `.pickle` file contains model metadata, including all information needed to reconstruct the model's architecture. The `.h5` file contains the values of the trained neural network weights.

Parameters

filename: (str) File directory and root. Do not include extensions.

verbose: (bool) Whether to print feedback.

Returns

loaded_model: (mavenn.Model) MAVE-NN model object.

5.4 Visualization

MAVE-NN provides the following two methods to facilitate the visualization of inferred genotype-phenotype maps.

`mavenn.heatmap(`*values*, *alphabet*, *seq=None*, *seq_kwargs=None*, *ax=None*, *show_spines=False*, *cbar=True*, *cax=None*, *clim=None*, *clim_quantile=1*, *ccenter=None*, *cmap='coolwarm'*, *cmap_size='5%'*, *cmap_pad=0.1*`)`
Draw a heatmap illustrating an `L` x `C` matrix of values, where `L` is sequence length and `C` is the alphabet size.

Parameters

values: (np.ndarray) Array of shape `(L,C)` that contains values to plot.

alphabet: (str, np.ndarray) Alphabet name `'dna'`, `'rna'`, or `'protein'`, or 1D array containing characters in the alphabet.

seq: (str, None) The sequence to show, if any, using dots plotted on top of the heatmap. Must have length `L` and be comprised of characters in `alphabet`.

seq_kwargs: (dict) Arguments to pass to `Axes.scatter()` when drawing dots to illustrate the characters in `seq`.

ax: (matplotlib.axes.Axes) The `Axes` object on which the heatmap will be drawn. If `None`, one will be created. If specified, `cbar=True`, and `cax=None`, `ax` will be split in two to make room for a colorbar.

show_spines: (bool) Whether to show spines around the edges of the heatmap.

cbar: (bool) Whether to draw a colorbar next to the heatmap.

cax: (matplotlib.axes.Axes, None) The `Axes` object on which the colorbar will be drawn, if requested. If `None`, one will be created by splitting `ax` in two according to `cmap_size` and `cmap_pad`.

clim: (list, None) List of the form `[cmin, cmax]`, specifying the maximum `cmax` and minimum `cmin` values spanned by the colormap. Overrides `clim_quantile`.

clim_quantile: (float) Must be a float in the range [0,1]. `clim` will be automatically chosen to include this central quantile of values.

ccenter: (float) Value at which to position the center of a diverging colormap. Setting `ccenter=0` often makes sense.

cmap: (str, matplotlib.colors.Colormap) Colormap to use.

cmap_size: (str) Fraction of `ax` width to be used for the colorbar. For formatting requirements, see the documentation for `mpl_toolkits.axes_grid1.make_axes_locatable()`.

cmap_pad: (float) Space between colorbar and the shrunken heatmap `Axes`. For formatting requirements, see the documentation for `mpl_toolkits.axes_grid1.make_axes_locatable()`.

Returns

ax: (matplotlib.axes.Axes) `Axes` object containing the heatmap.

cb: (matplotlib.colorbar.Colorbar, None) Colorbar object linked to `ax`, or `None` if no colorbar was drawn.

mavenn.**heatmap_pairwise**(*values*, *alphabet*, *seq=None*, *seq_kwargs=None*, *ax=None*, *gpmap_type='pairwise'*, *show_position=False*, *position_size=None*, *position_pad=1*, *show_alphabet=True*, *alphabet_size=None*, *alphabet_pad=1*, *show_seplines=True*, *sepline_kwargs=None*, *xlim_pad=0.1*, *ylim_pad=0.1*, *cbar=True*, *cax=None*, *clim=None*, *clim_quantile=1*, *ccenter=0*, *cmap='coolwarm'*, *cmap_size='5%'*, *cmap_pad=0.1*)

Draw a heatmap illustrating pairwise or neighbor values, e.g. representing model parameters, mutational effects, etc.

Note: The resulting plot has aspect ratio of 1 and is scaled so that pixels have half-diagonal lengths given by `half_pixel_diag = 1/(C*2)`, and blocks of characters have half-diagonal lengths given by `half_block_diag = 1/2`. This is done so that the horizontal distance between positions (as indicated by x-ticks) is 1.

Parameters

values: (np.array) An array, shape `(L,C,L,C)`, containing pairwise or neighbor values. Note that only values at coordinates `[l1, c1, l2, c2]` with `l2 > l1` will be plotted. NaN values will not be plotted.

alphabet: (str, np.ndarray) Alphabet name `'dna'`, `'rna'`, or `'protein'`, or 1D array containing characters in the alphabet.

seq: (str, None) The sequence to show, if any, using dots plotted on top of the heatmap. Must have length `L` and be comprised of characters in `alphabet`.

seq_kwargs: (dict) Arguments to pass to `Axes.scatter()` when drawing dots to illustrate the characters in `seq`.

ax: (matplotlib.axes.Axes) The `Axes` object on which the heatmap will be drawn. If `None`, one will be created. If specified, `cbar=True`, and `cax=None`, `ax` will be split in two to make room for a colorbar.

gpmap_type: (str) Determines how many pairwise parameters are plotted. Must be `'pairwise'` or `'neighbor'`. If `'pairwise'`, a triangular heatmap will be plotted. If `'neighbor'`, a heatmap resembling a string of diamonds will be plotted.

show_position: (bool) Whether to annotate the heatmap with position labels.

position_size: (float) Font size to use for position labels. Must be >= 0.

position_pad: (float) Additional padding, in units of `half_pixel_diag`, used to space the position labels further from the heatmap.

show_alphabet: (bool) Whether to annotate the heatmap with character labels.

alphabet_size: (float) Font size to use for alphabet. Must be >= 0.

alphabet_pad: (float) Additional padding, in units of `half_pixel_diag`, used to space the alphabet labels from the heatmap.

show_seplines: (bool) Whether to draw lines separating character blocks for different position pairs.

sepline_kwargs: (dict) Keywords to pass to `Axes.plot()` when drawing seplines.

xlim_pad: (float) Additional padding to add (in absolute units) both left and right of the heatmap.

ylim_pad: (float) Additional padding to add (in absolute units) both above and below the heatmap.

cbar: (bool) Whether to draw a colorbar next to the heatmap.

cax: (matplotlib.axes.Axes, None) The `Axes` object on which the colorbar will be drawn, if requested. If `None`, one will be created by splitting `ax` in two according to `cmap_size` and `cmap_pad`.

clim: (list, None) List of the form `[cmin, cmax]`, specifying the maximum `cmax` and minimum `cmin` values spanned by the colormap. Overrides `clim_quantile`.

clim_quantile: (float) Must be a float in the range [0,1]. `clim` will be automatically chosen to include this central quantile of values.

ccenter: (float) Value at which to position the center of a diverging colormap. Setting `ccenter=0` often makes sense.

cmap: (str, matplotlib.colors.Colormap) Colormap to use.

cmap_size: (str) Fraction of `ax` width to be used for the colorbar. For formatting requirements, see the documentation for `mpl_toolkits.axes_grid1.make_axes_locatable()`.

cmap_pad: (float) Space between colorbar and the shrunken heatmap `Axes`. For formatting requirements, see the documentation for `mpl_toolkits.axes_grid1.make_axes_locatable()`.

Returns

ax: (matplotlib.axes.Axes) `Axes` object containing the heatmap.

cb: (matplotlib.colorbar.Colorbar, None) Colorbar object linked to `ax`, or `None` if no colorbar was drawn.

5.5 Models

The `mavenn.Model` class represents all neural-network-based models inferred by MAVE-NN. A variety of class methods make it easy to,

- define models,
- fit models to data,
- access model parameters and metadata,
- save models,
- evaluate models on new data.

In particular, these methods allow users to train and analyze models without prior knowledge of TensorFlow 2, the deep learning framework used by MAVE-NN as a backend.

class mavenn.**Model**(*L*, *alphabet*, *regression_type*, *gpmap_type='additive'*, *gpmap_kwargs={}*, *Y=2*, *ge_nonlinearity_type='nonlinear'*, *ge_nonlinearity_monotonic=True*, *ge_nonlinearity_hidden_nodes=50*, *ge_noise_model_type='Gaussian'*, *ge_heteroskedasticity_order=0*, *normalize_phi=True*, *mpa_hidden_nodes=50*, *theta_regularization=0.001*, *eta_regularization=0.1*, *ohe_batch_size=50000*, *custom_gpmap=None*, *initial_weights=None*)

Represents a MAVE-NN model, which includes a genotype-phenotype (G-P) map as well as a measurement process. For global epistasis (GE) regression, set `regression_type='GE'`; for measurement process agnostic (MPA) regression, set `regression_type='MPA'`.

Parameters

L: (int) Length of each training sequence. Must be >= 1.

alphabet: (str, np.ndarray) Either the alphabet name (`'dna'`, `'rna'`, or `'protein'`) or a 1D array of characters to be used as the alphabet.

regression_type: (str) Type of regression implemented by the model. Choices are `'GE'` (for a global epistasis model) and `'MPA'` (for a measurement process agnostic model).

gpmap_type: (str) Type of G-P map to infer. Choices are `'additive'`, `'neighbor'`, `'pairwise'`, and `'blackbox'`.

gpmap_kwargs: (dict) Additional keyword arguments used for specifying the G-P map.

Y: (int) The number if discrete y bins to use when defining an MPA model. Must be >= 2. Has no effect on MPA models.

ge_nonlinearity_monotonic: (boolean) Whether to enforce a monotonicity constraint on the GE nonlinearity. Has no effect on MPA models.

ge_nonlinearity_hidden_nodes: (int) Number of hidden nodes (i.e. sigmoidal contributions) to use when defining the nonlinearity component of a GE model. Has no effect on MPA models.

ge_noise_model_type: (str) Noise model to use for when defining a GE model. Choices are `'Gaussian'`, `'Cauchy'`, `'SkewedT'`, or `'Empirical'`. Has no effect on MPA models.

ge_heteroskedasticity_order: (int) In the GE model context, this represents the order of the polynomial(s) used to define noise model parameters as functions of `yhat`. The larger this is, the more heteroskedastic an inferred noise model is likely to be. Set to `0` to enforce a homoskedastic noise model. Has no effect on MPA models. Must be >= `0`.

normalize_phi: (bool) Whether to fix diffeomorphic modes after model training.

mpa_hidden_nodes: Number of hidden nodes (i.e. sigmoidal contributions) to use when defining the MPA measurement process. Must be >= 1.

theta_regularization: (float) L2 regularization strength for G-P map parameters `theta`. Must be >= 0; use 0 for no regularization.

eta_regularization: (float) L2 regularization strength for measurement process parameters `eta`. Must be >= 0; use 0 for no regularization.

ohe_batch_size: (int) DISABLED. How many sequences to one-hot encode at a time when calling `Model.set_data()`. Typically, the larger this number is the quicker the encoding will happen. A number too large, however, may cause the computer's memory to run out. Must be >= 1.

custom_gpmap: (GPMapLayer sub-class) Defines custom gpmap, provided by user. Inherited class of GP-MAP layer, which defines the functionality for x_to_phi_layer.

initial_weights: (np.array) Numpy array of weights that gets set as initial weights of a model if not set to None.

Methods

I_predictive(x, y[, ct, knn, knn_fuzz, ...])	Estimate predictive information.
I_variational(x, y[, ct, knn_fuzz, uncertainty])	Estimate variational information.
bootstrap(data_df[, num_models, verbose, ...])	Sample plausible models using parametric bootstrapping.
fit([epochs, learning_rate, ...])	Infer values for model parameters.
get_nn()	Return the underlying TensorFlow neural network.
get_theta([gauge, p_lc, x_wt, unobserved_value])	Return parameters of the G-P map.
p_of_y_given_phi(y, phi[, paired])	Compute probabilities p(`y` | `phi`).
p_of_y_given_x(y, x[, paired])	Compute probabilities p(`y` | `x`).
p_of_y_given_yhat(y, yhat[, paired])	Compute probabilities p(`y` | `yhat`); GE models only.
phi_to_yhat(phi)	Compute `phi` given `yhat`; GE models only.
save(filename[, verbose])	Save model.
set_data(x, y[, dy, ct, validation_frac, ...])	Set training data.
simulate_dataset(template_df)	Generate a simulated dataset.
x_to_phi(x)	Compute `phi` given `x`.
x_to_yhat(x)	Compute `yhat` given `x`.
yhat_to_yq(yhat[, q, paired])	Compute quantiles of p(`y` | `yhat`); GE models only.

I_predictive(*x*, *y*, *ct=None*, *knn=5*, *knn_fuzz=0.01*, *uncertainty=True*, *num_subsamples=25*, *use_LNC=False*, *alpha_LNC=0.5*, *verbose=False*)

Estimate predictive information.

Predictive information, `I_pred`, is the mutual information I[`phi` ; `y`] between latent phenotypes `phi` and measurements `y`. Unlike variational information, `I_pred` does not assume that the inferred measurement process p(`y` | `phi`) is correct. `I_pred` is estimated using the k'th nearest neighbor methods from the NPEET package.

Parameters

x: (np.ndarray) 1D array of `N` sequences, each of length `L`.

y: (np.ndarray) Array of measurements. For GE models, `y` must be a 1D array of `N` floats. For MPA models, `y` must be either a 1D or 2D array of nonnegative ints. If 1D, `y` must be of length `N`, and will be interpreted as listing bin numbers, i.e. `0` , `1` , ..., `Y-1`. If 2D, `y`

must be of shape `(N,Y)`, and will be interpreted as listing the observed counts for each of the `N` sequences in each of the `Y` bins.

ct: (np.ndarray, None) Only used for MPA models when `y` is 1D. In this case, `ct` must be a 1D array, length `N`, of nonnegative integers, and represents the number of observations of each sequence in each bin. Use `y=None` for GE models, as well as for MPA models when `y` is 2D.

knn: (int>0) Number of nearest neighbors to use in the entropy estimators from the NPEET package.

knn_fuzz: (float>0) Amount of noise to add to `phi` values before passing them to the KNN estimators. Specifically, Gaussian noise with standard deviation `knn_fuzz * np.std(phi)` is added to `phi` values. This is a hack and is not ideal, but is needed to get the KNN estimates to behave well on real MAVE data.

uncertainty: (bool) Whether to estimate the uncertainty in `I_pred`. Substantially increases runtime if `True`.

num_subsamples: (int) Number of subsamples to use when estimating the uncertainty in `I_pred`.

use_LNC: (bool) Whether to use the Local Nonuniform Correction (LNC) of Gao et al., 2015 when computing `I_pred` for GE models. Substantially increases runtime set to `True`.

alpha_LNC: (float in (0,1)) Value of `alpha` to use when computing the LNC correction. See Gao et al., 2015 for details. Used only for GE models.

verbose: (bool) Whether to print results and execution time.

Returns

I_pred: (float) Estimated variational information, in bits.

dI_pred: (float) Standard error for `I_pred`. Is `0` if `uncertainty=False` is used.

`I_variational`(*x*, *y*, *ct=None*, *knn_fuzz=0.01*, *uncertainty=True*)

Estimate variational information.

Likelihood information, `I_var`, is the mutual information I[`phi` ; `y`] between latent phenotypes `phi` and measurements `y` under the assumption that the inferred measurement process p(`y` | `phi`) is correct. `I_var` is an affine transformation of log likelihood and thus provides a useful metric during model training. When evaluated on test data, `I_var` also provides a lower bound to the predictive information `I_pred`, which does not assume that the inferred measurement process is correct. The difference `I_pred - I_var` thus quantifies the mismatch between the inferred measurement process and the true conditional distribution p(`y` | `phi`).

Parameters

x: (np.ndarray) 1D array of `N` sequences, each of length `L`.

y: (np.ndarray) Array of measurements. For GE models, `y` must be a 1D array of `N` floats. For MPA models, `y` must be either a 1D or 2D array of nonnegative ints. If 1D, `y` must be of length `N`, and will be interpreted as listing bin numbers, i.e. `0` , `1` , ..., `Y-1`. If 2D, `y` must be of shape `(N,Y)`, and will be interpreted as listing the observed counts for each of the `N` sequences in each of the `Y` bins.

ct: (np.ndarray, None) Only used for MPA models when `y` is 1D. In this case, `ct` must be a 1D array, length `N`, of nonnegative integers, and represents the number of observations of each sequence in each bin. Use `y=None` for GE models, as well as for MPA models when `y` is 2D.

knn_fuzz: (float>0) Amount of noise to add to `y` values before passing them to the KNN estimators. Specifically, Gaussian noise with standard deviation `knn_fuzz * np.std(y)` is added to `y` values. This is a hack and is not ideal, but is needed to get the KNN estimates to behave well on real MAVE data. Only used for GE regression models.

uncertainty: (bool) Whether to estimate the uncertainty of `I_var`.

Returns

I_var: (float) Estimated variational information, in bits.

dI_var: (float) Standard error for `I_var`. Is `0` if `uncertainty=False` is used.

bootstrap(*data_df*, *num_models=10*, *verbose=True*, *initialize_from_self=False*, *fit_kwargs={}*)
Sample plausible models using parametric bootstrapping.

Given a copy `data_df` of the initial dataset used to train/test the model, this function first simulates `num_models` datasets, each of which has the same sequences and corresponding training, validation, and test set designations as `data_df`, but simulated measurement values (either `y` column or `ct_#` column values) generated using `self`. One model having the same form as `self` is then fit to each dataset, and the list of resulting models in returned to the user.

Parameters

data_df: (str) The dataset used to fit the original model (i.e., `self`). Must have a column `'x'` listing sequences, as well as a column `'set'` whose entries are `'training'`, `'validation'`, or `'test'`.

num_models: (int > 0) Number of models to return.

verbose: (bool) Whether to print feedback.

initialize_from_self: (bool) Whether to initiate each bootstrapped model from the inferred parameters of `self`. WARNING: using this option can cause systematic underestimation of parameter uncertainty.

fit_kwargs: (dict) Dictionary of keyword arguments. Entries will override the keyword arguments that were passed to `self.fit()` during initial model training, and which are used by default for training the simulation-inferred model here.

Returns

models: (list) List of `mavenn.Model` objects.

fit(*epochs=50*, *learning_rate=0.005*, *validation_split=0.2*, *verbose=True*, *early_stopping=True*, *early_stopping_patience=20*, *batch_size=50*, *linear_initialization=True*, *freeze_theta=False*, *callbacks=[]*, *try_tqdm=True*, *optimizer='Adam'*, *optimizer_kwargs={}*, *fit_kwargs={}*)
Infer values for model parameters.

Uses training algorithms from TensorFlow to learn model parameters. Before this is run, the training data must be set using `Model.set_data()`.

Parameters

epochs: (int) Maximum number of epochs to complete during model training. Must be >= `0`.

learning_rate: (float) Learning rate. Must be > `0.`

validation_split: (float in [0,1]) Fraction of training data to reserve for validation.

verbose: (boolean) Whether to show progress during training.

early_stopping: (bool) Whether to use early stopping.

early_stopping_patience: (int) Number of epochs to wait, after a minimum value of validation loss is observed, before terminating the model training process.

batch_size: (None, int) Batch size to use for stochastic gradient descent and related algorithms. If None, a full-sized batch is used. Note that the negative log likelihood loss function used by MAVE-NN is extrinsic in batch_size.

linear_initialization: (bool) Whether to initialize the results of a linear regression computation. Has no effect when `gpmap_type='blackbox'`.

freeze_theta: (bool) Whether to set the weights of the G-P map layer to be non-trainable. Note that setting `linear_initialization=True` and `freeze_theta=True` will set theta to be initialized at the linear regression solution and then become frozen during training.

callbacks: (list) Optional list of `tf.keras.callbacks.Callback` objects to use during training.

try_tqdm: (bool) If true, mavenn will attempt to load the package *tqdm* and append *TqdmCallback(verbose=0)* to the *callbacks* list in order to improve the visual display of training progress. If users do not have tqdm installed, this will do nothing.

optimizer: (str) Optimizer to use for training. Valid options include: `'SGD'`, `'RMSprop'`, `'Adam'`, `'Adadelta'`, `'Adagrad'`, `'Adamax'`, `'Nadam'`, `'Ftrl'`.

optimizer_kwargs: (dict) Additional keyword arguments to pass to the `tf.keras.optimizers.Optimizer` constructor.

fit_kwargs: (dict): Additional keyword arguments to pass to `tf.keras.Model.fit()`

Returns

history: (tf.keras.callbacks.History) Standard TensorFlow record of the training session.

`get_nn()`

Return the underlying TensorFlow neural network.

Parameters

None

Returns

nn: (tf.keras.Model) The backend TensorFlow model.

`get_theta(`*gauge='empirical'*, *p_lc=None*, *x_wt=None*, *unobserved_value=nan*`)`

Return parameters of the G-P map.

This function returns a `dict` containing the parameters of the model's G-P map. Keys are of type `str`, values are of type `np.ndarray` . Relevant (key, value) pairs are: `'theta_0'` , constant term; `'theta_lc'` , additive effects in the form of a 2D array with shape `(L,C)`; `'theta_lclc'` , pairwise effects in the form of a 4D array of shape `(L,C,L,C)`; `'theta_bb'` , all parameters for `gpmap_type='blackbox'` models.

Importantly this function gauge-fixes model parameters before returning them, i.e., it pins down non-identifiable degrees of freedom. Gauge fixing is performed using a hierarchical gauge, which maximizes the fraction of variance in `phi` explained by the lowest-order terms. Computing such variances requires assuming probability distribution over sequence space, however, and using different distributions will result in different ways of fixing the gauge.

This function assumes that the distribution used to define the gauge factorizes across sequence positions, and can thus be represented by an `L` x `C` probability matrix `p_lc` that lists the probability of each character `c` at each position `l`.

An important special case is the wild-type gauge, in which `p_lc` is the one-hot encoding of a "wild-type" specific sequence `x_wt`. In this case, the constant parameter `theta_0` is the value of `phi` for `x_wt`, additive parameters `theta_lc` represent the effect of single-point mutations away from `x_wt`, and so on.

Parameters

gauge: (str) String specification of which gauge to use. Allowed values are: `'uniform'` , hierarchical gauge using a uniform sequence distribution over the characters at each position observed in the training set (unobserved characters are assigned probability 0). `'empirical'` , hierarchical gauge using an empirical distribution computed from the training data; `'consensus'` , wild-type gauge using the training data consensus sequence; `'user'` , gauge using either `p_lc` or `x_wt` supplied by the user; `'none'` , no gauge fixing.

p_lc: (None, array) Custom probability matrix to use for hierarchical gauge fixing. Must be a `np.ndarray` of shape `(L,C)` . If using this, also set `gauge='user'`.

x_wt: (str, None) Custom wild-type sequence to use for wild-type gauge fixing. Must be a `str` of length `L`. If using this, also set `gauge='user'`.

unobserved_value: (float, None) Value to use for parameters when no corresponding sequences were present in the training data. If `None`, these parameters will be left alone. Using `np.nan` can help when visualizing models using `mavenn.heatmap()` or `mavenn.heatmap_pariwise()`.

Returns

theta: (dict) Model parameters provided as a `dict` of numpy arrays.

`p_of_y_given_phi`(*y*, *phi*, *paired=False*)
Compute probabilities p(`y` | `phi`).

Parameters

y: (np.ndarray) Measurement values. For GE models, must be an array of floats. For MPA models, must be an array of ints representing bin numbers.

phi: (np.ndarray) Latent phenotype values, provided as an array of floats.

paired: (bool) Whether values in `y` and `phi` should be treated as paired. If `True`, the probability of each value in `y` value will be computed using the single paired value in `phi`. If `False`, the probability of each value in `y` will be computed against all values of in `phi`.

Returns

p: (np.ndarray) Probability of `y` given `phi`. If `paired=True`, `p.shape` will be equal to both `y.shape` and `phi.shape`. If `paired=False`, `p.shape` will be given by `y.shape + phi.shape`.

`p_of_y_given_x`(*y*, *x*, *paired=True*)
Compute probabilities p(`y` | `x`).

Parameters

y: (np.ndarray) Measurement values. For GE models, must be an array of floats. For MPA models, must be an array of ints representing bin numbers.

x: (np.ndarray) Sequences, provided as an array of strings, each of length `L`.

paired: (bool) Whether values in `y` and `x` should be treated as paired. If `True`, the probability of each value in `y` value will be computed using the single paired value in `x`. If `False`, the probability of each value in `y` will be computed against all values of in `x`.

Returns

p: (np.ndarray) Probability of `y` given `x`. If `paired=True`, `p.shape` will be equal to both `y.shape` and `x.shape`. If `paired=False`, `p.shape` will be given by `y.shape + x.shape`.

`p_of_y_given_yhat`(*y*, *yhat*, *paired=False*)
Compute probabilities p(`y` | `yhat`); GE models only.

Parameters

y: (np.ndarray) Measurement values, provided as an array of floats.

yhat: (np.ndarray) Observable values, provided as an array of floats.

paired: (bool) Whether values in `y` and `yhat` should be treated as paired. If `True`, the probability of each value in `y` value will be computed using the single paired value in `yhat`. If `False`, the probability of each value in `y` will be computed against all values of in `yhat`.

Returns

p: (np.ndarray) Probability of `y` given `yhat`. If `paired=True`, `p.shape` will be equal to both `y.shape` and `yhat.shape`. If `paired=False`, `p.shape` will be given by `y.shape + yhat.shape` .

`phi_to_yhat`(*phi*)
Compute `phi` given `yhat`; GE models only.

Parameters

phi: (array-like) Latent phenotype values, provided as an `np.ndarray` of floats.

Returns

y_hat: (array-like) Observable values in an `np.ndarray` the same shape as `phi`.

`save`(*filename*, *verbose=True*)
Save model.

Saved models are represented by two files having the same root and two different extensions, `.pickle` and `.h5`. The `.pickle` file contains model metadata, including all information needed to reconstruct the model's architecture. The `.h5` file contains the values of the trained neural network weights. Note that training data is not saved.

Parameters

filename: (str) File directory and root. Do not include extensions.

verbose: (bool) Whether to print feedback.

Returns

None

`set_data`(*x*, *y*, *dy=None*, *ct=None*, *validation_frac=0.2*, *validation_flags=None*, *shuffle=True*, *knn_fuzz=0.01*, *verbose=True*)
Set training data.

Prepares data for use during training, e.g. by shuffling and one-hot encoding training data sequences. Must be called before `Model.fit()`.

Parameters

x: (np.ndarray) 1D array of `N` sequences, each of length `L`.

y: (np.ndarray) Array of measurements. For GE models, `y` must be a 1D array of `N` floats. For MPA models, `y` must be either a 1D or 2D array of nonnegative ints. If 1D, `y` must be of length `N`, and will be interpreted as listing bin numbers, i.e. `0` , `1` , ..., `Y-1`. If 2D, `y`

must be of shape `(N,Y)`, and will be interpreted as listing the observed counts for each of the `N` sequences in each of the `Y` bins.

dy [(np.ndarray)] User supplied error bars associated with continuous measurements to be used as sigma in the Gaussian noise model.

ct: (np.ndarray, None) Only used for MPA models when `y` is 1D. In this case, `ct` must be a 1D array, length `N`, of nonnegative integers, and represents the number of observations of each sequence in each bin. Use `y=None` for GE models, as well as for MPA models when `y` is 2D.

validation_frac (float): Fraction of observations to use for the validation set. Is overridden when setting `validation_flags`. Must be in the range [0,1].

validation_flags (np.ndarray, None): 1D array of `N` boolean numbers, with `True` indicating which observations should be reserved for the validation set. If `None`, the training and validation sets will be randomly assigned based on the value of `validation_frac`.

shuffle: (bool) Whether to shuffle the observations, e.g., to ensure similar composition of the training and validation sets when `validation_flags` is not set.

knn_fuzz: (float>0) Amount of noise to add to `y` values before passing them to the KNN estimator (for computing I_var during training). Specifically, Gaussian noise with standard deviation `knn_fuzz * np.std(y)` is added to `y` values. This is needed to mitigate errors caused by multiple observations of the same sequence. Only used for GE regression.

verbose: (bool) Whether to provide printed feedback.

Returns

None

`simulate_dataset`(*template_df*)

Generate a simulated dataset.

Parameters

template_df: (pd.DataFrame) Dataset off of which to base the simulated dataset. Specifically, the simulated dataset will have the same sequences and the same train/validation/test flags, but different values for `'y'` (in the case of a GE regression model) or `'ct_#'` (in the case of an MPA regression model.

Returns

simulated_df: (pd.DataFrame) Simulated dataset in the form of a dataframe. Columns include `'set'` , `'phi'` , and `'x'` . For GE models, additional columns `'yhat'` and `'y'` are added. For MPA models, multiple columns of the form `'ct_#'` are added.

`x_to_phi`(*x*)

Compute `phi` given `x`.

Parameters

x: (np.ndarray) Sequences, provided as an `np.ndarray` of strings, each of length `L`.

Returns

phi: (array-like of float) Latent phenotype values, provided as floats within an `np.ndarray` the same shape as `x`.

`x_to_yhat`(*x*)

Compute `yhat` given `x`.

Parameters

x: (np.ndarray) Sequences, provided as an `np.ndarray` of strings, each of length `L`.

Returns

yhat: (np.ndarray) Observation values, provided as floats within an `np.ndarray` the same shape as `x`.

`yhat_to_yq`(*yhat*, *q=[0.16, 0.84]*, *paired=False*)

Compute quantiles of p(`y` | `yhat`); GE models only.

Parameters

yhat: (np.ndarray) Observable values, provided as an array of floats.

q: (np.ndarray) Quantile specifications, provided as an array of floats in the range [0,1].

paired: (bool) Whether values in `yhat` and `q` should be treated as paired. If `True`, quantiles will be computed using each value in `yhat` paired with the corresponding value in `q`. If `False`, the quantile for each value in `yhat` will be computed for every value in `q`.

Returns

yq: (array of floats) Quantiles of p(`y` | `yhat`). If `paired=True`, `yq.shape` will be equal to both `yhat.shape` and `q.shape`. If `paired=False`, `yq.shape` will be given by `yhat.shape + q.shape`.

CHAPTER

SIX

LINKS

- Kinney Lab
- Cold Spring Harbor Laboratory

INDEX