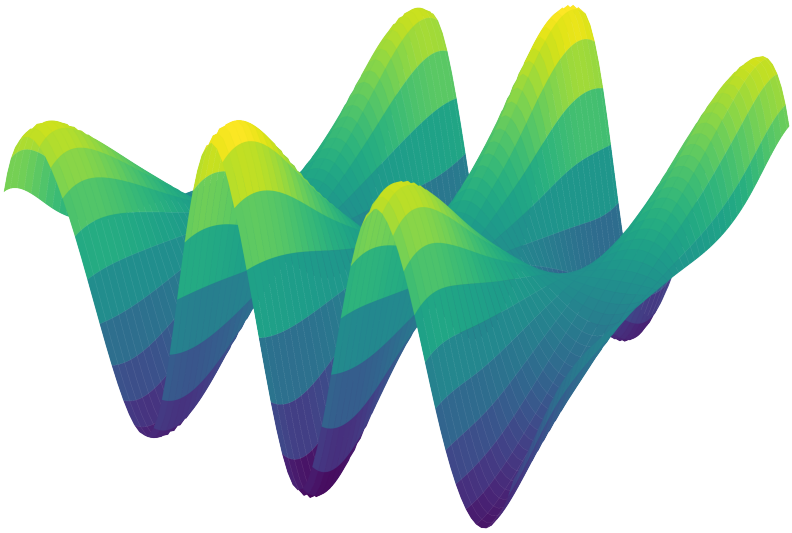
pyGAM Documentation

Daniel Servén

Jul 15, 2020

Contents:



pyGAM is a package for building Generalized Additive Models in Python, with an emphasis on modularity and performance. The API will be immediately familiar to anyone with experience of scikit-learn or scipy.

CHAPTER 1

Installation

pyGAM is on pypi, and can be installed using `pip`:

```
pip install pygam
```

Or via `conda-forge`, however this is typically less up-to-date:

```
conda install -c conda-forge pyGAM
```

You can install the bleeding edge from github using `flit`. First clone the repo, `cd` into the main directory and do:

```
pip install flit
flit install
```

1.1 Optional

To speed up optimization on large models with constraints, it helps to have `scikit-sparse` installed because it contains a slightly faster, sparse version of Cholesky factorization. The import from `scikit-sparse` references `nose`, so you'll need that too.

The easiest way is to use Conda:

```
conda install -c conda-forge scikit-sparse nose
```

More information is available in the scikit-sparse docs.

CHAPTER 2

Dependencies

pyGAM is tested on Python 2.7 and 3.6 and depends on `NumPy`, `SciPy`, and `progressbar2` (see `requirements.txt` for version information).

Optional: `scikit-sparse`.

In addtion to the above dependencies, the `datasets` submodule relies on `Pandas`.

CHAPTER 3

Citing pyGAM

Servén D., Brummitt C. (2018). pyGAM: Generalized Additive Models in Python. Zenodo. DOI: 10.5281/zenodo.1208723

CHAPTER 4

Contact

To report an issue with pyGAM please use the issue tracker.

CHAPTER 5

License

GNU General Public License v3.0

CHAPTER 6

Getting Started

If you're new to pyGAM, read *the Tour of pyGAM* for an introduction to the package.

6.1 Quick Start

This quick start will show how to do the following:

- `Install` everything needed to use pyGAM.
- `fit a regression model` with custom terms
- search for the `best smoothing parameters`
- plot `partial dependence` functions

6.1.1 Install pyGAM

Pip

```
pip install pygam
```

Conda

pyGAM is on conda-forge, however this is typically less up-to-date:

```
conda install -c conda-forge pygam
```

Bleeding edge

You can install the bleeding edge from github using `flit`. First clone the repo, `cd` into the main directory and do:

```
pip install flit
flit install
```

Get `pandas` and `matplotlib`

```
pip install pandas matplotlib
```

6.1.2 Fit a Model

Let's get to it. First we need some data:

[1]:
```
from pygam.datasets import wage

X, y = wage()
```

```
/home/dswah/miniconda3/envs/pygam36/lib/python3.6/importlib/_bootstrap.py:219:␣
↪RuntimeWarning: numpy.dtype size changed, may indicate binary incompatibility.␣
↪Expected 96, got 88
  return f(*args, **kwds)
```

Now let's import a GAM that's made for regression problems.

Let's fit a spline term to the first 2 features, and a factor term to the 3rd feature.

[2]:
```
from pygam import LinearGAM, s, f

gam = LinearGAM(s(0) + s(1) + f(2)).fit(X, y)
```

Let's take a look at the model fit:

[3]:
```
gam.summary()
```

```
LinearGAM
=============================================␣
↪============================================================
Distribution:                        NormalDist Effective DoF:                     ␣
↪             25.1911
Link Function:                     IdentityLink Log Likelihood:                    ␣
↪         -24118.6847
Number of Samples:                         3000 AIC:                               ␣
↪          48289.7516
                                                AICc:                              ␣
↪          48290.2307
                                                GCV:                               ␣
↪           1255.6902
                                                Scale:                             ␣
↪           1236.7251
                                                Pseudo R-Squared:                  ␣
↪              0.2955
==========================================================================================
Feature Function                  Lambda               Rank         EDoF         P >␣
↪x        Sig. Code
```

(continues on next page)

(continued from previous page)

```
================================= ==================== ============ ============
 ↪============ ============
s(0)                              [0.6]                20           7.1          5.
 ↪95e-03     **
s(1)                              [0.6]                20           14.1         1.
 ↪11e-16     ***
f(2)                              [0.6]                5            4.0          1.
 ↪11e-16     ***
intercept                                              1            0.0          1.
 ↪11e-16     ***
==========================================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model␣
 ↪identifiability problem
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models␣
 ↪or models with
         known smoothing parameters, but when smoothing parameters have been␣
 ↪estimated, the p-values
         are typically lower than they should be, meaning that the tests reject the␣
 ↪null too readily.
```

Even though we have 3 terms with a total of `(20 + 20 + 5) = 45` free variables, the default smoothing penalty (`lam=0.6`) reduces the effective degrees of freedom to just ~25.

By default, the spline terms, `s(...)`, use 20 basis functions. This is a good starting point. The rule of thumb is to use a fairly large amount of flexibility, and then let the smoothing penalty regularize the model.

However, we can always use our expert knowledge to add flexibility where it is needed, or remove basis functions, and make fitting easier:

[22]:
```
gam = LinearGAM(s(0, n_splines=5) + s(1) + f(2)).fit(X, y)
```

6.1.3 Automatically tune the model

By default, spline terms, `s()` have a penalty on their 2nd derivative, which encourages the functions to be smoother, while factor terms, `f()` and linear terms `l()`, have a l2, ie ridge penalty, which encourages them to take on smaller values.

`lam`, short for λ, controls the strength of the regularization penalty on each term. Terms can have multiple penalties, and therefore multiple `lam`.

[14]:
```
print(gam.lam)
```

```
[[0.6], [0.6], [0.6]]
```

Our model has 3 `lam` parameters, currently just one per term.

Let's perform a grid-search over multiple `lam` values to see if we can improve our model.
We will seek the model with the lowest generalized cross-validation (GCV) score.

Our search space is 3-dimensional, so we have to be conservative with the number of points we consider per dimension.

Let's try 5 values for each smoothing parameter, resulting in a total of `5*5*5 = 125` points in our grid.

[15]:
```
import numpy as np

lam = np.logspace(-3, 5, 5)
lams = [lam] * 3

gam.gridsearch(X, y, lam=lams)
gam.summary()
```

```
100% (125 of 125) |######################| Elapsed Time: 0:00:07 Time:  0:00:07
LinearGAM
=============================================
 ↪=============================================================
Distribution:                        NormalDist Effective DoF:                      ⌟
 ↪               9.2948
Link Function:                     IdentityLink Log Likelihood:                     ⌟
 ↪          -24119.7277
Number of Samples:                         3000 AIC:                                ⌟
 ↪           48260.0451
                                                AICc:                               ⌟
 ↪           48260.1229
                                                GCV:                                ⌟
 ↪             1244.089
                                                Scale:                              ⌟
 ↪            1237.1528
                                                Pseudo R-Squared:                   ⌟
 ↪               0.2915
=========================================================================================================
Feature Function                  Lambda               Rank         EDoF         P >⌟
 ↪x        Sig. Code
================================= ==================== ============ ============⌟
 ↪============ ============
s(0)                              [100000.]            5            2.0          7.
 ↪54e-03     **
s(1)                              [1000.]              20           3.3          1.
 ↪11e-16     ***
f(2)                              [0.1]                5            4.0          1.
 ↪11e-16     ***
intercept                                              1            0.0          1.
 ↪11e-16     ***
=========================================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model⌟
 ↪identifiability problem
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models⌟
 ↪or models with
         known smoothing parameters, but when smoothing parameters have been⌟
 ↪estimated, the p-values
         are typically lower than they should be, meaning that the tests reject the⌟
 ↪null too readily.
```

This is quite a bit better. Even though the in-sample R^2 value is lower, we can expect our model to generalize better because the GCV error is lower.

We could be more rigorous by using a train/test split, and checking our model's error on the test set. We were also

quite lazy and only tried 125 values in our hyperopt. We might find a better model if we spent more time searching across more points.

For high-dimensional search-spaces, it is sometimes a good idea to try a **randomized search**.
We can acheive this by using numpy's `random` module:

[16]:
```
lams = np.random.rand(100, 3) # random points on [0, 1], with shape (100, 3)
lams = lams * 6 - 3 # shift values to -3, 3
lams = 10 ** lams # transforms values to 1e-3, 1e3
```

[17]:
```
random_gam =  LinearGAM(s(0) + s(1) + f(2)).gridsearch(X, y, lam=lams)
random_gam.summary()
```

```
100% (100 of 100) |######################| Elapsed Time: 0:00:07 Time:  0:00:07
LinearGAM
=============================================
=============================================================
Distribution:                        NormalDist Effective DoF:
             15.6683
Link Function:                     IdentityLink Log Likelihood:
         -24115.6727
Number of Samples:                         3000 AIC:
          48264.6819
                                                AICc:
          48264.8794
                                                GCV:
           1247.2011
                                                Scale:
           1235.4817
                                                Pseudo R-Squared:
              0.2939
=================================================================================================
Feature Function                  Lambda               Rank         EDoF         P >
x        Sig. Code
================================= ==================== ============ ============
============ ============
s(0)                              [137.6336]           20           6.3          7.
08e-03     **
s(1)                              [128.3511]           20           5.4          1.
11e-16     ***
f(2)                              [0.3212]             5            4.0          1.
11e-16     ***
intercept                                              1            0.0          1.
11e-16     ***
=================================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model
identifiability problem
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models
or models with
         known smoothing parameters, but when smoothing parameters have been
estimated, the p-values
```

(continues on next page)

(continued from previous page)

```
        are typically lower than they should be, meaning that the tests reject the␣
↪null too readily.
```

In this case, our deterministic search found a better model:

[18]:
```
gam.statistics_['GCV'] < random_gam.statistics_['GCV']
```

[18]:
```
True
```

The `statistics_` attribute is populated after the model has been fitted. There are lots of interesting model statistics to check out, although many are automatically reported in the model summary:

[19]:
```
list(gam.statistics_.keys())
```

[19]:
```
['n_samples',
 'm_features',
 'edof_per_coef',
 'edof',
 'scale',
 'cov',
 'se',
 'AIC',
 'AICc',
 'pseudo_r2',
 'GCV',
 'UBRE',
 'loglikelihood',
 'deviance',
 'p_values']
```

6.1.4 Partial Dependence Functions

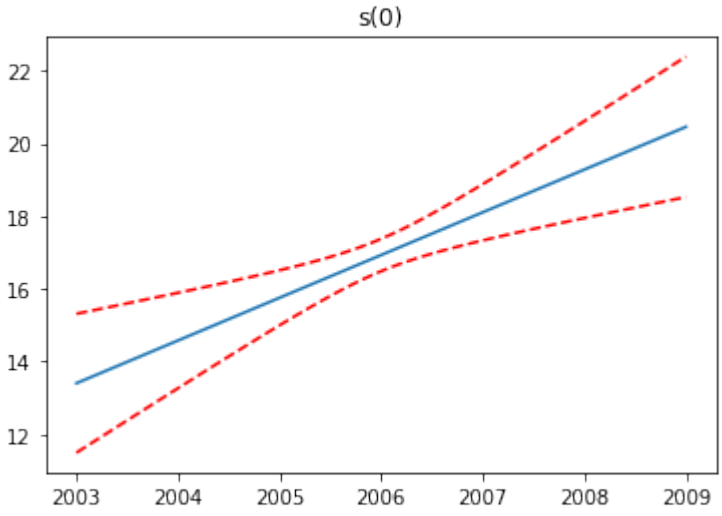
One of the most attractive properties of GAMs is that we can decompose and inspect the contribution of each feature to the overall prediction.

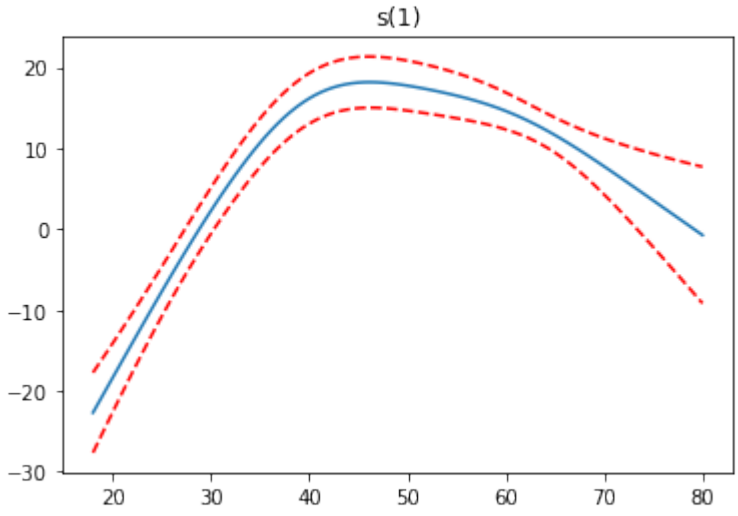
This is done via **partial dependence** functions.

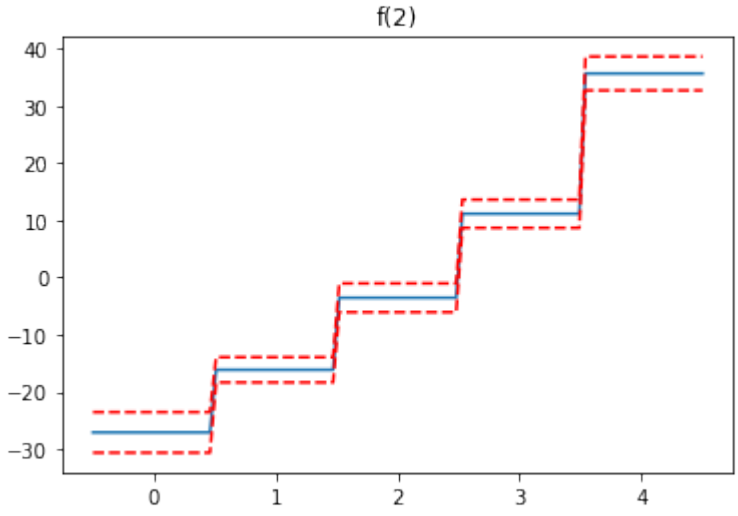
Let's plot the partial dependence for each term in our model, along with a 95% confidence interval for the estimated function.

[20]:
```
import matplotlib.pyplot as plt
```

[21]:
```
for i, term in enumerate(gam.terms):
    if term.isintercept:
        continue

    XX = gam.generate_X_grid(term=i)
    pdep, confi = gam.partial_dependence(term=i, X=XX, width=0.95)

    plt.figure()
    plt.plot(XX[:, term.feature], pdep)
    plt.plot(XX[:, term.feature], confi, c='r', ls='--')
    plt.title(repr(term))
    plt.show()
```







Note: we skip the intercept term because it has nothing interesting to plot.

[ ]:

6.2 A Tour of pyGAM

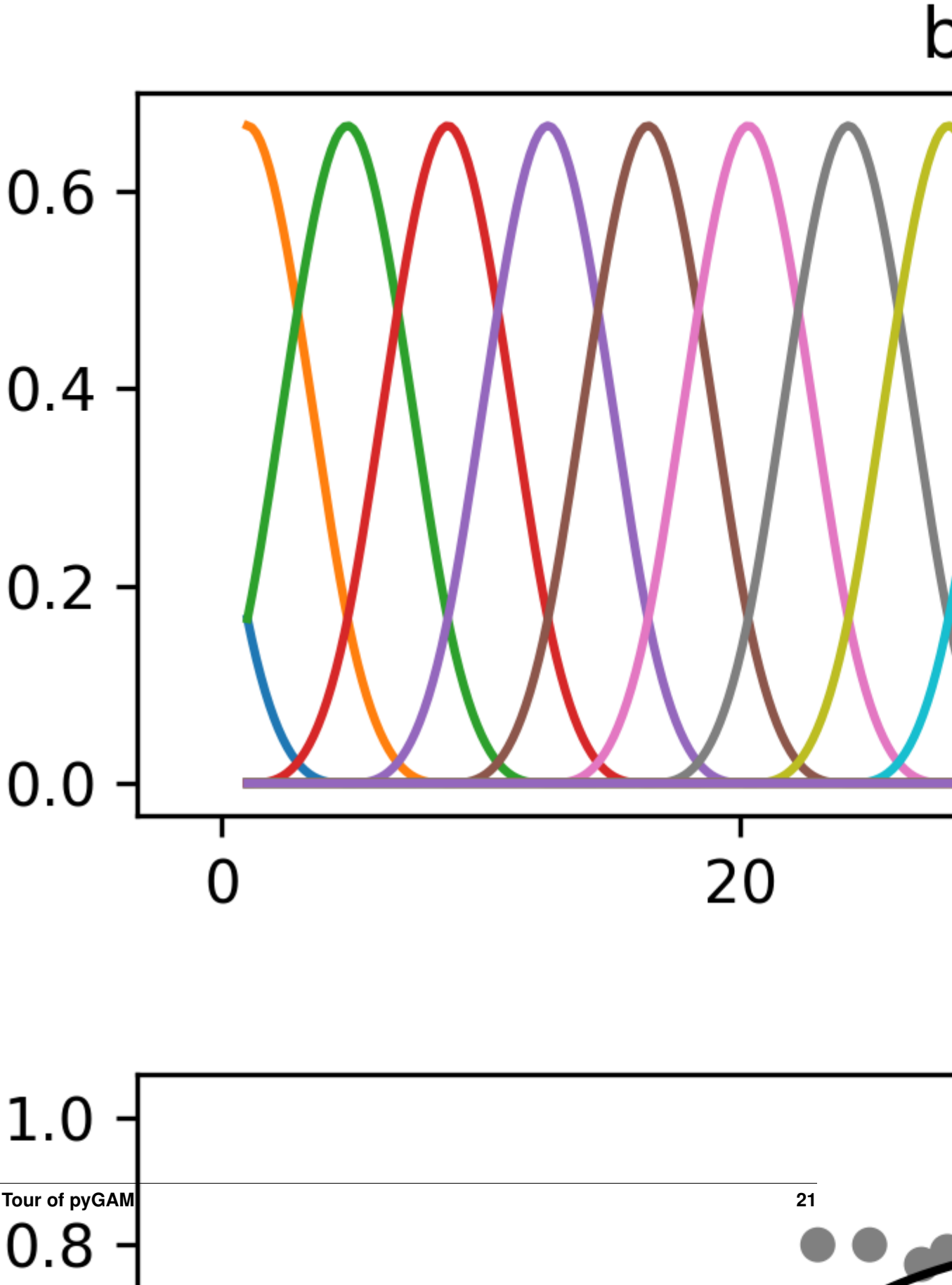
6.2.1 Introduction

Generalized Additive Models (GAMs) are smooth semi-parametric models of the form:

$$g(\mathbb{E}[y|X]) = \beta_0 + f_1(X_1) + f_2(X_2, X3) + \ldots + f_M(X_N)$$

where `X.T = [X_1, X_2, ..., X_N]` are independent variables, `y` is the dependent variable, and `g()` is the link function that relates our predictor variables to the expected value of the dependent variable.

The feature functions `f_i()` are built using **penalized B splines**, which allow us to **automatically model non-linear relationships** without having to manually try out many different transformations on each variable.


b
0.6
0.4
0.2
0.0
0
20
1.0
0.8

GAMs extend generalized linear models by allowing non-linear functions of features while maintaining additivity. Since the model is additive, it is easy to examine the effect of each `X_i` on `Y` individually while holding all other predictors constant.

The result is a very flexible model, where it is easy to incorporate prior knowledge and control overfitting.

6.2.2 Generalized Additive Models, in general

$$y \sim ExponentialFamily(\mu|X)$$

where

$$g(\mu|X) = \beta_0 + f_1(X_1) + f_2(X_2, X3) + \ldots + f_M(X_N)$$

So we can see that a GAM has 3 components:

- `distribution` from the exponential family
- `link function` $g(\cdot)$
- `functional form` with an additive structure $\beta_0 + f_1(X_1) + f_2(X_2, X3) + \ldots + f_M(X_N)$

Distribution:

Specified via: `GAM(distribution='...')`

Currently you can choose from the following:

- `'normal'`
- `'binomial'`
- `'poisson'`
- `'gamma'`
- `'inv_gauss'`

Link function:

We specify this using: `GAM(link='...')`

Link functions take the distribution mean to the linear prediction. So far, the following are available:

- `'identity'`
- `'logit'`
- `'inverse'`
- `'log'`
- `'inverse-squared'`

Functional Form:

Speficied in `GAM(terms=...)` or more simply `GAM(...)`

In pyGAM, we specify the functional form using terms:

- `l()` linear terms: for terms like X_i
- `s()` spline terms
- `f()` factor terms
- `te()` tensor products
- `intercept`

With these, we can quickly and compactly build models like:

[12]:
```
from pygam import GAM, s, te

GAM(s(0, n_splines=200) + te(3,1) + s(2), distribution='poisson', link='log')
```

[12]:
```
GAM(callbacks=['deviance', 'diffs'], distribution='poisson',
   fit_intercept=True, link='log', max_iter=100,
   terms=s(0) + te(3, 1) + s(2), tol=0.0001, verbose=False)
```

which specifies that we want a:

- spline function on feature 0, with 200 basis functions
- tensor spline interaction on features 1 and 3
- spline function on feature 2

Note:

`GAM(..., intercept=True)` so models include an intercept by default.

in Practice...

in **pyGAM** you can build custom models by specifying these 3 elements, **or** you can choose from **common models**:

- `LinearGAM` identity link and normal distribution
- `LogisticGAM` logit link and binomial distribution
- `PoissonGAM` log link and Poisson distribution
- `GammaGAM` log link and gamma distribution
- `InvGauss` log link and inv_gauss distribution

The benefit of the common models is that they have some extra features, apart from reducing boilerplate code.

6.2.3 Terms and Interactions

pyGAM can also fit interactions using tensor products via `te()`

[58]:
```
from pygam import PoissonGAM, s, te
from pygam.datasets import chicago

X, y = chicago(return_X_y=True)
```

(continues on next page)

(continued from previous page)

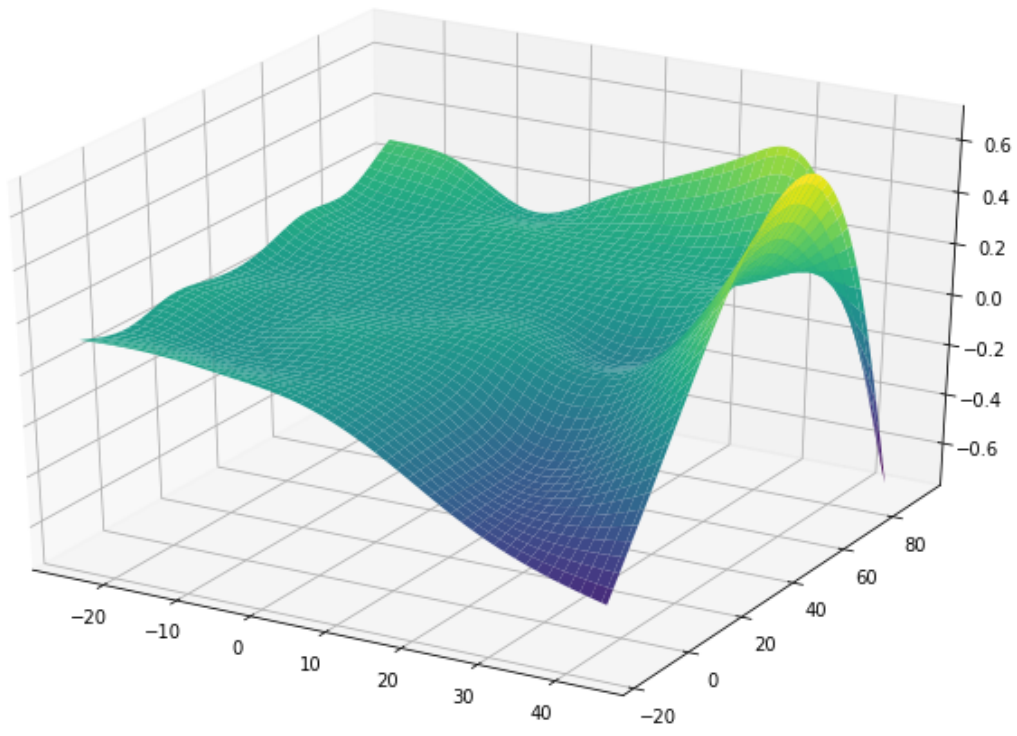
```
gam = PoissonGAM(s(0, n_splines=200) + te(3, 1) + s(2)).fit(X, y)
```

and plot a 3D surface:

[60]:
```
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

plt.ion()
plt.rcParams['figure.figsize'] = (12, 8)
```

[61]:
```
XX = gam.generate_X_grid(term=1, meshgrid=True)
Z = gam.partial_dependence(term=1, X=XX, meshgrid=True)

ax = plt.axes(projection='3d')
ax.plot_surface(XX[0], XX[1], Z, cmap='viridis')
```

[61]:
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f58f3427cc0>
```



For simple interactions it is sometimes useful to add a by-variable to a term

[10]:
```
from pygam import LinearGAM, s
from pygam.datasets import toy_interaction

X, y = toy_interaction(return_X_y=True)

gam = LinearGAM(s(0, by=1)).fit(X, y)
gam.summary()
```

```
LinearGAM
=============================================
 ↪============================================================
Distribution:                        NormalDist Effective DoF:                      ␣
 ↪             20.8449
Link Function:                     IdentityLink Log Likelihood:                     ␣
 ↪       -2317525.6219
Number of Samples:                        50000 AIC:                                ␣
 ↪        4635094.9336
                                                AICc:                               ␣
 ↪        4635094.9536
                                                GCV:                                ␣
 ↪                0.01
                                                Scale:                              ␣
 ↪                0.01
                                                Pseudo R-Squared:                   ␣
 ↪              0.9976
==========================================================================================
Feature Function                  Lambda               Rank         EDoF         P >␣
 ↪x        Sig. Code
================================= ==================== ============ ============␣
 ↪============ ============
s(0)                              [0.6]                20           19.8         1.
 ↪11e-16     ***
intercept                                              1            1.0          1.
 ↪79e-01
==========================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model␣
 ↪identifiability problem
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models␣
 ↪or models with
         known smoothing parameters, but when smoothing parameters have been␣
 ↪estimated, the p-values
         are typically lower than they should be, meaning that the tests reject the␣
 ↪null too readily.
```

6.2.4 Regression

For **regression** problems, we can use a **linear GAM** which models:

$$\mathbb{E}[y|X] = \beta_0 + f_1(X_1) + f_2(X_2, X3) + \cdots + f_M(X_N)$$

[17]:
```
from pygam import LinearGAM, s, f
from pygam.datasets import wage

X, y = wage(return_X_y=True)

## model
gam = LinearGAM(s(0) + s(1) + f(2))
gam.gridsearch(X, y)
```

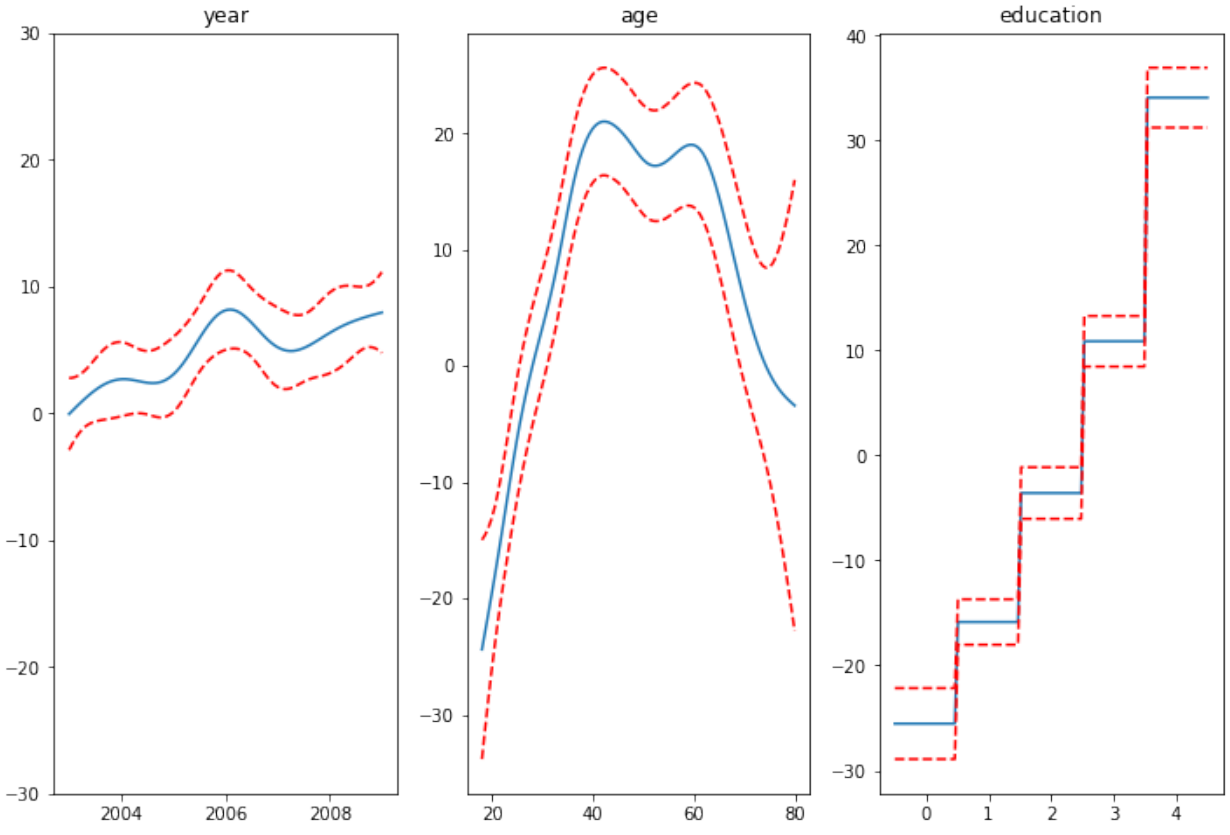
(continues on next page)

(continued from previous page)

```
## plotting
plt.figure();
fig, axs = plt.subplots(1,3);

titles = ['year', 'age', 'education']
for i, ax in enumerate(axs):
    XX = gam.generate_X_grid(term=i)
    ax.plot(XX[:, i], gam.partial_dependence(term=i, X=XX))
    ax.plot(XX[:, i], gam.partial_dependence(term=i, X=XX, width=.95)[1], c='r', ls='-
↪-')
    if i == 0:
        ax.set_ylim(-30,30)
    ax.set_title(titles[i]);
```

```
100% (11 of 11) |########################| Elapsed Time: 0:00:01 Time:  0:00:01
```

```
<Figure size 864x576 with 0 Axes>
```



Even though our model allows coefficients, our smoothing penalty reduces us to just 19 effective degrees of freedom:

[4]:
```
gam.summary()
```

```
LinearGAM
=============================================␣
↪============================================================
Distribution:                        NormalDist Effective DoF:
↪                     19.2602
```

(continues on next page)

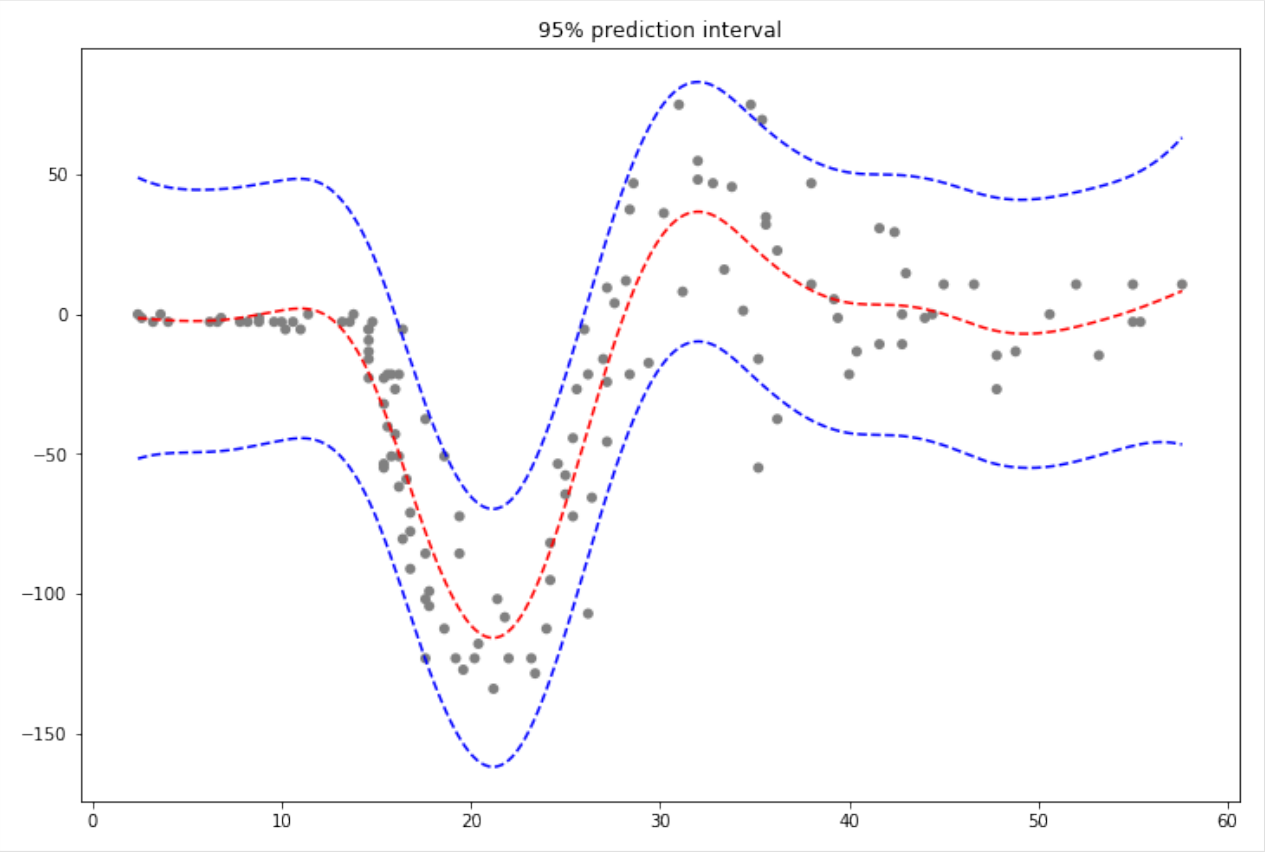
(continued from previous page)

```
Link Function:                     IdentityLink Log Likelihood:                                ␣
↪         -24116.7451
Number of Samples:                         3000 AIC:                                           ␣
↪          48274.0107
                                                AICc:                                          ␣
↪          48274.2999
                                                GCV:                                           ␣
↪           1250.3656
                                                Scale:                                         ␣
↪           1235.9245
                                                Pseudo R-Squared:                              ␣
↪              0.2945
==========================================================================================================
Feature Function                  Lambda               Rank         EDoF         P >␣
↪x        Sig. Code
================================= ==================== ============ ============␣
↪============ ============
s(0)                              [15.8489]            20           6.9          5.
↪52e-03     **
s(1)                              [15.8489]            20           8.5          1.
↪11e-16     ***
f(2)                              [15.8489]            5            3.8          1.
↪11e-16     ***
intercept                         0                    1            0.0          1.
↪11e-16     ***
==========================================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model␣
↪identifiability problem
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models␣
↪or models with
         known smoothing parameters, but when smoothing parameters have been␣
↪estimated, the p-values
         are typically lower than they should be, meaning that the tests reject the␣
↪null too readily.
```

With **LinearGAMs**, we can also check the **prediction intervals**:

[56]:
```
from pygam import LinearGAM
from pygam.datasets import mcycle

X, y = mcycle(return_X_y=True)

gam = LinearGAM(n_splines=25).gridsearch(X, y)
XX = gam.generate_X_grid(term=0, n=500)

plt.plot(XX, gam.predict(XX), 'r--')
plt.plot(XX, gam.prediction_intervals(XX, width=.95), color='b', ls='--')

plt.scatter(X, y, facecolor='gray', edgecolors='none')
plt.title('95% prediction interval');
```

```
100% (11 of 11) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
```

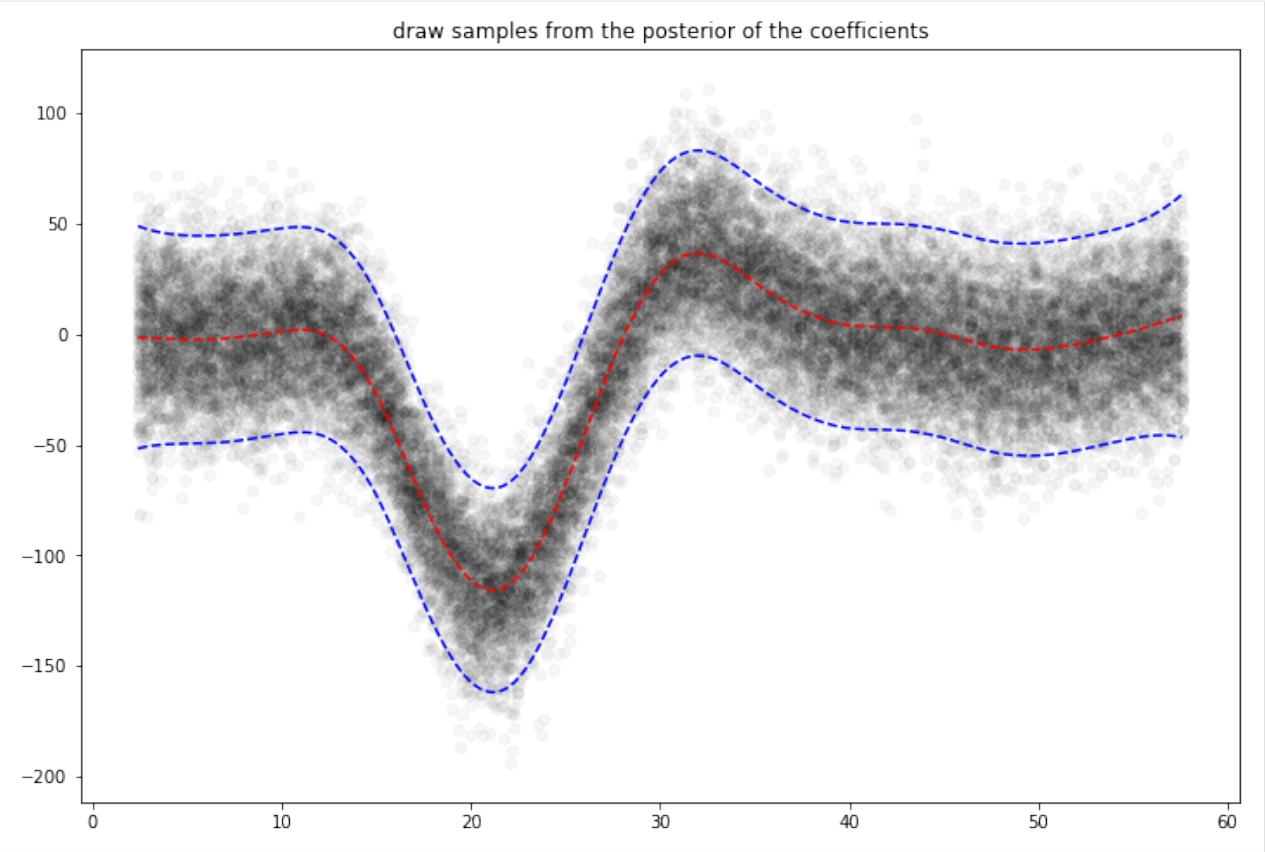


And simulate from the posterior:

[57]:
```
# continuing last example with the mcycle dataset
for response in gam.sample(X, y, quantity='y', n_draws=50, sample_at_X=XX):
    plt.scatter(XX, response, alpha=.03, color='k')
plt.plot(XX, gam.predict(XX), 'r--')
plt.plot(XX, gam.prediction_intervals(XX, width=.95), color='b', ls='--')
plt.title('draw samples from the posterior of the coefficients')
```

```
100% (11 of 11) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
100% (11 of 11) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
100% (11 of 11) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
100% (11 of 11) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
```

[57]:
```
Text(0.5,1,'draw samples from the posterior of the coefficients')
```



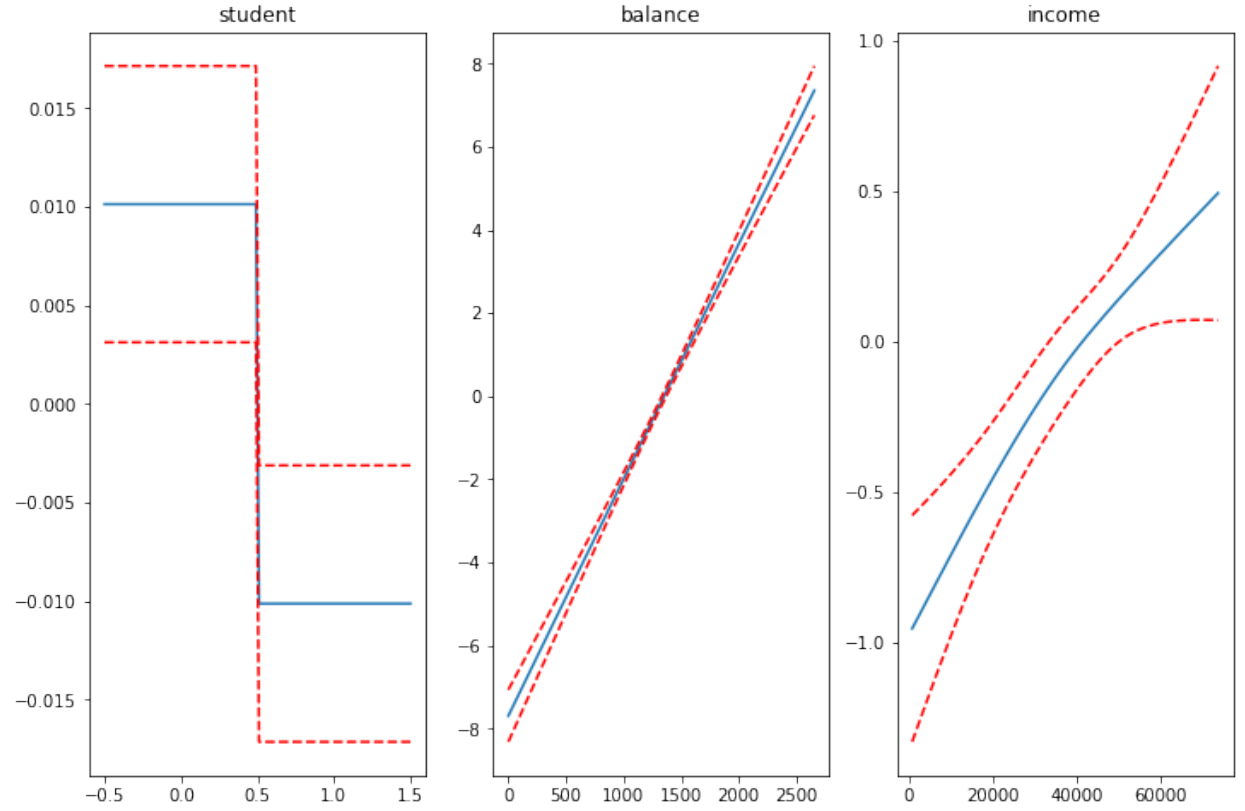
6.2.5 Classification

For **binary classification** problems, we can use a **logistic GAM** which models:

$$log\left(\frac{P(y=1|X)}{P(y=0|X)}\right) = \beta_0 + f_1(X_1) + f_2(X_2, X3) + \cdots + f_M(X_N)$$

[11]:
```
from pygam import LogisticGAM, s, f
from pygam.datasets import default

X, y = default(return_X_y=True)

gam = LogisticGAM(f(0) + s(1) + s(2)).gridsearch(X, y)

fig, axs = plt.subplots(1, 3)
titles = ['student', 'balance', 'income']

for i, ax in enumerate(axs):
    XX = gam.generate_X_grid(term=i)
    pdep, confi = gam.partial_dependence(term=i, width=.95)

    ax.plot(XX[:, i], pdep)
    ax.plot(XX[:, i], confi, c='r', ls='--')
    ax.set_title(titles[i]);
```

```
100% (11 of 11) |########################| Elapsed Time: 0:00:04 Time:  0:00:04
```



We can then check the accuracy:

[8]:
```
gam.accuracy(X, y)
```

[8]:
```
0.9739
```

Since the **scale** of the **Binomial distribution** is known, our gridsearch minimizes the **Un-Biased Risk Estimator** (UBRE) objective:

[9]:
```
gam.summary()
```

```
LogisticGAM
=============================================
 ↪============================================================
Distribution:                        BinomialDist Effective DoF:
 ↪               3.8047
Link Function:                          LogitLink Log Likelihood:
 ↪             -788.877
Number of Samples:                           10000 AIC:
 ↪            1585.3634
                                                   AICc:
 ↪             1585.369
                                                   UBRE:
 ↪               2.1588
                                                   Scale:
 ↪                  1.0
                                                   Pseudo R-Squared:
 ↪               0.4598
```

(continues on next page)

(continued from previous page)

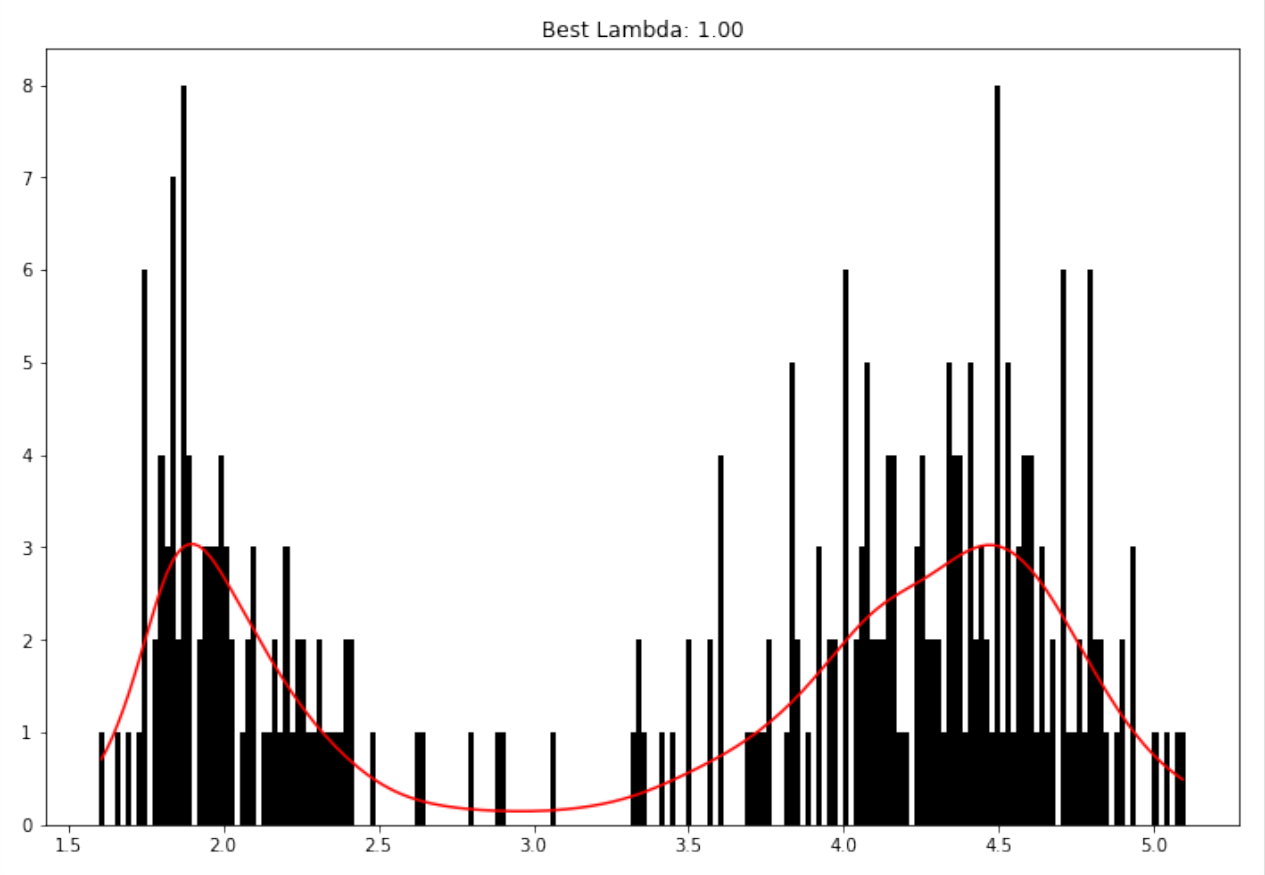
```
================================================================================================================
Feature Function                  Lambda               Rank         EDoF         P >␣
↪x        Sig. Code
================================= ==================== ============ ============␣
↪============ ============
f(0)                              [1000.]              2            1.7          4.
↪61e-03     **
s(1)                              [1000.]              20           1.2          0.
↪00e+00     ***
s(2)                              [1000.]              20           0.8          3.
↪29e-02     *
intercept                         0                    1            0.0          0.
↪00e+00     ***
================================================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model␣
↪identifiability problem
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models␣
↪or models with
         known smoothing parameters, but when smoothing parameters have been␣
↪estimated, the p-values
         are typically lower than they should be, meaning that the tests reject the␣
↪null too readily.
```

6.2.6 Poisson and Histogram Smoothing

We can intuitively perform **histogram smoothing** by modeling the counts in each bin as being distributed Poisson via **PoissonGAM**.

[10]:
```
from pygam import PoissonGAM
from pygam.datasets import faithful

X, y = faithful(return_X_y=True)

gam = PoissonGAM().gridsearch(X, y)

plt.hist(faithful(return_X_y=False)['eruptions'], bins=200, color='k');
plt.plot(X, gam.predict(X), color='r')
plt.title('Best Lambda: {0:.2f}'.format(gam.lam[0][0]));
```

```
100% (11 of 11) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
```



6.2.7 Expectiles

GAMs with a Normal distribution suffer from the limitation of an assumed constant variance. Sometimes this is not an appropriate assumption, because we'd like the variance of our error distribution to vary.

In this case we can resort to modeling the **expectiles** of a distribution.

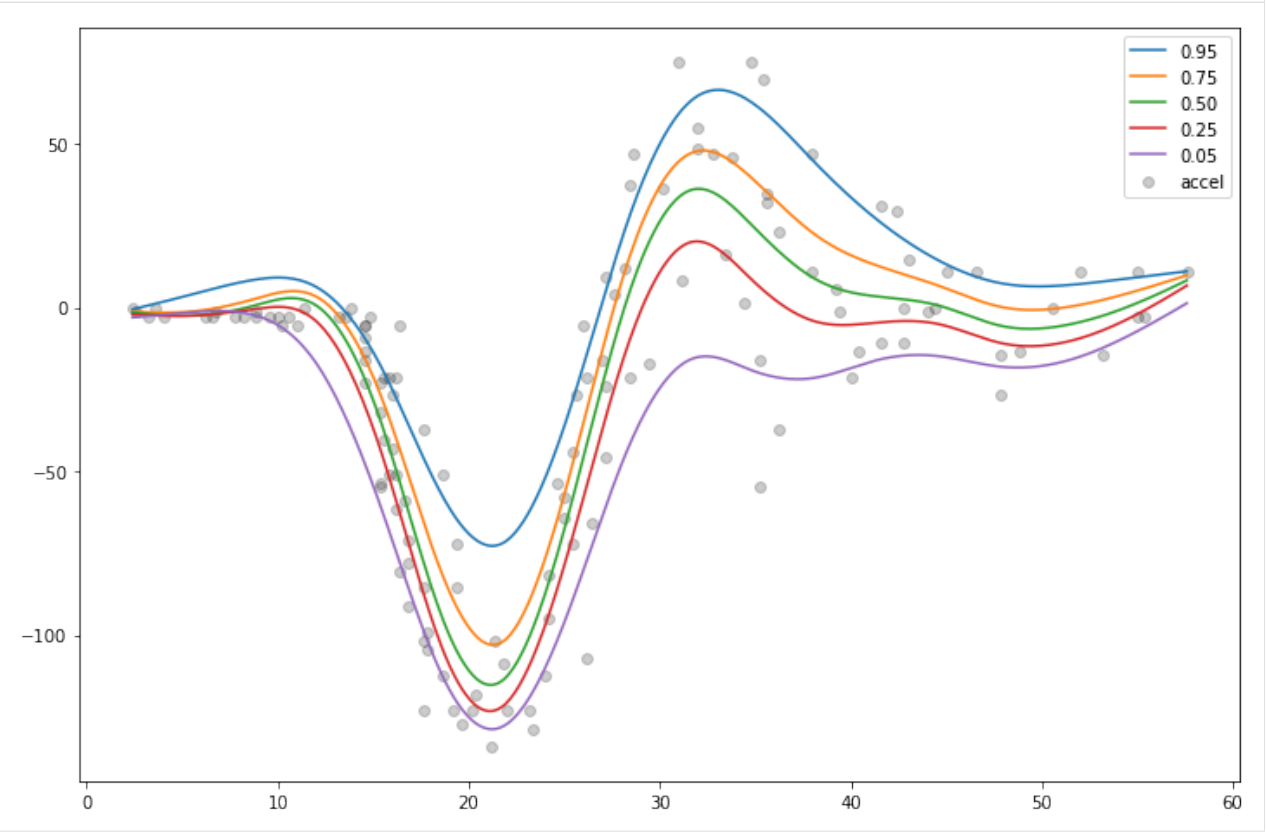
Expectiles are intuitively similar to quantiles, but model tail expectations instead of tail mass. Although they are less interpretable, expectiles are **much** faster to fit, and can also be used to non-parametrically model a distribution.

[52]:
```
from pygam import ExpectileGAM
from pygam.datasets import mcycle

X, y = mcycle(return_X_y=True)

# lets fit the mean model first by CV
gam50 = ExpectileGAM(expectile=0.5).gridsearch(X, y)

# and copy the smoothing to the other models
lam = gam50.lam

# now fit a few more models
gam95 = ExpectileGAM(expectile=0.95, lam=lam).fit(X, y)
gam75 = ExpectileGAM(expectile=0.75, lam=lam).fit(X, y)
gam25 = ExpectileGAM(expectile=0.25, lam=lam).fit(X, y)
gam05 = ExpectileGAM(expectile=0.05, lam=lam).fit(X, y)
```

```
100% (11 of 11) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
```

[55]:
```
XX = gam50.generate_X_grid(term=0, n=500)

plt.scatter(X, y, c='k', alpha=0.2)
plt.plot(XX, gam95.predict(XX), label='0.95')
plt.plot(XX, gam75.predict(XX), label='0.75')
plt.plot(XX, gam50.predict(XX), label='0.50')
plt.plot(XX, gam25.predict(XX), label='0.25')
plt.plot(XX, gam05.predict(XX), label='0.05')
plt.legend()
```

[55]:
```
<matplotlib.legend.Legend at 0x7f58f816c3c8>
```



We fit the **mean model** by cross-validation in order to find the best smoothing parameter `lam` and then copy it over to the other models.

This practice makes the expectiles less likely to cross.

6.2.8 Custom Models

It's also easy to build custom models by using the base **GAM** class and specifying the **distribution** and the **link function**:

[27]:
```
from pygam import GAM
from pygam.datasets import trees

X, y = trees(return_X_y=True)
```

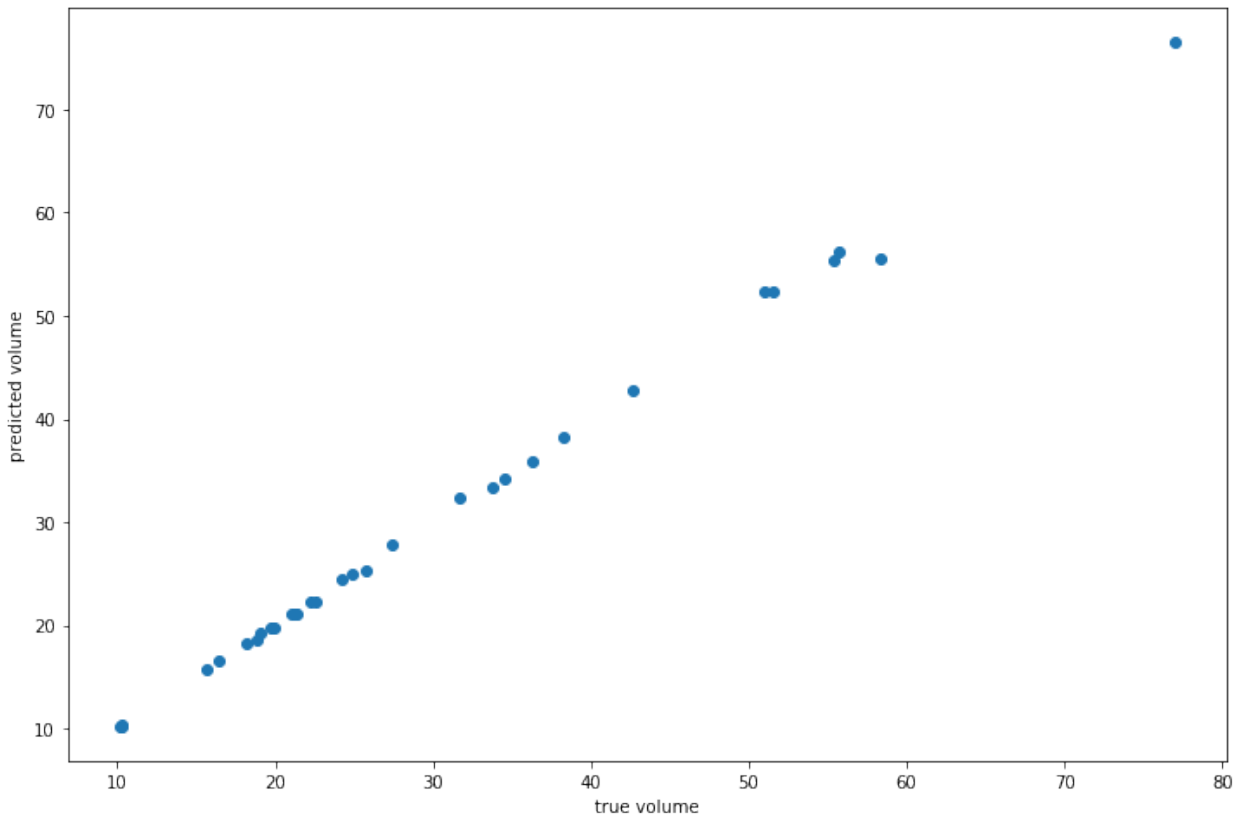
(continues on next page)

(continued from previous page)

```
gam = GAM(distribution='gamma', link='log')
gam.gridsearch(X, y)

plt.scatter(y, gam.predict(X))
plt.xlabel('true volume')
plt.ylabel('predicted volume')
```

```
100% (11 of 11) |########################| Elapsed Time: 0:00:01 Time:  0:00:01
```

[27]:
```
Text(0,0.5,'predicted volume')
```



We can check the quality of the fit by looking at the Pseudo R-Squared:

[28]:
```
gam.summary()
```

```
GAM
=============================================
 ↪============================================================
Distribution:                        GammaDist Effective DoF:
 ↪              25.3616
Link Function:                         LogLink Log Likelihood:
 ↪             -26.1673
Number of Samples:                           31 AIC:
 ↪             105.0579
                                                AICc:
 ↪             501.5549
                                                GCV:
 ↪               0.0088
```

(continues on next page)

(continued from previous page)

```
                                                Scale:                                 ␣
↪               0.001
                                                Pseudo R-Squared:                      ␣
↪              0.9993
==========================================================================================================
Feature Function                  Lambda               Rank         EDoF         P >␣
↪x        Sig. Code
================================= ==================== ============ ============␣
↪============ ============
s(0)                              [0.001]              20                        2.
↪04e-08     ***
s(1)                              [0.001]              20                        7.
↪36e-06     ***
intercept                         0                    1                         4.
↪39e-13     ***
==========================================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model␣
↪identifiability problem
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models␣
↪or models with
         known smoothing parameters, but when smoothing parameters have been␣
↪estimated, the p-values
         are typically lower than they should be, meaning that the tests reject the␣
↪null too readily.
```

6.2.9 Penalties / Constraints

With GAMs we can encode **prior knowledge** and **control overfitting** by using penalties and constraints.

Available penalties - second derivative smoothing (default on numerical features) - L2 smoothing (default on categorical features)

Availabe constraints - monotonic increasing/decreasing smoothing - convex/concave smoothing - periodic smoothing [soon...]

We can inject our intuition into our model by using **monotonic** and **concave** constraints:

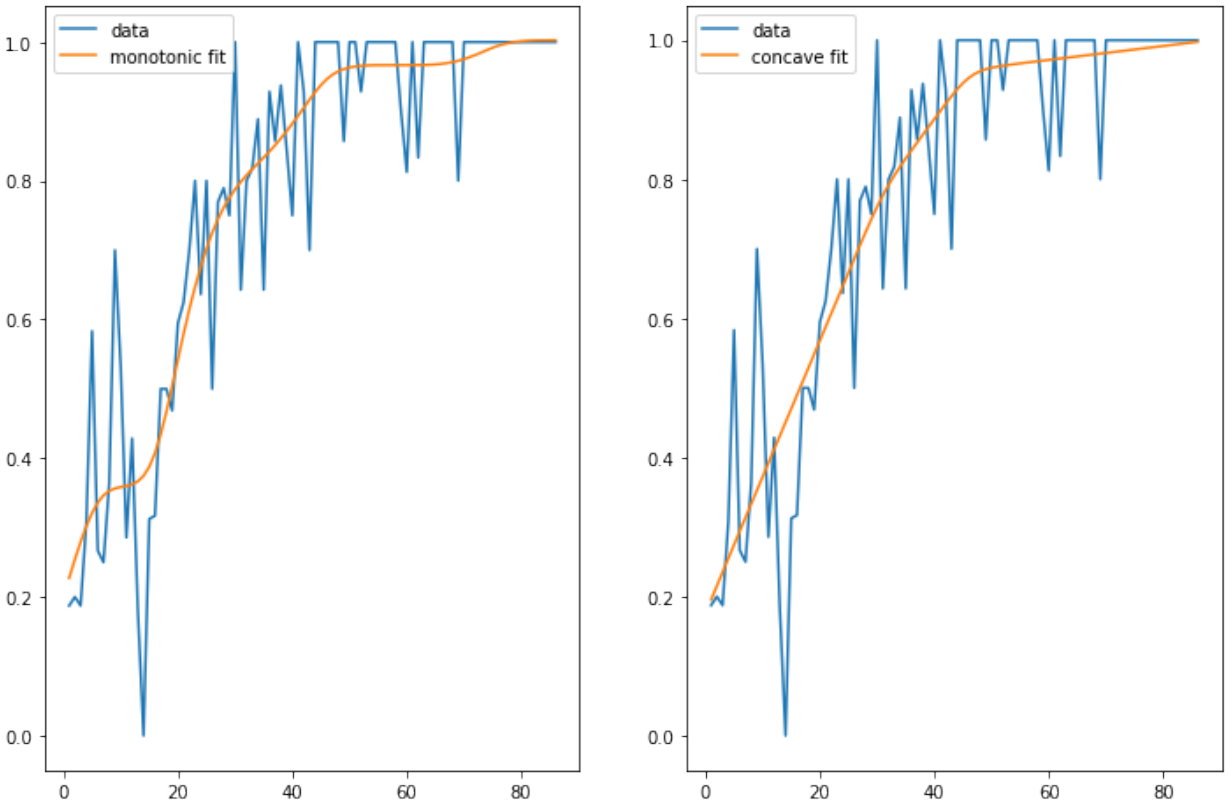
[29]:
```
from pygam import LinearGAM, s
from pygam.datasets import hepatitis

X, y = hepatitis(return_X_y=True)

gam1 = LinearGAM(s(0, constraints='monotonic_inc')).fit(X, y)
gam2 = LinearGAM(s(0, constraints='concave')).fit(X, y)

fig, ax = plt.subplots(1, 2)
ax[0].plot(X, y, label='data')
ax[0].plot(X, gam1.predict(X), label='monotonic fit')
ax[0].legend()

ax[1].plot(X, y, label='data')
```

(continues on next page)

(continued from previous page)

```
ax[1].plot(X, gam2.predict(X), label='concave fit')
ax[1].legend()
```

[29]:
```
<matplotlib.legend.Legend at 0x7fa3970b19e8>
```



6.2.10 API

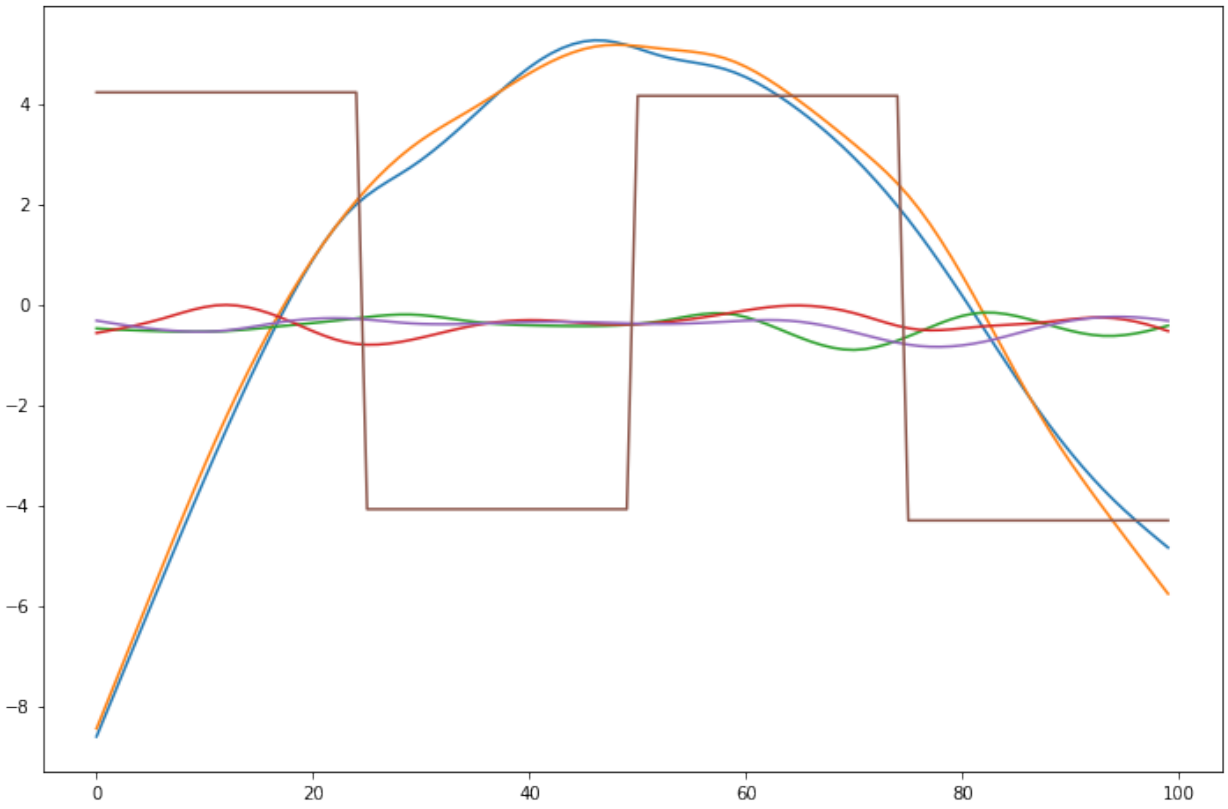
pyGAM is intuitive, modular, and adheres to a familiar API:

[30]:
```
from pygam import LogisticGAM, s, f
from pygam.datasets import toy_classification

X, y = toy_classification(return_X_y=True, n=5000)

gam = LogisticGAM(s(0) + s(1) + s(2) + s(3) + s(4) + f(5))
gam.fit(X, y)
```

[30]:
```
LogisticGAM(callbacks=[Deviance(), Diffs(), Accuracy()],
   fit_intercept=True, lam=[0.6, 0.6, 0.6, 0.6, 0.6, 0.6],
   max_iter=100,
   terms=s(0) + s(1) + s(2) + s(3) + s(4) + f(5) + intercept,
   tol=0.0001, verbose=False)
```

Since GAMs are additive, it is also super easy to visualize each individual **feature function**, `f_i(X_i)`. These feature functions describe the effect of each `X_i` on `y` individually while marginalizing out all other predictors:

[31]:
```
plt.figure()
for i, term in enumerate(gam.terms):
    if term.isintercept:
        continue
    plt.plot(gam.partial_dependence(term=i))
```



6.2.11 Current Features

Models

pyGAM comes with many models out-of-the-box:

- GAM (base class for constructing custom models)
- LinearGAM
- LogisticGAM
- GammaGAM
- PoissonGAM
- InvGaussGAM
- ExpectileGAM

Terms

- `l()` linear terms
- `s()` spline terms
- `f()` factor terms
- `te()` tensor products
- `intercept`

Distributions

- Normal
- Binomial
- Gamma
- Poisson
- Inverse Gaussian

Link Functions

Link functions take the distribution mean to the linear prediction. These are the canonical link functions for the above distributions:

- Identity
- Logit
- Inverse
- Log
- Inverse-squared

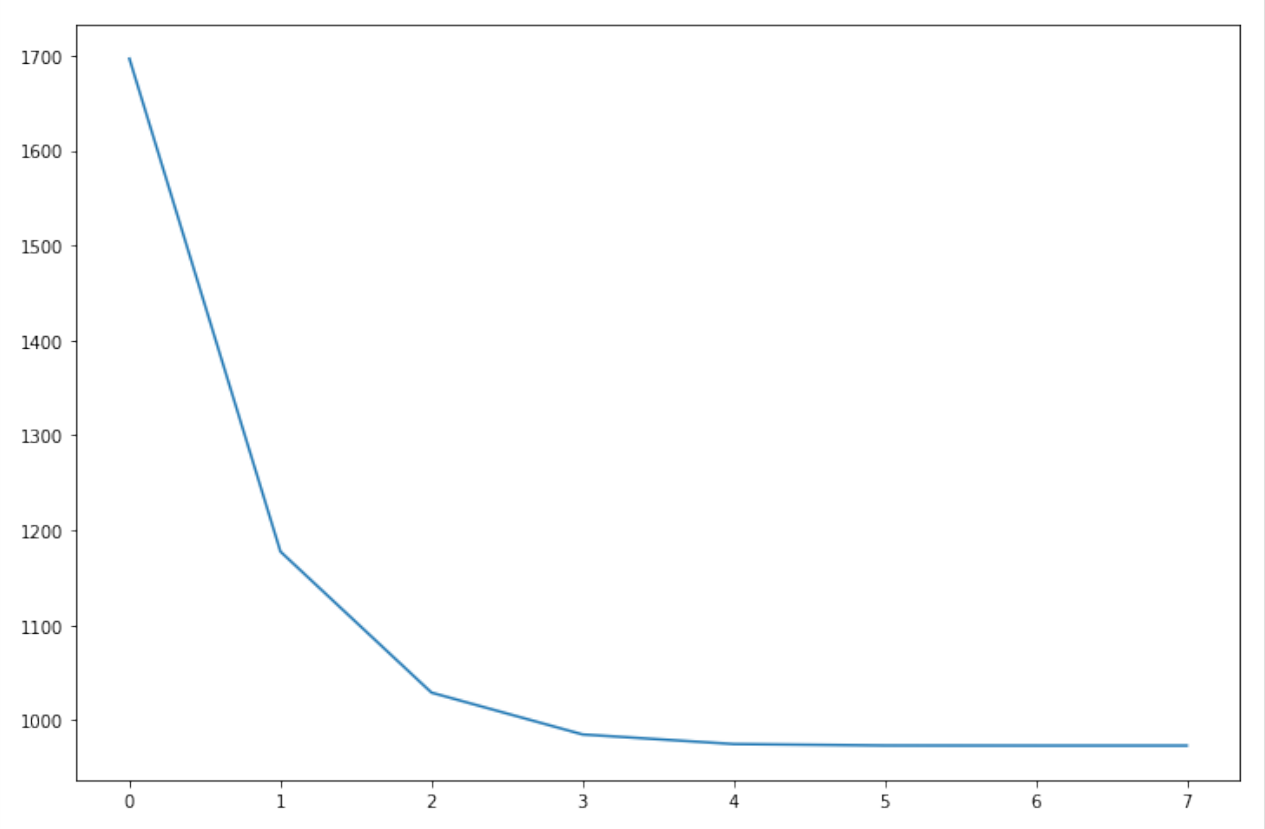
Callbacks

Callbacks are performed during each optimization iteration. It's also easy to write your own.

- deviance - model deviance
- diffs - differences of coefficient norm
- accuracy - model accuracy for LogisticGAM
- coef - coefficient logging

You can check a callback by inspecting:

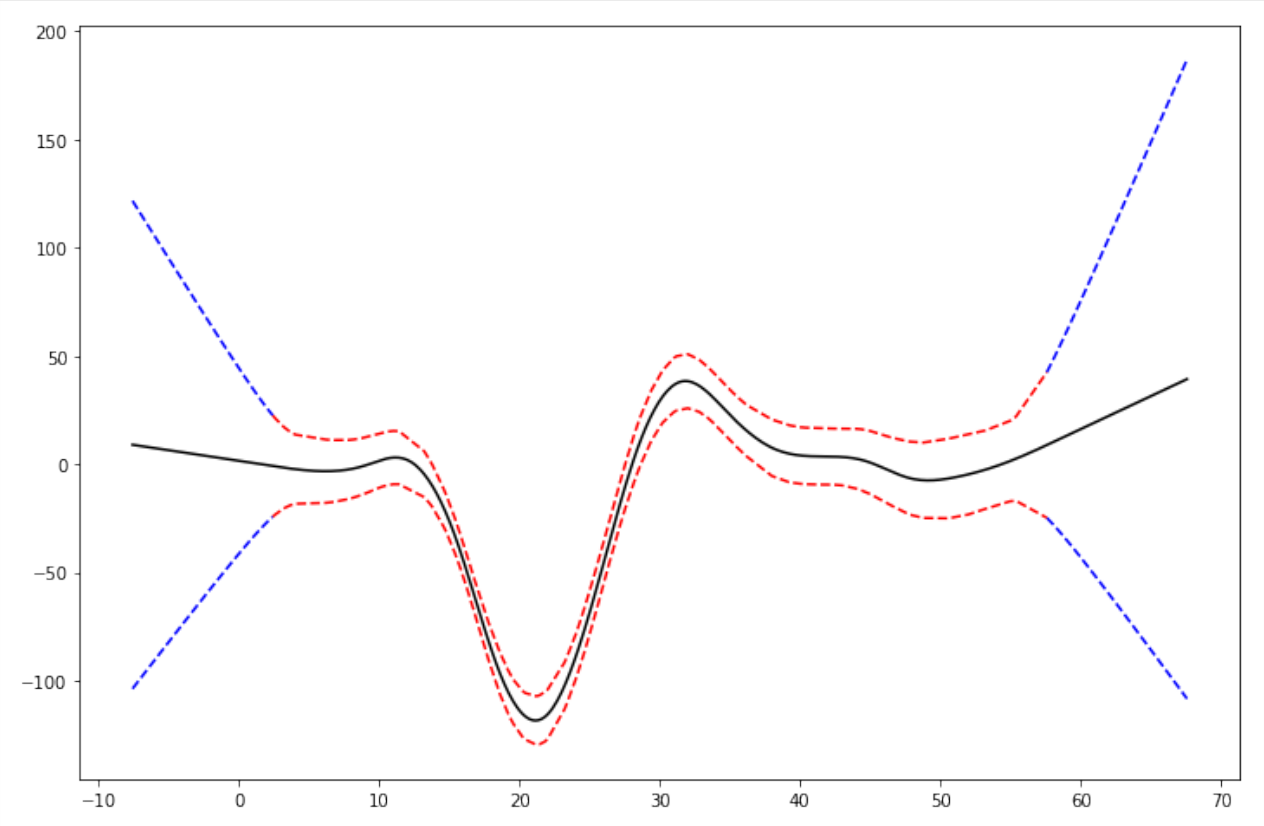
[32]:
```
_ = plt.plot(gam.logs_['deviance'])
```



Linear Extrapolation

[33]:
```
from pygam import LinearGAM
from pygam.datasets import mcycle

X, y = mcycle()

gam = LinearGAM()
gam.gridsearch(X, y)

XX = gam.generate_X_grid(term=0)

m = X.min()
M = X.max()
XX = np.linspace(m - 10, M + 10, 500)
Xl = np.linspace(m - 10, m, 50)
Xr = np.linspace(M, M + 10, 50)

plt.figure()

plt.plot(XX, gam.predict(XX), 'k')
plt.plot(Xl, gam.confidence_intervals(Xl), color='b', ls='--')
plt.plot(Xr, gam.confidence_intervals(Xr), color='b', ls='--')
_ = plt.plot(X, gam.confidence_intervals(X), color='r', ls='--')
```

```
100% (11 of 11) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
```



6.2.12 References

1. Simon N. Wood, 2006
   Generalized Additive Models: an introduction with R
2. Hastie, Tibshirani, Friedman
   The Elements of Statistical Learning
   http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf
3. James, Witten, Hastie and Tibshirani
   An Introduction to Statistical Learning
   http://www-bcf.usc.edu/~gareth/ISL/ISLR%20Sixth%20Printing.pdf
4. Paul Eilers & Brian Marx, 1996 Flexible Smoothing with B-splines and Penalties http://www.stat.washington.edu/courses/stat527/s13/readings/EilersMarx_StatSci_1996.pdf
5. Kim Larsen, 2015
   GAM: The Predictive Modeling Silver Bullet
   http://multithreaded.stitchfix.com/assets/files/gam.pdf
6. Deva Ramanan, 2008
   UCI Machine Learning: Notes on IRLS
   http://www.ics.uci.edu/~dramanan/teaching/ics273a_winter08/homework/irls_notes.pdf
7. Paul Eilers & Brian Marx, 2015
   International Biometric Society: A Crash Course on P-splines

http://www.ibschannel2015.nl/project/userfiles/Crash_course_handout.pdf

8. Keiding, Niels, 1991
   Age-specific incidence and prevalence: a statistical perspective

6.3 User API

6.3.1 Generalized Additive Model Classes

GAM

class pygam.pygam.**GAM**(*terms='auto', max_iter=100, tol=0.0001, distribution='normal', link='identity', callbacks=['deviance', 'diffs'], fit_intercept=True, verbose=False, **kwargs*)

Bases: `pygam.core.Core`, `pygam.terms.MetaTermMixin`

Generalized Additive Model

Parameters

- **terms** (*expression specifying terms to model, optional.*) – By default a univariate spline term will be allocated for each feature.

  For example:

  ```
  >>> GAM(s(0) + l(1) + f(2) + te(3, 4))
  ```

  will fit a spline term on feature 0, a linear term on feature 1, a factor term on feature 2, and a tensor term on features 3 and 4.
- **callbacks** (*list of str or list of CallBack objects, optional*) – Names of callback objects to call during the optimization loop.
- **distribution** (*str or Distribution object, optional*) – Distribution to use in the model.
- **link** (*str or Link object, optional*) – Link function to use in the model.
- **fit_intercept** (*bool, optional*) – Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function. Note: the intercept receives no smoothing penalty.
- **max_iter** (*int, optional*) – Maximum number of iterations allowed for the solver to converge.
- **tol** (*float, optional*) – Tolerance for stopping criteria.
- **verbose** (*bool, optional*) – whether to show pyGAM warnings.

coef_
Coefficient of the features in the decision function. If fit_intercept is True, then self.coef_[0] will contain the bias.

Type array, shape (n_classes, m_features)

statistics_
Dictionary containing model statistics like GCV/UBRE scores, AIC/c, parameter covariances, estimated degrees of freedom, etc.

Type dict

logs_
Dictionary containing the outputs of any callbacks at each optimization loop.

The logs are structured as `{callback: [...]}`

Type dict

References

Simon N. Wood, 2006 Generalized Additive Models: an introduction with R

Hastie, Tibshirani, Friedman The Elements of Statistical Learning http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf

Paul Eilers & Brian Marx, 2015 International Biometric Society: A Crash Course on P-splines http://www.ibschannel2015.nl/project/userfiles/Crash_course_handout.pdf

confidence_intervals(*X*, *width=0.95*, *quantiles=None*)
estimate confidence intervals for the model.

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – Input data matrix
- **width** (*float on [0,1], optional*) –
- **quantiles** (*array-like of floats in (0, 1), optional*) – Instead of specifying the prediciton width, one can specify the quantiles. So `width=.95` is equivalent to `quantiles=[.025, .975]`

Returns **intervals**

Return type np.array of shape (n_samples, 2 or len(quantiles))

Notes

Wood 2006, section 4.9 Confidence intervals based on section 4.8 rely on large sample results to deal with non-Gaussian distributions, and treat the smoothing parameters as fixed, when in reality they are estimated from the data.

deviance_residuals(*X*, *y*, *weights=None*, *scaled=False*)
method to compute the deviance residuals of the model

these are analogous to the residuals of an OLS.

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones
- **scaled** (*bool, optional*) – whether to scale the deviance by the (estimated) distribution scale

Returns **deviance_residuals** – with shape (n_samples,)

Return type np.array

fit(*X*, *y*, *weights=None*)
Fit the generalized additive model.

Parameters

- **X** (*array-like, shape (n_samples, m_features)*) – Training vectors.
- **y** (*array-like, shape (n_samples,)*) – Target values, ie integers in classification, real numbers in regression)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **self** – Returns fitted GAM object

Return type object

generate_X_grid(*term*, *n=100*, *meshgrid=False*)

create a nice grid of X data

array is sorted by feature and uniformly spaced, so the marginal and joint distributions are likely wrong

if term is >= 0, we generate n samples per feature, which results in n^deg samples, where deg is the degree of the interaction of the term

Parameters

- **term** (*int,*) – Which term to process.
- **n** (*int, optional*) – number of data points to create
- **meshgrid** (*bool, optional*) – Whether to return a meshgrid (useful for 3d plotting) or a feature matrix (useful for inference like partial predictions)

Returns

- *if meshgrid is False* – np.array of shape (n, n_features) where m is the number of (sub)terms in the requested (tensor)term.
- *else* – tuple of len m, where m is the number of (sub)terms in the requested (tensor)term.

  each element in the tuple contains a np.ndarray of size (n)^m

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

gridsearch(*X*, *y*, *weights=None*, *return_scores=False*, *keep_best=True*, *objective='auto'*, *progress=True*, ***param_grids*)

Performs a grid search over a space of parameters for a given objective

Warning: `gridsearch` is lazy and will not remove useless combinations from the search space, eg.

```
>>> n_splines=np.arange(5,10), fit_splines=[True, False]
```

will result in 10 loops, of which 5 are equivalent because `fit_splines = False`

Also, it is not recommended to search over a grid that alternates between known scales and unknown scales, as the scores of the candidate models will not be comparable.

Parameters

- **X** (*array-like*) – input data of shape (n_samples, m_features)
- **y** (*array-like*) – label data of shape (n_samples,)
- **weights** (*array-like shape (n_samples,), optional*) – sample weights

- **return_scores** (*boolean, optional*) – whether to return the hyperpamaters and score for each element in the grid
- **keep_best** (*boolean, optional*) – whether to keep the best GAM as self.
- **objective** (*{'auto', 'AIC', 'AICc', 'GCV', 'UBRE'}, optional*) – Metric to optimize. If *auto*, then grid search will optimize *GCV* for models with unknown scale and *UBRE* for models with known scale.
- **progress** (*bool, optional*) – whether to display a progress bar
- ****kwargs** – pairs of parameters and iterables of floats, or parameters and iterables of iterables of floats.

  If no parameter are specified, `lam=np.logspace(-3, 3, 11)` is used. This results in a 11 points, placed diagonally across lam space.

  If grid is iterable of iterables of floats, the outer iterable must have length `m_features`. the cartesian product of the subgrids in the grid will be tested.

  If grid is a 2d numpy array, each row of the array will be tested.

  The method will make a grid of all the combinations of the parameters and fit a GAM to each combination.

Returns

- if `return_scores=True` – model_scores: dict containing each fitted model as keys and corresponding objective scores as values
- *else* – self: ie possibly the newly fitted model

Examples

For a model with 4 terms, and where we expect 4 lam values, our search space for lam must have 4 dimensions.

We can search the space in 3 ways:

1. via cartesian product by specifying the grid as a list. our grid search will consider `11 ** 4` points:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = [lam] * 4
>>> gam.gridsearch(X, y, lam=lams)
```

2. directly by specifying the grid as a np.ndarray. This is useful for when the dimensionality of the search space is very large, and we would prefer to execute a randomized search:

```
>>> lams = np.exp(np.random.random(50, 4) * 6 - 3)
>>> gam.gridsearch(X, y, lam=lams)
```

3. copying grids for parameters with multiple dimensions. if we specify a 1D np.ndarray for lam, we are implicitly testing the space where all points have the same value

```
>>> gam.gridsearch(lam=np.logspace(-3, 3, 11))
```

is equivalent to:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = np.array([lam] * 4)
>>> gam.gridsearch(X, y, lam=lams)
```

loglikelihood(*X*, *y*, *weights=None*)
compute the log-likelihood of the dataset using the current model

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset
- **y** (*array-like of shape (n,)*) – containing target values
- **weights** (*array-like of shape (n,), optional*) – containing sample weights

Returns **log-likelihood** – containing log-likelihood scores

Return type np.array of shape (n,)

partial_dependence(*term*, *X=None*, *width=None*, *quantiles=None*, *meshgrid=False*)
Computes the term functions for the GAM and possibly their confidence intervals.

if both width=None and quantiles=None, then no confidence intervals are computed

Parameters

- **term** (*int, optional*) – Term for which to compute the partial dependence functions.
- **X** (*array-like with input data, optional*) – if *meshgrid=False*, then *X* should be an array-like of shape (n_samples, m_features).

  if *meshgrid=True*, then *X* should be a tuple containing an array for each feature in the term.

  if None, an equally spaced grid of points is generated.
- **width** (*float on (0, 1), optional*) – Width of the confidence interval.
- **quantiles** (*array-like of floats on (0, 1), optional*) – instead of specifying the prediciton width, one can specify the quantiles. so width=.95 is equivalent to quantiles=[.025, .975]. if None, defaults to width.
- **meshgrid** (*bool, whether to return and accept meshgrids.*) – Useful for creating outputs that are suitable for 3D plotting.

  Note, for simple terms with no interactions, the output of this function will be the same for `meshgrid=True` and `meshgrid=False`, but the inputs will need to be different.

Returns

- **pdeps** (*np.array of shape (n_samples,)*)
- **conf_intervals** (*list of length len(term)*) – containing np.arrays of shape (n_samples, 2 or len(quantiles))

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

See also:

generate_X_grid() for help creating meshgrids.

predict(*X*)
preduct expected value of target given model and input X often this is done via expected value of GAM given input X

Parameters **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset

Returns **y** – containing predicted values under the model

Return type np.array of shape (n_samples,)

predict_mu(*X*)
preduct expected value of target given model and input X

Parameters **X** (*array-like of shape (n_samples, m_features),*) – containing the input dataset

Returns **y** – containing expected values under the model

Return type np.array of shape (n_samples,)

sample(*X*, *y*, *quantity='y'*, *sample_at_X=None*, *weights=None*, *n_draws=100*, *n_bootstraps=5*, *objective='auto'*)
Simulate from the posterior of the coefficients and smoothing params.

Samples are drawn from the posterior of the coefficients and smoothing parameters given the response in an approximate way. The GAM must already be fitted before calling this method; if the model has not been fitted, then an exception is raised. Moreover, it is recommended that the model and its hyperparameters be chosen with *gridsearch* (with the parameter *keep_best=True*) before calling *sample*, so that the result of that gridsearch can be used to generate useful response data and so that the model's coefficients (and their covariance matrix) can be used as the first bootstrap sample.

These samples are drawn as follows. Details are in the reference below.

1. `n_bootstraps` many "bootstrap samples" of the response (`y`) are simulated by drawing random samples from the model's distribution evaluated at the expected values (`mu`) for each sample in `X`.

2. A copy of the model is fitted to each of those bootstrap samples of the response. The result is an approximation of the distribution over the smoothing parameter `lam` given the response data `y`.

3. Samples of the coefficients are simulated from a multivariate normal using the bootstrap samples of the coefficients and their covariance matrices.

Notes

A `gridsearch` is done `n_bootstraps` many times, so keep `n_bootstraps` small. Make `n_bootstraps < n_draws` to take advantage of the expensive bootstrap samples of the smoothing parameters.

Parameters

- **X** (*array of shape (n_samples, m_features)*) – empirical input data
- **y** (*array of shape (n_samples,)*) – empirical response vector
- **quantity** (*{'y', 'coef', 'mu'}, default: 'y'*) – What quantity to return pseudorandom samples of. If *sample_at_X* is not None and *quantity* is either *'y'* or *'mu'*, then samples are drawn at the values of *X* specified in *sample_at_X*.
- **sample_at_X** (*array of shape (n_samples_to_simulate, m_features) or*) –
- **optional** (*None,*) – Input data at which to draw new samples.

  Only applies for *quantity* equal to *'y'* or to *'mu*'. If *None*, then *sample_at_X* is replaced by *X*.

- **weights** (*np.array of shape (n_samples,)*) – sample weights
- **n_draws** (*positive int, optional (default=100)*) – The number of samples to draw from the posterior distribution of the coefficients and smoothing parameters
- **n_bootstraps** (*positive int, optional (default=5)*) – The number of bootstrap samples to draw from simulations of the response (from the already fitted model) to estimate the distribution of the smoothing parameters given the response data. If *n_bootstraps* is 1, then only the already fitted model's smoothing parameter is used, and the distribution over the smoothing parameters is not estimated using bootstrap sampling.
- **objective** (*string, optional (default='auto')* – metric to optimize in grid search. must be in ['AIC', 'AICc', 'GCV', 'UBRE', 'auto'] if 'auto', then grid search will optimize GCV for models with unknown scale and UBRE for models with known scale.

Returns

draws – Simulations of the given *quantity* using samples from the posterior distribution of the coefficients and smoothing parameter given the response data. Each row is a pseudorandom sample.

If *quantity == 'coef'*, then the number of columns of *draws* is the number of coefficients (*len(self.coef_)*).

Otherwise, the number of columns of *draws* is the number of rows of *sample_at_X* if *sample_at_X* is not *None* or else the number of rows of *X*.

Return type 2D array of length n_draws

References

Simon N. Wood, 2006. Generalized Additive Models: an introduction with R. Section 4.9.3 (pages 198–199) and Section 5.4.2 (page 256–257).

score(*X*, *y*, *weights=None*)

method to compute the explained deviance for a trained model for a given X data and y labels

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **explained deviancce score**

Return type np.array() (n_samples,)

summary()

produce a summary of the model statistics

Parameters **None** –

Returns

Return type None

LinearGAM

class pygam.pygam.**LinearGAM**(*terms='auto', max_iter=100, tol=0.0001, scale=None, callbacks=['deviance', 'diffs'], fit_intercept=True, verbose=False, **kwargs*)

Bases: *pygam.pygam.GAM*

Linear GAM

This is a GAM with a Normal error distribution, and an identity link.

Parameters

- **terms** (*expression specifying terms to model, optional.*) – By default a univariate spline term will be allocated for each feature.

  For example:

  ```
  >>> GAM(s(0) + l(1) + f(2) + te(3, 4))
  ```

  will fit a spline term on feature 0, a linear term on feature 1, a factor term on feature 2, and a tensor term on features 3 and 4.
- **callbacks** (*list of str or list of CallBack objects, optional*) – Names of callback objects to call during the optimization loop.
- **fit_intercept** (*bool, optional*) – Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function. Note: the intercept receives no smoothing penalty.
- **max_iter** (*int, optional*) – Maximum number of iterations allowed for the solver to converge.
- **tol** (*float, optional*) – Tolerance for stopping criteria.
- **verbose** (*bool, optional*) – whether to show pyGAM warnings.

coef_
Coefficient of the features in the decision function. If fit_intercept is True, then self.coef_[0] will contain the bias.

Type array, shape (n_classes, m_features)

statistics_
Dictionary containing model statistics like GCV/UBRE scores, AIC/c, parameter covariances, estimated degrees of freedom, etc.

Type dict

logs_
Dictionary containing the outputs of any callbacks at each optimization loop.

The logs are structured as `{callback: [...]}`

Type dict

References

Simon N. Wood, 2006 Generalized Additive Models: an introduction with R

Hastie, Tibshirani, Friedman The Elements of Statistical Learning http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf

Paul Eilers & Brian Marx, 2015 International Biometric Society: A Crash Course on P-splines http://www.ibschannel2015.nl/project/userfiles/Crash_course_handout.pdf

confidence_intervals(*X*, *width=0.95*, *quantiles=None*)
estimate confidence intervals for the model.

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – Input data matrix
- **width** (*float on [0,1], optional*) –
- **quantiles** (*array-like of floats in (0, 1), optional*) – Instead of specifying the prediciton width, one can specify the quantiles. So `width=.95` is equivalent to `quantiles=[.025, .975]`

Returns **intervals**

Return type np.array of shape (n_samples, 2 or len(quantiles))

Notes

Wood 2006, section 4.9 Confidence intervals based on section 4.8 rely on large sample results to deal with non-Gaussian distributions, and treat the smoothing parameters as fixed, when in reality they are estimated from the data.

deviance_residuals(*X*, *y*, *weights=None*, *scaled=False*)
method to compute the deviance residuals of the model

these are analogous to the residuals of an OLS.

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones
- **scaled** (*bool, optional*) – whether to scale the deviance by the (estimated) distribution scale

Returns **deviance_residuals** – with shape (n_samples,)

Return type np.array

fit(*X*, *y*, *weights=None*)
Fit the generalized additive model.

Parameters

- **X** (*array-like, shape (n_samples, m_features)*) – Training vectors.
- **y** (*array-like, shape (n_samples,)*) – Target values, ie integers in classification, real numbers in regression)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **self** – Returns fitted GAM object

Return type object

generate_X_grid(*term*, *n=100*, *meshgrid=False*)
create a nice grid of X data

array is sorted by feature and uniformly spaced, so the marginal and joint distributions are likely wrong

if term is >= 0, we generate n samples per feature, which results in n^deg samples, where deg is the degree of the interaction of the term

Parameters

- **term** (*int,*) – Which term to process.
- **n** (*int, optional*) – number of data points to create
- **meshgrid** (*bool, optional*) – Whether to return a meshgrid (useful for 3d plotting) or a feature matrix (useful for inference like partial predictions)

Returns

- *if meshgrid is False* – np.array of shape (n, n_features) where m is the number of (sub)terms in the requested (tensor)term.
- *else* – tuple of len m, where m is the number of (sub)terms in the requested (tensor)term.

  each element in the tuple contains a np.ndarray of size (n)^m

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

get_params(*deep=False*)
returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

gridsearch(*X*, *y*, *weights=None*, *return_scores=False*, *keep_best=True*, *objective='auto'*, *progress=True*, ***param_grids*)
Performs a grid search over a space of parameters for a given objective

Warning: `gridsearch` is lazy and will not remove useless combinations from the search space, eg.

```
>>> n_splines=np.arange(5,10), fit_splines=[True, False]
```

will result in 10 loops, of which 5 are equivalent because `fit_splines = False`

Also, it is not recommended to search over a grid that alternates between known scales and unknown scales, as the scores of the candidate models will not be comparable.

Parameters

- **X** (*array-like*) – input data of shape (n_samples, m_features)
- **y** (*array-like*) – label data of shape (n_samples,)
- **weights** (*array-like shape (n_samples,), optional*) – sample weights
- **return_scores** (*boolean, optional*) – whether to return the hyperpamaters and score for each element in the grid
- **keep_best** (*boolean, optional*) – whether to keep the best GAM as self.

- **objective** (*{'auto', 'AIC', 'AICc', 'GCV', 'UBRE'}, optional*) – Metric to optimize. If *auto*, then grid search will optimize *GCV* for models with unknown scale and *UBRE* for models with known scale.
- **progress** (*bool, optional*) – whether to display a progress bar
- ****kwargs** – pairs of parameters and iterables of floats, or parameters and iterables of iterables of floats.

  If no parameter are specified, `lam=np.logspace(-3, 3, 11)` is used. This results in a 11 points, placed diagonally across lam space.

  If grid is iterable of iterables of floats, the outer iterable must have length `m_features`. the cartesian product of the subgrids in the grid will be tested.

  If grid is a 2d numpy array, each row of the array will be tested.

  The method will make a grid of all the combinations of the parameters and fit a GAM to each combination.

Returns

- if `return_scores=True` – model_scores: dict containing each fitted model as keys and corresponding objective scores as values
- *else* – self: ie possibly the newly fitted model

Examples

For a model with 4 terms, and where we expect 4 lam values, our search space for lam must have 4 dimensions.

We can search the space in 3 ways:

1. via cartesian product by specifying the grid as a list. our grid search will consider `11 ** 4` points:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = [lam] * 4
>>> gam.gridsearch(X, y, lam=lams)
```

2. directly by specifying the grid as a np.ndarray. This is useful for when the dimensionality of the search space is very large, and we would prefer to execute a randomized search:

```
>>> lams = np.exp(np.random.random(50, 4) * 6 - 3)
>>> gam.gridsearch(X, y, lam=lams)
```

3. copying grids for parameters with multiple dimensions. if we specify a 1D np.ndarray for lam, we are implicitly testing the space where all points have the same value

```
>>> gam.gridsearch(lam=np.logspace(-3, 3, 11))
```

is equivalent to:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = np.array([lam] * 4)
>>> gam.gridsearch(X, y, lam=lams)
```

loglikelihood(*X*, *y*, *weights=None*)

compute the log-likelihood of the dataset using the current model

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset
- **y** (*array-like of shape (n,)*) – containing target values
- **weights** (*array-like of shape (n,), optional*) – containing sample weights

Returns **log-likelihood** – containing log-likelihood scores

Return type np.array of shape (n,)

partial_dependence(*term*, *X=None*, *width=None*, *quantiles=None*, *meshgrid=False*)

Computes the term functions for the GAM and possibly their confidence intervals.

if both width=None and quantiles=None, then no confidence intervals are computed

Parameters

- **term** (*int, optional*) – Term for which to compute the partial dependence functions.
- **X** (*array-like with input data, optional*) – if *meshgrid=False*, then *X* should be an array-like of shape (n_samples, m_features).

  if *meshgrid=True*, then *X* should be a tuple containing an array for each feature in the term.

  if None, an equally spaced grid of points is generated.
- **width** (*float on (0, 1), optional*) – Width of the confidence interval.
- **quantiles** (*array-like of floats on (0, 1), optional*) – instead of specifying the prediciton width, one can specify the quantiles. so width=.95 is equivalent to quantiles=[.025, .975]. if None, defaults to width.
- **meshgrid** (*bool, whether to return and accept meshgrids.*) – Useful for creating outputs that are suitable for 3D plotting.

  Note, for simple terms with no interactions, the output of this function will be the same for `meshgrid=True` and `meshgrid=False`, but the inputs will need to be different.

Returns

- **pdeps** (*np.array of shape (n_samples,)*)
- **conf_intervals** (*list of length len(term)*) – containing np.arrays of shape (n_samples, 2 or len(quantiles))

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

See also:

generate_X_grid() for help creating meshgrids.

predict(*X*)

preduct expected value of target given model and input X often this is done via expected value of GAM given input X

Parameters **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset

Returns **y** – containing predicted values under the model

Return type np.array of shape (n_samples,)

predict_mu(*X*)
preduct expected value of target given model and input X

Parameters **X** (*array-like of shape (n_samples, m_features),*) – containing the input dataset

Returns **y** – containing expected values under the model

Return type np.array of shape (n_samples,)

prediction_intervals(*X*, *width=0.95*, *quantiles=None*)
estimate prediction intervals for LinearGAM

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – input data matrix
- **width** (*float on [0,1], optional (default=0.95*) –
- **quantiles** (*array-like of floats in [0, 1], default: None)*) – instead of specifying the prediciton width, one can specify the quantiles. so width=.95 is equivalent to quantiles=[.025, .975]

Returns **intervals**

Return type np.array of shape (n_samples, 2 or len(quantiles))

sample(*X*, *y*, *quantity='y'*, *sample_at_X=None*, *weights=None*, *n_draws=100*, *n_bootstraps=5*, *objective='auto'*)
Simulate from the posterior of the coefficients and smoothing params.

Samples are drawn from the posterior of the coefficients and smoothing parameters given the response in an approximate way. The GAM must already be fitted before calling this method; if the model has not been fitted, then an exception is raised. Moreover, it is recommended that the model and its hyperparameters be chosen with *gridsearch* (with the parameter *keep_best=True*) before calling *sample*, so that the result of that gridsearch can be used to generate useful response data and so that the model's coefficients (and their covariance matrix) can be used as the first bootstrap sample.

These samples are drawn as follows. Details are in the reference below.

1. `n_bootstraps` many "bootstrap samples" of the response (`y`) are simulated by drawing random samples from the model's distribution evaluated at the expected values (`mu`) for each sample in `X`.

2. A copy of the model is fitted to each of those bootstrap samples of the response. The result is an approximation of the distribution over the smoothing parameter `lam` given the response data `y`.

3. Samples of the coefficients are simulated from a multivariate normal using the bootstrap samples of the coefficients and their covariance matrices.

Notes

A `gridsearch` is done `n_bootstraps` many times, so keep `n_bootstraps` small. Make `n_bootstraps < n_draws` to take advantage of the expensive bootstrap samples of the smoothing parameters.

Parameters

- **X** (*array of shape (n_samples, m_features)*) – empirical input data
- **y** (*array of shape (n_samples,)*) – empirical response vector

- **quantity** (*{'y', 'coef', 'mu'}, default: 'y'*) – What quantity to return pseudorandom samples of. If *sample_at_X* is not None and *quantity* is either *'y'* or *'mu'*, then samples are drawn at the values of *X* specified in *sample_at_X*.
- **sample_at_X** (*array of shape (n_samples_to_simulate, m_features) or*) –
- **optional** (*None,*) – Input data at which to draw new samples.

  Only applies for *quantity* equal to *'y'* or to *'mu'*. If *None*, then *sample_at_X* is replaced by *X*.
- **weights** (*np.array of shape (n_samples,)*) – sample weights
- **n_draws** (*positive int, optional (default=100)*) – The number of samples to draw from the posterior distribution of the coefficients and smoothing parameters
- **n_bootstraps** (*positive int, optional (default=5)*) – The number of bootstrap samples to draw from simulations of the response (from the already fitted model) to estimate the distribution of the smoothing parameters given the response data. If *n_bootstraps* is 1, then only the already fitted model's smoothing parameter is used, and the distribution over the smoothing parameters is not estimated using bootstrap sampling.
- **objective** (*string, optional (default='auto'*) – metric to optimize in grid search. must be in ['AIC', 'AICc', 'GCV', 'UBRE', 'auto'] if 'auto', then grid search will optimize GCV for models with unknown scale and UBRE for models with known scale.

Returns

draws – Simulations of the given *quantity* using samples from the posterior distribution of the coefficients and smoothing parameter given the response data. Each row is a pseudorandom sample.

If *quantity == 'coef'*, then the number of columns of *draws* is the number of coefficients (*len(self.coef_)*).

Otherwise, the number of columns of *draws* is the number of rows of *sample_at_X* if *sample_at_X* is not *None* or else the number of rows of *X*.

Return type 2D array of length n_draws

References

Simon N. Wood, 2006. Generalized Additive Models: an introduction with R. Section 4.9.3 (pages 198–199) and Section 5.4.2 (page 256–257).

score (*X*, *y*, *weights=None*)

method to compute the explained deviance for a trained model for a given X data and y labels

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **explained deviancce score**

Return type np.array() (n_samples,)

set_params(*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters
- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have
- ****parameters** (*paramters to set*) –

Returns

Return type self

summary()
produce a summary of the model statistics

Parameters **None** –

Returns

Return type None

GammaGAM

class pygam.pygam.**GammaGAM**(*terms='auto', max_iter=100, tol=0.0001, scale=None, callbacks=['deviance', 'diffs'], fit_intercept=True, verbose=False, **kwargs*)
Bases: *pygam.pygam.GAM*

Gamma GAM

This is a GAM with a Gamma error distribution, and a log link.

NB Although canonical link function for the Gamma GLM is the inverse link, this function can create problems for numerical software because it becomes difficult to enforce the requirement that the mean of the Gamma distribution be positive. The log link guarantees this.

If you need to use the inverse link function, simply construct a custom GAM:

```
>>> from pygam import GAM
>>> gam = GAM(distribution='gamma', link='inverse')
```

Parameters

- **terms** (*expression specifying terms to model, optional.*) – By default a univariate spline term will be allocated for each feature.

  For example:

  ```
  >>> GAM(s(0) + l(1) + f(2) + te(3, 4))
  ```

  will fit a spline term on feature 0, a linear term on feature 1, a factor term on feature 2, and a tensor term on features 3 and 4.
- **callbacks** (*list of str or list of CallBack objects, optional*) – Names of callback objects to call during the optimization loop.
- **fit_intercept** (*bool, optional*) – Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function. Note: the intercept receives no smoothing penalty.

- **max_iter** (*int, optional*) – Maximum number of iterations allowed for the solver to converge.
- **tol** (*float, optional*) – Tolerance for stopping criteria.
- **verbose** (*bool, optional*) – whether to show pyGAM warnings.

coef_
Coefficient of the features in the decision function. If fit_intercept is True, then self.coef_[0] will contain the bias.

Type array, shape (n_classes, m_features)

statistics_
Dictionary containing model statistics like GCV/UBRE scores, AIC/c, parameter covariances, estimated degrees of freedom, etc.

Type dict

logs_
Dictionary containing the outputs of any callbacks at each optimization loop.

The logs are structured as `{callback: [...]}`

Type dict

References

Simon N. Wood, 2006 Generalized Additive Models: an introduction with R

Hastie, Tibshirani, Friedman The Elements of Statistical Learning http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf

Paul Eilers & Brian Marx, 2015 International Biometric Society: A Crash Course on P-splines http://www.ibschannel2015.nl/project/userfiles/Crash_course_handout.pdf

confidence_intervals(*X*, *width=0.95*, *quantiles=None*)
estimate confidence intervals for the model.

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – Input data matrix
- **width** (*float on [0,1], optional*) –
- **quantiles** (*array-like of floats in (0, 1), optional*) – Instead of specifying the prediciton width, one can specify the quantiles. So `width=.95` is equivalent to `quantiles=[.025, .975]`

Returns **intervals**

Return type np.array of shape (n_samples, 2 or len(quantiles))

Notes

Wood 2006, section 4.9 Confidence intervals based on section 4.8 rely on large sample results to deal with non-Gaussian distributions, and treat the smoothing parameters as fixed, when in reality they are estimated from the data.

deviance_residuals(*X*, *y*, *weights=None*, *scaled=False*)
method to compute the deviance residuals of the model

these are analogous to the residuals of an OLS.

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones
- **scaled** (*bool, optional*) – whether to scale the deviance by the (estimated) distribution scale

Returns **deviance_residuals** – with shape (n_samples,)

Return type np.array

fit(*X*, *y*, *weights=None*)
Fit the generalized additive model.

Parameters

- **X** (*array-like, shape (n_samples, m_features)*) – Training vectors.
- **y** (*array-like, shape (n_samples,)*) – Target values, ie integers in classification, real numbers in regression)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **self** – Returns fitted GAM object

Return type object

generate_X_grid(*term*, *n=100*, *meshgrid=False*)
create a nice grid of X data

array is sorted by feature and uniformly spaced, so the marginal and joint distributions are likely wrong

if term is >= 0, we generate n samples per feature, which results in n^deg samples, where deg is the degree of the interaction of the term

Parameters

- **term** (*int,*) – Which term to process.
- **n** (*int, optional*) – number of data points to create
- **meshgrid** (*bool, optional*) – Whether to return a meshgrid (useful for 3d plotting) or a feature matrix (useful for inference like partial predictions)

Returns

- *if meshgrid is False* – np.array of shape (n, n_features) where m is the number of (sub)terms in the requested (tensor)term.
- *else* – tuple of len m, where m is the number of (sub)terms in the requested (tensor)term.

  each element in the tuple contains a np.ndarray of size (n)^m

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

get_params(*deep=False*)
returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

gridsearch(*X*, *y*, *weights=None*, *return_scores=False*, *keep_best=True*, *objective='auto'*, *progress=True*, ***param_grids*)
Performs a grid search over a space of parameters for a given objective

Warning: `gridsearch` is lazy and will not remove useless combinations from the search space, eg.

```
>>> n_splines=np.arange(5,10), fit_splines=[True, False]
```

will result in 10 loops, of which 5 are equivalent because `fit_splines = False`

Also, it is not recommended to search over a grid that alternates between known scales and unknown scales, as the scores of the candidate models will not be comparable.

Parameters

- **X** (*array-like*) – input data of shape (n_samples, m_features)
- **y** (*array-like*) – label data of shape (n_samples,)
- **weights** (*array-like shape (n_samples,), optional*) – sample weights
- **return_scores** (*boolean, optional*) – whether to return the hyperpamaters and score for each element in the grid
- **keep_best** (*boolean, optional*) – whether to keep the best GAM as self.
- **objective** (*{'auto', 'AIC', 'AICc', 'GCV', 'UBRE'}, optional*) – Metric to optimize. If *auto*, then grid search will optimize *GCV* for models with unknown scale and *UBRE* for models with known scale.
- **progress** (*bool, optional*) – whether to display a progress bar
- ****kwargs** – pairs of parameters and iterables of floats, or parameters and iterables of iterables of floats.

  If no parameter are specified, `lam=np.logspace(-3, 3, 11)` is used. This results in a 11 points, placed diagonally across lam space.

  If grid is iterable of iterables of floats, the outer iterable must have length `m_features`. the cartesian product of the subgrids in the grid will be tested.

  If grid is a 2d numpy array, each row of the array will be tested.

  The method will make a grid of all the combinations of the parameters and fit a GAM to each combination.

Returns

- if `return_scores=True` – model_scores: dict containing each fitted model as keys and corresponding objective scores as values
- *else* – self: ie possibly the newly fitted model

Examples

For a model with 4 terms, and where we expect 4 lam values, our search space for lam must have 4 dimensions.

We can search the space in 3 ways:

1. via cartesian product by specifying the grid as a list. our grid search will consider `11 ** 4` points:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = [lam] * 4
>>> gam.gridsearch(X, y, lam=lams)
```

2. directly by specifying the grid as a np.ndarray. This is useful for when the dimensionality of the search space is very large, and we would prefer to execute a randomized search:

```
>>> lams = np.exp(np.random.random(50, 4) * 6 - 3)
>>> gam.gridsearch(X, y, lam=lams)
```

3. copying grids for parameters with multiple dimensions. if we specify a 1D np.ndarray for lam, we are implicitly testing the space where all points have the same value

```
>>> gam.gridsearch(lam=np.logspace(-3, 3, 11))
```

is equivalent to:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = np.array([lam] * 4)
>>> gam.gridsearch(X, y, lam=lams)
```

loglikelihood(*X*, *y*, *weights=None*)
compute the log-likelihood of the dataset using the current model

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset
- **y** (*array-like of shape (n,)*) – containing target values
- **weights** (*array-like of shape (n,), optional*) – containing sample weights

Returns **log-likelihood** – containing log-likelihood scores

Return type np.array of shape (n,)

partial_dependence(*term*, *X=None*, *width=None*, *quantiles=None*, *meshgrid=False*)
Computes the term functions for the GAM and possibly their confidence intervals.

if both width=None and quantiles=None, then no confidence intervals are computed

Parameters

- **term** (*int, optional*) – Term for which to compute the partial dependence functions.
- **X** (*array-like with input data, optional*) – if *meshgrid=False*, then *X* should be an array-like of shape (n_samples, m_features).

  if *meshgrid=True*, then *X* should be a tuple containing an array for each feature in the term.

if None, an equally spaced grid of points is generated.

- **width** (*float on (0, 1), optional*) – Width of the confidence interval.
- **quantiles** (*array-like of floats on (0, 1), optional*) – instead of specifying the prediciton width, one can specify the quantiles. so width=.95 is equivalent to quantiles=[.025, .975]. if None, defaults to width.
- **meshgrid** (*bool, whether to return and accept meshgrids.*) – Useful for creating outputs that are suitable for 3D plotting.

  Note, for simple terms with no interactions, the output of this function will be the same for `meshgrid=True` and `meshgrid=False`, but the inputs will need to be different.

Returns

- **pdeps** (*np.array of shape (n_samples,)*)
- **conf_intervals** (*list of length len(term)*) – containing np.arrays of shape (n_samples, 2 or len(quantiles))

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

See also:

generate_X_grid() for help creating meshgrids.

predict(*X*)

preduct expected value of target given model and input X often this is done via expected value of GAM given input X

Parameters **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset

Returns **y** – containing predicted values under the model

Return type np.array of shape (n_samples,)

predict_mu(*X*)

preduct expected value of target given model and input X

Parameters **X** (*array-like of shape (n_samples, m_features),*) – containing the input dataset

Returns **y** – containing expected values under the model

Return type np.array of shape (n_samples,)

sample(*X*, *y*, *quantity='y'*, *sample_at_X=None*, *weights=None*, *n_draws=100*, *n_bootstraps=5*, *objective='auto'*)

Simulate from the posterior of the coefficients and smoothing params.

Samples are drawn from the posterior of the coefficients and smoothing parameters given the response in an approximate way. The GAM must already be fitted before calling this method; if the model has not been fitted, then an exception is raised. Moreover, it is recommended that the model and its hyperparameters be chosen with *gridsearch* (with the parameter *keep_best=True*) before calling *sample*, so that the result of that gridsearch can be used to generate useful response data and so that the model's coefficients (and their covariance matrix) can be used as the first bootstrap sample.

These samples are drawn as follows. Details are in the reference below.

1. `n_bootstraps` many "bootstrap samples" of the response (`y`) are simulated by drawing random samples from the model's distribution evaluated at the expected values (`mu`) for each sample in `X`.

2. A copy of the model is fitted to each of those bootstrap samples of the response. The result is an approximation of the distribution over the smoothing parameter `lam` given the response data `y`.

3. Samples of the coefficients are simulated from a multivariate normal using the bootstrap samples of the coefficients and their covariance matrices.

Notes

A `gridsearch` is done `n_bootstraps` many times, so keep `n_bootstraps` small. Make `n_bootstraps < n_draws` to take advantage of the expensive bootstrap samples of the smoothing parameters.

Parameters

- **X** (*array of shape (n_samples, m_features)*) – empirical input data
- **y** (*array of shape (n_samples,)*) – empirical response vector
- **quantity** (*{'y', 'coef', 'mu'}, default: 'y'*) – What quantity to return pseudorandom samples of. If *sample_at_X* is not None and *quantity* is either *'y'* or *'mu'*, then samples are drawn at the values of *X* specified in *sample_at_X*.
- **sample_at_X** (*array of shape (n_samples_to_simulate, m_features) or*) –
- **optional** (*None,*) – Input data at which to draw new samples.

  Only applies for *quantity* equal to *'y'* or to *'mu*'. If *None*, then *sample_at_X* is replaced by *X*.
- **weights** (*np.array of shape (n_samples,)*) – sample weights
- **n_draws** (*positive int, optional (default=100)*) – The number of samples to draw from the posterior distribution of the coefficients and smoothing parameters
- **n_bootstraps** (*positive int, optional (default=5)*) – The number of bootstrap samples to draw from simulations of the response (from the already fitted model) to estimate the distribution of the smoothing parameters given the response data. If *n_bootstraps* is 1, then only the already fitted model's smoothing parameter is used, and the distribution over the smoothing parameters is not estimated using bootstrap sampling.
- **objective** (*string, optional (default='auto')* – metric to optimize in grid search. must be in ['AIC', 'AICc', 'GCV', 'UBRE', 'auto'] if 'auto', then grid search will optimize GCV for models with unknown scale and UBRE for models with known scale.

Returns

draws – Simulations of the given *quantity* using samples from the posterior distribution of the coefficients and smoothing parameter given the response data. Each row is a pseudorandom sample.

If *quantity == 'coef'*, then the number of columns of *draws* is the number of coefficients (*len(self.coef_)*).

Otherwise, the number of columns of *draws* is the number of rows of *sample_at_X* if *sample_at_X* is not *None* or else the number of rows of *X*.

Return type 2D array of length n_draws

References

Simon N. Wood, 2006. Generalized Additive Models: an introduction with R. Section 4.9.3 (pages 198–199) and Section 5.4.2 (page 256–257).

score(*X*, *y*, *weights=None*)
method to compute the explained deviance for a trained model for a given X data and y labels

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **explained deviancce score**

Return type np.array() (n_samples,)

set_params(*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters
- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have
- ****parameters** (*paramters to set*) –

Returns

Return type self

summary()
produce a summary of the model statistics

Parameters **None** –

Returns

Return type None

InvGaussGAM

class pygam.pygam.**InvGaussGAM**(*terms='auto', max_iter=100, tol=0.0001, scale=None, callbacks=['deviance', 'diffs'], fit_intercept=True, verbose=False, **kwargs*)

Bases: *pygam.pygam.GAM*

Inverse Gaussian GAM

This is a GAM with a Inverse Gaussian error distribution, and a log link.

NB Although canonical link function for the Inverse Gaussian GLM is the inverse squared link, this function can create problems for numerical software because it becomes difficult to enforce the requirement that the mean of the Inverse Gaussian distribution be positive. The log link guarantees this.

If you need to use the inverse squared link function, simply construct a custom GAM:

```
>>> from pygam import GAM
>>> gam = GAM(distribution='inv_gauss', link='inv_squared')
```

Parameters

- **terms** (*expression specifying terms to model, optional.*) – By default a univariate spline term will be allocated for each feature.

  For example:

  ```
  >>> GAM(s(0) + l(1) + f(2) + te(3, 4))
  ```

  will fit a spline term on feature 0, a linear term on feature 1, a factor term on feature 2, and a tensor term on features 3 and 4.

- **callbacks** (*list of str or list of CallBack objects, optional*) – Names of callback objects to call during the optimization loop.
- **fit_intercept** (*bool, optional*) – Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function. Note: the intercept receives no smoothing penalty.
- **max_iter** (*int, optional*) – Maximum number of iterations allowed for the solver to converge.
- **tol** (*float, optional*) – Tolerance for stopping criteria.
- **verbose** (*bool, optional*) – whether to show pyGAM warnings.

coef_
Coefficient of the features in the decision function. If fit_intercept is True, then self.coef_[0] will contain the bias.

Type array, shape (n_classes, m_features)

statistics_
Dictionary containing model statistics like GCV/UBRE scores, AIC/c, parameter covariances, estimated degrees of freedom, etc.

Type dict

logs_
Dictionary containing the outputs of any callbacks at each optimization loop.

The logs are structured as `{callback: [...]}`

Type dict

References

Simon N. Wood, 2006 Generalized Additive Models: an introduction with R

Hastie, Tibshirani, Friedman The Elements of Statistical Learning http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf

Paul Eilers & Brian Marx, 2015 International Biometric Society: A Crash Course on P-splines http://www.ibschannel2015.nl/project/userfiles/Crash_course_handout.pdf

confidence_intervals(*X*, *width=0.95*, *quantiles=None*)
estimate confidence intervals for the model.

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – Input data matrix
- **width** (*float on [0,1], optional*) –
- **quantiles** (*array-like of floats in (0, 1), optional*) – Instead of specifying the prediciton width, one can specify the quantiles. So `width=.95` is equivalent to `quantiles=[.025, .975]`

Returns **intervals**

Return type np.array of shape (n_samples, 2 or len(quantiles))

Notes

Wood 2006, section 4.9 Confidence intervals based on section 4.8 rely on large sample results to deal with non-Gaussian distributions, and treat the smoothing parameters as fixed, when in reality they are estimated from the data.

deviance_residuals (*X*, *y*, *weights=None*, *scaled=False*)
method to compute the deviance residuals of the model

these are analogous to the residuals of an OLS.

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones
- **scaled** (*bool, optional*) – whether to scale the deviance by the (estimated) distribution scale

Returns **deviance_residuals** – with shape (n_samples,)

Return type np.array

fit (*X*, *y*, *weights=None*)
Fit the generalized additive model.

Parameters

- **X** (*array-like, shape (n_samples, m_features)*) – Training vectors.
- **y** (*array-like, shape (n_samples,)*) – Target values, ie integers in classification, real numbers in regression)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **self** – Returns fitted GAM object

Return type object

generate_X_grid (*term*, *n=100*, *meshgrid=False*)
create a nice grid of X data

array is sorted by feature and uniformly spaced, so the marginal and joint distributions are likely wrong

if term is >= 0, we generate n samples per feature, which results in n^deg samples, where deg is the degree of the interaction of the term

Parameters

- **term** (*int,*) – Which term to process.
- **n** (*int, optional*) – number of data points to create
- **meshgrid** (*bool, optional*) – Whether to return a meshgrid (useful for 3d plotting) or a feature matrix (useful for inference like partial predictions)

Returns

- *if meshgrid is False* – np.array of shape (n, n_features) where m is the number of (sub)terms in the requested (tensor)term.
- *else* – tuple of len m, where m is the number of (sub)terms in the requested (tensor)term.

  each element in the tuple contains a np.ndarray of size (n)^m

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

get_params (*deep=False*)
returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

gridsearch (*X, y, weights=None, return_scores=False, keep_best=True, objective='auto', progress=True, **param_grids*)
Performs a grid search over a space of parameters for a given objective

Warning: `gridsearch` is lazy and will not remove useless combinations from the search space, eg.

```
>>> n_splines=np.arange(5,10), fit_splines=[True, False]
```

will result in 10 loops, of which 5 are equivalent because `fit_splines = False`

Also, it is not recommended to search over a grid that alternates between known scales and unknown scales, as the scores of the candidate models will not be comparable.

Parameters

- **X** (*array-like*) – input data of shape (n_samples, m_features)
- **y** (*array-like*) – label data of shape (n_samples,)
- **weights** (*array-like shape (n_samples,), optional*) – sample weights
- **return_scores** (*boolean, optional*) – whether to return the hyperpamaters and score for each element in the grid
- **keep_best** (*boolean, optional*) – whether to keep the best GAM as self.
- **objective** (*{'auto', 'AIC', 'AICc', 'GCV', 'UBRE'}, optional*) – Metric to optimize. If *auto*, then grid search will optimize *GCV* for models with unknown scale and *UBRE* for models with known scale.
- **progress** (*bool, optional*) – whether to display a progress bar
- ****kwargs** – pairs of parameters and iterables of floats, or parameters and iterables of iterables of floats.

If no parameter are specified, `lam=np.logspace(-3, 3, 11)` is used. This results in a 11 points, placed diagonally across lam space.

If grid is iterable of iterables of floats, the outer iterable must have length `m_features`. the cartesian product of the subgrids in the grid will be tested.

If grid is a 2d numpy array, each row of the array will be tested.

The method will make a grid of all the combinations of the parameters and fit a GAM to each combination.

Returns

- if `return_scores=True` – model_scores: dict containing each fitted model as keys and corresponding objective scores as values
- *else* – self: ie possibly the newly fitted model

Examples

For a model with 4 terms, and where we expect 4 lam values, our search space for lam must have 4 dimensions.

We can search the space in 3 ways:

1. via cartesian product by specifying the grid as a list. our grid search will consider `11 ** 4` points:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = [lam] * 4
>>> gam.gridsearch(X, y, lam=lams)
```

2. directly by specifying the grid as a np.ndarray. This is useful for when the dimensionality of the search space is very large, and we would prefer to execute a randomized search:

```
>>> lams = np.exp(np.random.random(50, 4) * 6 - 3)
>>> gam.gridsearch(X, y, lam=lams)
```

3. copying grids for parameters with multiple dimensions. if we specify a 1D np.ndarray for lam, we are implicitly testing the space where all points have the same value

```
>>> gam.gridsearch(lam=np.logspace(-3, 3, 11))
```

is equivalent to:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = np.array([lam] * 4)
>>> gam.gridsearch(X, y, lam=lams)
```

loglikelihood(*X*, *y*, *weights=None*)

compute the log-likelihood of the dataset using the current model

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset
- **y** (*array-like of shape (n,)*) – containing target values
- **weights** (*array-like of shape (n,), optional*) – containing sample weights

Returns **log-likelihood** – containing log-likelihood scores

Return type np.array of shape (n,)

partial_dependence(*term*, *X=None*, *width=None*, *quantiles=None*, *meshgrid=False*)
Computes the term functions for the GAM and possibly their confidence intervals.

if both width=None and quantiles=None, then no confidence intervals are computed

Parameters

- **term** (*int, optional*) – Term for which to compute the partial dependence functions.
- **X** (*array-like with input data, optional*) – if *meshgrid=False*, then *X* should be an array-like of shape (n_samples, m_features).

  if *meshgrid=True*, then *X* should be a tuple containing an array for each feature in the term.

  if None, an equally spaced grid of points is generated.
- **width** (*float on (0, 1), optional*) – Width of the confidence interval.
- **quantiles** (*array-like of floats on (0, 1), optional*) – instead of specifying the prediciton width, one can specify the quantiles. so width=.95 is equivalent to quantiles=[.025, .975]. if None, defaults to width.
- **meshgrid** (*bool, whether to return and accept meshgrids.*) – Useful for creating outputs that are suitable for 3D plotting.

  Note, for simple terms with no interactions, the output of this function will be the same for `meshgrid=True` and `meshgrid=False`, but the inputs will need to be different.

Returns

- **pdeps** (*np.array of shape (n_samples,)*)
- **conf_intervals** (*list of length len(term)*) – containing np.arrays of shape (n_samples, 2 or len(quantiles))

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

See also:

generate_X_grid() for help creating meshgrids.

predict(*X*)
preduct expected value of target given model and input X often this is done via expected value of GAM given input X

Parameters **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset

Returns **y** – containing predicted values under the model

Return type np.array of shape (n_samples,)

predict_mu(*X*)
preduct expected value of target given model and input X

Parameters **X** (*array-like of shape (n_samples, m_features),*) – containing the input dataset

Returns **y** – containing expected values under the model

Return type np.array of shape (n_samples,)

sample(*X*, *y*, *quantity='y'*, *sample_at_X=None*, *weights=None*, *n_draws=100*, *n_bootstraps=5*, *objective='auto'*)

Simulate from the posterior of the coefficients and smoothing params.

Samples are drawn from the posterior of the coefficients and smoothing parameters given the response in an approximate way. The GAM must already be fitted before calling this method; if the model has not been fitted, then an exception is raised. Moreover, it is recommended that the model and its hyperparameters be chosen with *gridsearch* (with the parameter *keep_best=True*) before calling *sample*, so that the result of that gridsearch can be used to generate useful response data and so that the model's coefficients (and their covariance matrix) can be used as the first bootstrap sample.

These samples are drawn as follows. Details are in the reference below.

1. `n_bootstraps` many "bootstrap samples" of the response (`y`) are simulated by drawing random samples from the model's distribution evaluated at the expected values (`mu`) for each sample in `X`.

2. A copy of the model is fitted to each of those bootstrap samples of the response. The result is an approximation of the distribution over the smoothing parameter `lam` given the response data `y`.

3. Samples of the coefficients are simulated from a multivariate normal using the bootstrap samples of the coefficients and their covariance matrices.

Notes

A `gridsearch` is done `n_bootstraps` many times, so keep `n_bootstraps` small. Make `n_bootstraps < n_draws` to take advantage of the expensive bootstrap samples of the smoothing parameters.

Parameters

- **X** (*array of shape (n_samples, m_features)*) – empirical input data
- **y** (*array of shape (n_samples,)*) – empirical response vector
- **quantity** (*{'y', 'coef', 'mu'}, default: 'y'*) – What quantity to return pseudorandom samples of. If *sample_at_X* is not None and *quantity* is either *'y'* or *'mu'*, then samples are drawn at the values of *X* specified in *sample_at_X*.
- **sample_at_X** (*array of shape (n_samples_to_simulate, m_features) or*) –
- **optional** (*None,*) – Input data at which to draw new samples.

  Only applies for *quantity* equal to *'y'* or to *'mu*'. If *None*, then *sample_at_X* is replaced by *X*.
- **weights** (*np.array of shape (n_samples,)*) – sample weights
- **n_draws** (*positive int, optional (default=100)*) – The number of samples to draw from the posterior distribution of the coefficients and smoothing parameters
- **n_bootstraps** (*positive int, optional (default=5)*) – The number of bootstrap samples to draw from simulations of the response (from the already fitted model) to estimate the distribution of the smoothing parameters given the response data. If *n_bootstraps* is 1, then only the already fitted model's smoothing parameter is used, and the distribution over the smoothing parameters is not estimated using bootstrap sampling.

- **objective** (*string, optional (default='auto')*) – metric to optimize in grid search. must be in ['AIC', 'AICc', 'GCV', 'UBRE', 'auto'] if 'auto', then grid search will optimize GCV for models with unknown scale and UBRE for models with known scale.

Returns

draws – Simulations of the given *quantity* using samples from the posterior distribution of the coefficients and smoothing parameter given the response data. Each row is a pseudorandom sample.

If *quantity == 'coef'*, then the number of columns of *draws* is the number of coefficients (*len(self.coef_)*).

Otherwise, the number of columns of *draws* is the number of rows of *sample_at_X* if *sample_at_X* is not *None* or else the number of rows of *X*.

Return type 2D array of length n_draws

References

Simon N. Wood, 2006. Generalized Additive Models: an introduction with R. Section 4.9.3 (pages 198–199) and Section 5.4.2 (page 256–257).

score(*X*, *y*, *weights=None*)
method to compute the explained deviance for a trained model for a given X data and y labels

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **explained deviancce score**

Return type np.array() (n_samples,)

set_params(*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters
- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have
- ****parameters** (*paramters to set*) –

Returns

Return type self

summary()
produce a summary of the model statistics

Parameters **None** –

Returns

Return type None

LogisticGAM

class pygam.pygam.**LogisticGAM**(*terms='auto', max_iter=100, tol=0.0001, callbacks=['deviance', 'diffs', 'accuracy'], fit_intercept=True, verbose=False, **kwargs*)

Bases: *pygam.pygam.GAM*

Logistic GAM

This is a GAM with a Binomial error distribution, and a logit link.

Parameters

- **terms** (*expression specifying terms to model, optional.*) – By default a univariate spline term will be allocated for each feature.

  For example:

  ```
  >>> GAM(s(0) + l(1) + f(2) + te(3, 4))
  ```

  will fit a spline term on feature 0, a linear term on feature 1, a factor term on feature 2, and a tensor term on features 3 and 4.

- **callbacks** (*list of str or list of CallBack objects, optional*) – Names of callback objects to call during the optimization loop.
- **fit_intercept** (*bool, optional*) – Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function. Note: the intercept receives no smoothing penalty.
- **max_iter** (*int, optional*) – Maximum number of iterations allowed for the solver to converge.
- **tol** (*float, optional*) – Tolerance for stopping criteria.
- **verbose** (*bool, optional*) – whether to show pyGAM warnings.

coef_
Coefficient of the features in the decision function. If fit_intercept is True, then self.coef_[0] will contain the bias.

Type array, shape (n_classes, m_features)

statistics_
Dictionary containing model statistics like GCV/UBRE scores, AIC/c, parameter covariances, estimated degrees of freedom, etc.

Type dict

logs_
Dictionary containing the outputs of any callbacks at each optimization loop.

The logs are structured as `{callback: [...]}`

Type dict

References

Simon N. Wood, 2006 Generalized Additive Models: an introduction with R

Hastie, Tibshirani, Friedman The Elements of Statistical Learning http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf

Paul Eilers & Brian Marx, 2015 International Biometric Society: A Crash Course on P-splines http://www.ibschannel2015.nl/project/userfiles/Crash_course_handout.pdf

accuracy(*X=None*, *y=None*, *mu=None*)
computes the accuracy of the LogisticGAM

Parameters

- **note** (*X or mu must be defined. defaults to mu*) –
- **X** (*array-like of shape (n_samples, m_features), optional (default=None)*) – containing input data
- **y** (*array-like of shape (n,)*) – containing target data
- **mu** (*array-like of shape (n_samples,), optional (default=None*) – expected value of the targets given the model and inputs

Returns

Return type float in [0, 1]

confidence_intervals(*X*, *width=0.95*, *quantiles=None*)
estimate confidence intervals for the model.

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – Input data matrix
- **width** (*float on [0,1], optional*) –
- **quantiles** (*array-like of floats in (0, 1), optional*) – Instead of specifying the prediciton width, one can specify the quantiles. So `width=.95` is equivalent to `quantiles=[.025, .975]`

Returns **intervals**

Return type np.array of shape (n_samples, 2 or len(quantiles))

Notes

Wood 2006, section 4.9 Confidence intervals based on section 4.8 rely on large sample results to deal with non-Gaussian distributions, and treat the smoothing parameters as fixed, when in reality they are estimated from the data.

deviance_residuals(*X*, *y*, *weights=None*, *scaled=False*)
method to compute the deviance residuals of the model

these are analogous to the residuals of an OLS.

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones
- **scaled** (*bool, optional*) – whether to scale the deviance by the (estimated) distribution scale

Returns **deviance_residuals** – with shape (n_samples,)

Return type np.array

fit (*X*, *y*, *weights=None*)
Fit the generalized additive model.

Parameters

- **X** (*array-like, shape (n_samples, m_features)*) – Training vectors.
- **y** (*array-like, shape (n_samples,)*) – Target values, ie integers in classification, real numbers in regression)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **self** – Returns fitted GAM object

Return type object

generate_X_grid (*term*, *n=100*, *meshgrid=False*)
create a nice grid of X data

array is sorted by feature and uniformly spaced, so the marginal and joint distributions are likely wrong

if term is >= 0, we generate n samples per feature, which results in n^deg samples, where deg is the degree of the interaction of the term

Parameters

- **term** (*int,*) – Which term to process.
- **n** (*int, optional*) – number of data points to create
- **meshgrid** (*bool, optional*) – Whether to return a meshgrid (useful for 3d plotting) or a feature matrix (useful for inference like partial predictions)

Returns

- *if meshgrid is False* – np.array of shape (n, n_features) where m is the number of (sub)terms in the requested (tensor)term.
- *else* – tuple of len m, where m is the number of (sub)terms in the requested (tensor)term.

  each element in the tuple contains a np.ndarray of size (n)^m

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

get_params (*deep=False*)
returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

gridsearch (*X*, *y*, *weights=None*, *return_scores=False*, *keep_best=True*, *objective='auto'*, *progress=True*, ***param_grids*)
Performs a grid search over a space of parameters for a given objective

Warning: `gridsearch` is lazy and will not remove useless combinations from the search space, eg.

```
>>> n_splines=np.arange(5,10), fit_splines=[True, False]
```

will result in 10 loops, of which 5 are equivalent because `fit_splines = False`

Also, it is not recommended to search over a grid that alternates between known scales and unknown scales, as the scores of the candidate models will not be comparable.

Parameters

- **X** (*array-like*) – input data of shape (n_samples, m_features)
- **y** (*array-like*) – label data of shape (n_samples,)
- **weights** (*array-like shape (n_samples,), optional*) – sample weights
- **return_scores** (*boolean, optional*) – whether to return the hyperpamaters and score for each element in the grid
- **keep_best** (*boolean, optional*) – whether to keep the best GAM as self.
- **objective** (*{'auto', 'AIC', 'AICc', 'GCV', 'UBRE'}, optional*) – Metric to optimize. If *auto*, then grid search will optimize *GCV* for models with unknown scale and *UBRE* for models with known scale.
- **progress** (*bool, optional*) – whether to display a progress bar
- ****kwargs** – pairs of parameters and iterables of floats, or parameters and iterables of iterables of floats.

  If no parameter are specified, `lam=np.logspace(-3, 3, 11)` is used. This results in a 11 points, placed diagonally across lam space.

  If grid is iterable of iterables of floats, the outer iterable must have length `m_features`. the cartesian product of the subgrids in the grid will be tested.

  If grid is a 2d numpy array, each row of the array will be tested.

  The method will make a grid of all the combinations of the parameters and fit a GAM to each combination.

Returns

- if `return_scores=True` – model_scores: dict containing each fitted model as keys and corresponding objective scores as values
- *else* – self: ie possibly the newly fitted model

Examples

For a model with 4 terms, and where we expect 4 lam values, our search space for lam must have 4 dimensions.

We can search the space in 3 ways:

1. via cartesian product by specifying the grid as a list. our grid search will consider `11 ** 4` points:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = [lam] * 4
>>> gam.gridsearch(X, y, lam=lams)
```

2. directly by specifying the grid as a np.ndarray. This is useful for when the dimensionality of the search space is very large, and we would prefer to execute a randomized search:

```
>>> lams = np.exp(np.random.random(50, 4) * 6 - 3)
>>> gam.gridsearch(X, y, lam=lams)
```

3. copying grids for parameters with multiple dimensions. if we specify a 1D np.ndarray for lam, we are implicitly testing the space where all points have the same value

```
>>> gam.gridsearch(lam=np.logspace(-3, 3, 11))
```

is equivalent to:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = np.array([lam] * 4)
>>> gam.gridsearch(X, y, lam=lams)
```

loglikelihood(*X*, *y*, *weights=None*)

compute the log-likelihood of the dataset using the current model

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset
- **y** (*array-like of shape (n,)*) – containing target values
- **weights** (*array-like of shape (n,), optional*) – containing sample weights

Returns **log-likelihood** – containing log-likelihood scores

Return type np.array of shape (n,)

partial_dependence(*term*, *X=None*, *width=None*, *quantiles=None*, *meshgrid=False*)

Computes the term functions for the GAM and possibly their confidence intervals.

if both width=None and quantiles=None, then no confidence intervals are computed

Parameters

- **term** (*int, optional*) – Term for which to compute the partial dependence functions.
- **X** (*array-like with input data, optional*) – if *meshgrid=False*, then *X* should be an array-like of shape (n_samples, m_features).

  if *meshgrid=True*, then *X* should be a tuple containing an array for each feature in the term.

  if None, an equally spaced grid of points is generated.
- **width** (*float on (0, 1), optional*) – Width of the confidence interval.
- **quantiles** (*array-like of floats on (0, 1), optional*) – instead of specifying the prediciton width, one can specify the quantiles. so width=.95 is equivalent to quantiles=[.025, .975]. if None, defaults to width.
- **meshgrid** (*bool, whether to return and accept meshgrids.*) – Useful for creating outputs that are suitable for 3D plotting.

  Note, for simple terms with no interactions, the output of this function will be the same for `meshgrid=True` and `meshgrid=False`, but the inputs will need to be different.

Returns

- **pdeps** (*np.array of shape (n_samples,)*)

- **conf_intervals** (*list of length len(term)*) – containing np.arrays of shape (n_samples, 2 or len(quantiles))

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

See also:

generate_X_grid() for help creating meshgrids.

predict(*X*)

preduct binary targets given model and input X

Parameters **X** (*array-like of shape (n_samples, m_features), optional (default=None)*) – containing the input dataset

Returns **y** – containing binary targets under the model

Return type np.array of shape (n_samples,)

predict_mu(*X*)

preduct expected value of target given model and input X

Parameters **X** (*array-like of shape (n_samples, m_features),*) – containing the input dataset

Returns **y** – containing expected values under the model

Return type np.array of shape (n_samples,)

predict_proba(*X*)

preduct targets given model and input X

Parameters **X** (*array-like of shape (n_samples, m_features), optional (default=None*) – containing the input dataset

Returns **y** – containing expected values under the model

Return type np.array of shape (n_samples,)

sample(*X*, *y*, *quantity='y'*, *sample_at_X=None*, *weights=None*, *n_draws=100*, *n_bootstraps=5*, *objective='auto'*)

Simulate from the posterior of the coefficients and smoothing params.

Samples are drawn from the posterior of the coefficients and smoothing parameters given the response in an approximate way. The GAM must already be fitted before calling this method; if the model has not been fitted, then an exception is raised. Moreover, it is recommended that the model and its hyperparameters be chosen with *gridsearch* (with the parameter *keep_best=True*) before calling *sample*, so that the result of that gridsearch can be used to generate useful response data and so that the model's coefficients (and their covariance matrix) can be used as the first bootstrap sample.

These samples are drawn as follows. Details are in the reference below.

1. `n_bootstraps` many "bootstrap samples" of the response (`y`) are simulated by drawing random samples from the model's distribution evaluated at the expected values (`mu`) for each sample in `X`.

2. A copy of the model is fitted to each of those bootstrap samples of the response. The result is an approximation of the distribution over the smoothing parameter `lam` given the response data `y`.

3. Samples of the coefficients are simulated from a multivariate normal using the bootstrap samples of the coefficients and their covariance matrices.

Notes

A `gridsearch` is done `n_bootstraps` many times, so keep `n_bootstraps` small. Make `n_bootstraps < n_draws` to take advantage of the expensive bootstrap samples of the smoothing parameters.

Parameters

- **X** (*array of shape (n_samples, m_features)*) – empirical input data
- **y** (*array of shape (n_samples,)*) – empirical response vector
- **quantity** (*{'y', 'coef', 'mu'}, default: 'y'*) – What quantity to return pseudorandom samples of. If *sample_at_X* is not None and *quantity* is either *'y'* or *'mu'*, then samples are drawn at the values of *X* specified in *sample_at_X*.
- **sample_at_X** (*array of shape (n_samples_to_simulate, m_features) or*) –
- **optional** (*None,*) – Input data at which to draw new samples.

  Only applies for *quantity* equal to *'y'* or to *'mu*'. If *None*, then *sample_at_X* is replaced by *X*.
- **weights** (*np.array of shape (n_samples,)*) – sample weights
- **n_draws** (*positive int, optional (default=100)*) – The number of samples to draw from the posterior distribution of the coefficients and smoothing parameters
- **n_bootstraps** (*positive int, optional (default=5)*) – The number of bootstrap samples to draw from simulations of the response (from the already fitted model) to estimate the distribution of the smoothing parameters given the response data. If *n_bootstraps* is 1, then only the already fitted model's smoothing parameter is used, and the distribution over the smoothing parameters is not estimated using bootstrap sampling.
- **objective** (*string, optional (default='auto'*) – metric to optimize in grid search. must be in ['AIC', 'AICc', 'GCV', 'UBRE', 'auto'] if 'auto', then grid search will optimize GCV for models with unknown scale and UBRE for models with known scale.

Returns

draws – Simulations of the given *quantity* using samples from the posterior distribution of the coefficients and smoothing parameter given the response data. Each row is a pseudorandom sample.

If *quantity == 'coef'*, then the number of columns of *draws* is the number of coefficients (*len(self.coef_)*).

Otherwise, the number of columns of *draws* is the number of rows of *sample_at_X* if *sample_at_X* is not *None* or else the number of rows of *X*.

Return type 2D array of length n_draws

References

Simon N. Wood, 2006. Generalized Additive Models: an introduction with R. Section 4.9.3 (pages 198–199) and Section 5.4.2 (page 256–257).

score (*X*, *y*)

method to compute the accuracy for a trained model for a given X data and y labels

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)

Returns accuracy score

Return type np.array() (n_samples,)

set_params(*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters
- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have
- ****parameters** (*paramters to set*) –

Returns

Return type self

summary()
produce a summary of the model statistics

Parameters **None** –

Returns

Return type None

PoissonGAM

class pygam.pygam.**PoissonGAM**(*terms='auto', max_iter=100, tol=0.0001, callbacks=['deviance', 'diffs'], fit_intercept=True, verbose=False, **kwargs*)

Bases: *pygam.pygam.GAM*

Poisson GAM

This is a GAM with a Poisson error distribution, and a log link.

Parameters

- **terms** (*expression specifying terms to model, optional.*) – By default a univariate spline term will be allocated for each feature.

  For example:

  ```
  >>> GAM(s(0) + l(1) + f(2) + te(3, 4))
  ```

  will fit a spline term on feature 0, a linear term on feature 1, a factor term on feature 2, and a tensor term on features 3 and 4.
- **callbacks** (*list of str or list of CallBack objects, optional*) – Names of callback objects to call during the optimization loop.
- **fit_intercept** (*bool, optional*) – Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function. Note: the intercept receives no smoothing penalty.

- **max_iter** (*int, optional*) – Maximum number of iterations allowed for the solver to converge.
- **tol** (*float, optional*) – Tolerance for stopping criteria.
- **verbose** (*bool, optional*) – whether to show pyGAM warnings.

coef_
Coefficient of the features in the decision function. If fit_intercept is True, then self.coef_[0] will contain the bias.

Type array, shape (n_classes, m_features)

statistics_
Dictionary containing model statistics like GCV/UBRE scores, AIC/c, parameter covariances, estimated degrees of freedom, etc.

Type dict

logs_
Dictionary containing the outputs of any callbacks at each optimization loop.

The logs are structured as `{callback: [...]}`

Type dict

References

Simon N. Wood, 2006 Generalized Additive Models: an introduction with R

Hastie, Tibshirani, Friedman The Elements of Statistical Learning http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf

Paul Eilers & Brian Marx, 2015 International Biometric Society: A Crash Course on P-splines http://www.ibschannel2015.nl/project/userfiles/Crash_course_handout.pdf

confidence_intervals(*X*, *width=0.95*, *quantiles=None*)
estimate confidence intervals for the model.

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – Input data matrix
- **width** (*float on [0,1], optional*) –
- **quantiles** (*array-like of floats in (0, 1), optional*) – Instead of specifying the prediciton width, one can specify the quantiles. So `width=.95` is equivalent to `quantiles=[.025, .975]`

Returns **intervals**

Return type np.array of shape (n_samples, 2 or len(quantiles))

Notes

Wood 2006, section 4.9 Confidence intervals based on section 4.8 rely on large sample results to deal with non-Gaussian distributions, and treat the smoothing parameters as fixed, when in reality they are estimated from the data.

deviance_residuals(*X*, *y*, *weights=None*, *scaled=False*)
method to compute the deviance residuals of the model

these are analogous to the residuals of an OLS.

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones
- **scaled** (*bool, optional*) – whether to scale the deviance by the (estimated) distribution scale

Returns **deviance_residuals** – with shape (n_samples,)

Return type np.array

fit(*X*, *y*, *exposure=None*, *weights=None*)
Fit the generalized additive model.

Parameters

- **X** (*array-like, shape (n_samples, m_features)*) – Training vectors, where n_samples is the number of samples and m_features is the number of features.
- **y** (*array-like, shape (n_samples,)*) – Target values (integers in classification, real numbers in regression) For classification, labels must correspond to classes.
- **exposure** (*array-like shape (n_samples,) or None, default: None*) – containing exposures if None, defaults to array of ones
- **weights** (*array-like shape (n_samples,) or None, default: None*) – containing sample weights if None, defaults to array of ones

Returns **self** – Returns fitted GAM object

Return type object

generate_X_grid(*term*, *n=100*, *meshgrid=False*)
create a nice grid of X data

array is sorted by feature and uniformly spaced, so the marginal and joint distributions are likely wrong

if term is >= 0, we generate n samples per feature, which results in n^deg samples, where deg is the degree of the interaction of the term

Parameters

- **term** (*int,*) – Which term to process.
- **n** (*int, optional*) – number of data points to create
- **meshgrid** (*bool, optional*) – Whether to return a meshgrid (useful for 3d plotting) or a feature matrix (useful for inference like partial predictions)

Returns

- *if meshgrid is False* – np.array of shape (n, n_features) where m is the number of (sub)terms in the requested (tensor)term.
- *else* – tuple of len m, where m is the number of (sub)terms in the requested (tensor)term.

  each element in the tuple contains a np.ndarray of size (n)^m

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

get_params(*deep=False*)
returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

gridsearch(*X*, *y*, *exposure=None*, *weights=None*, *return_scores=False*, *keep_best=True*, *objective='auto'*, ***param_grids*)
performs a grid search over a space of parameters for a given objective

NOTE: gridsearch method is lazy and will not remove useless combinations from the search space, eg.

```
>>> n_splines=np.arange(5,10), fit_splines=[True, False]
```

will result in 10 loops, of which 5 are equivalent because even though fit_splines==False

it is not recommended to search over a grid that alternates between known scales and unknown scales, as the scores of the candidate models will not be comparable.

Parameters

- **X** (*array*) – input data of shape (n_samples, m_features)
- **y** (*array*) – label data of shape (n_samples,)
- **exposure** (*array-like shape (n_samples,) or None, default: None*) – containing exposures if None, defaults to array of ones
- **weights** (*array-like shape (n_samples,) or None, default: None*) – containing sample weights if None, defaults to array of ones
- **return_scores** (*boolean, default False*) – whether to return the hyperpamaters and score for each element in the grid
- **keep_best** (*boolean*) – whether to keep the best GAM as self. default: True
- **objective** (*string, default: 'auto'*) – metric to optimize. must be in ['AIC', 'AICc', 'GCV', 'UBRE', 'auto'] if 'auto', then grid search will optimize GCV for models with unknown scale and UBRE for models with known scale.
- ****kwargs** (*dict, default {'lam': np.logspace(-3, 3, 11)}*) – pairs of parameters and iterables of floats, or parameters and iterables of iterables of floats.

  if iterable of iterables of floats, the outer iterable must have length m_features.

  the method will make a grid of all the combinations of the parameters and fit a GAM to each combination.

Returns

- *if return_values == True* –

  model_scores [dict] Contains each fitted model as keys and corresponding objective scores as values

- *else* – self, ie possibly the newly fitted model

loglikelihood(*X*, *y*, *exposure=None*, *weights=None*)
compute the log-likelihood of the dataset using the current model

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset
- **y** (*array-like of shape (n,)*) – containing target values
- **exposure** (*array-like shape (n_samples,) or None, default: None*) – containing exposures if None, defaults to array of ones
- **weights** (*array-like of shape (n,)*) – containing sample weights

Returns **log-likelihood** – containing log-likelihood scores

Return type np.array of shape (n,)

partial_dependence(*term*, *X=None*, *width=None*, *quantiles=None*, *meshgrid=False*)

Computes the term functions for the GAM and possibly their confidence intervals.

if both width=None and quantiles=None, then no confidence intervals are computed

Parameters

- **term** (*int, optional*) – Term for which to compute the partial dependence functions.
- **X** (*array-like with input data, optional*) – if *meshgrid=False*, then *X* should be an array-like of shape (n_samples, m_features).

  if *meshgrid=True*, then *X* should be a tuple containing an array for each feature in the term.

  if None, an equally spaced grid of points is generated.
- **width** (*float on (0, 1), optional*) – Width of the confidence interval.
- **quantiles** (*array-like of floats on (0, 1), optional*) – instead of specifying the prediciton width, one can specify the quantiles. so width=.95 is equivalent to quantiles=[.025, .975]. if None, defaults to width.
- **meshgrid** (*bool, whether to return and accept meshgrids.*) – Useful for creating outputs that are suitable for 3D plotting.

  Note, for simple terms with no interactions, the output of this function will be the same for `meshgrid=True` and `meshgrid=False`, but the inputs will need to be different.

Returns

- **pdeps** (*np.array of shape (n_samples,)*)
- **conf_intervals** (*list of length len(term)*) – containing np.arrays of shape (n_samples, 2 or len(quantiles))

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

See also:

generate_X_grid() for help creating meshgrids.

predict(*X*, *exposure=None*)

preduct expected value of target given model and input X often this is done via expected value of GAM given input X

Parameters

- **X** (*array-like of shape (n_samples, m_features), default: None*) – containing the input dataset
- **exposure** (*array-like shape (n_samples,) or None, default: None*) – containing exposures if None, defaults to array of ones

Returns **y** – containing predicted values under the model

Return type np.array of shape (n_samples,)

predict_mu(*X*)
preduct expected value of target given model and input X

Parameters **X** (*array-like of shape (n_samples, m_features),*) – containing the input dataset

Returns **y** – containing expected values under the model

Return type np.array of shape (n_samples,)

sample(*X*, *y*, *quantity='y'*, *sample_at_X=None*, *weights=None*, *n_draws=100*, *n_bootstraps=5*, *objective='auto'*)
Simulate from the posterior of the coefficients and smoothing params.

Samples are drawn from the posterior of the coefficients and smoothing parameters given the response in an approximate way. The GAM must already be fitted before calling this method; if the model has not been fitted, then an exception is raised. Moreover, it is recommended that the model and its hyperparameters be chosen with *gridsearch* (with the parameter *keep_best=True*) before calling *sample*, so that the result of that gridsearch can be used to generate useful response data and so that the model's coefficients (and their covariance matrix) can be used as the first bootstrap sample.

These samples are drawn as follows. Details are in the reference below.

1. `n_bootstraps` many "bootstrap samples" of the response (`y`) are simulated by drawing random samples from the model's distribution evaluated at the expected values (`mu`) for each sample in `X`.

2. A copy of the model is fitted to each of those bootstrap samples of the response. The result is an approximation of the distribution over the smoothing parameter `lam` given the response data `y`.

3. Samples of the coefficients are simulated from a multivariate normal using the bootstrap samples of the coefficients and their covariance matrices.

Notes

A `gridsearch` is done `n_bootstraps` many times, so keep `n_bootstraps` small. Make `n_bootstraps < n_draws` to take advantage of the expensive bootstrap samples of the smoothing parameters.

Parameters

- **X** (*array of shape (n_samples, m_features)*) – empirical input data
- **y** (*array of shape (n_samples,)*) – empirical response vector
- **quantity** (*{'y', 'coef', 'mu'}, default: 'y'*) – What quantity to return pseudorandom samples of. If *sample_at_X* is not None and *quantity* is either *'y'* or *'mu'*, then samples are drawn at the values of *X* specified in *sample_at_X*.
- **sample_at_X** (*array of shape (n_samples_to_simulate, m_features) or*) –

- **optional** (*None,*) – Input data at which to draw new samples.

  Only applies for *quantity* equal to *'y'* or to *'mu'*. If *None*, then *sample_at_X* is replaced by *X*.

- **weights** (*np.array of shape (n_samples,)*) – sample weights
- **n_draws** (*positive int, optional (default=100)*) – The number of samples to draw from the posterior distribution of the coefficients and smoothing parameters
- **n_bootstraps** (*positive int, optional (default=5)*) – The number of bootstrap samples to draw from simulations of the response (from the already fitted model) to estimate the distribution of the smoothing parameters given the response data. If *n_bootstraps* is 1, then only the already fitted model's smoothing parameter is used, and the distribution over the smoothing parameters is not estimated using bootstrap sampling.
- **objective** (*string, optional (default='auto')* – metric to optimize in grid search. must be in ['AIC', 'AICc', 'GCV', 'UBRE', 'auto'] if 'auto', then grid search will optimize GCV for models with unknown scale and UBRE for models with known scale.

Returns

draws – Simulations of the given *quantity* using samples from the posterior distribution of the coefficients and smoothing parameter given the response data. Each row is a pseudorandom sample.

If *quantity == 'coef'*, then the number of columns of *draws* is the number of coefficients (*len(self.coef_)*).

Otherwise, the number of columns of *draws* is the number of rows of *sample_at_X* if *sample_at_X* is not *None* or else the number of rows of *X*.

Return type 2D array of length n_draws

References

Simon N. Wood, 2006. Generalized Additive Models: an introduction with R. Section 4.9.3 (pages 198–199) and Section 5.4.2 (page 256–257).

score (*X*, *y*, *weights=None*)
method to compute the explained deviance for a trained model for a given X data and y labels

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns explained deviancce score

Return type np.array() (n_samples,)

set_params (*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters

- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have
- ****parameters** (*paramters to set*) –

Returns

Return type self

summary()

produce a summary of the model statistics

Parameters **None** –

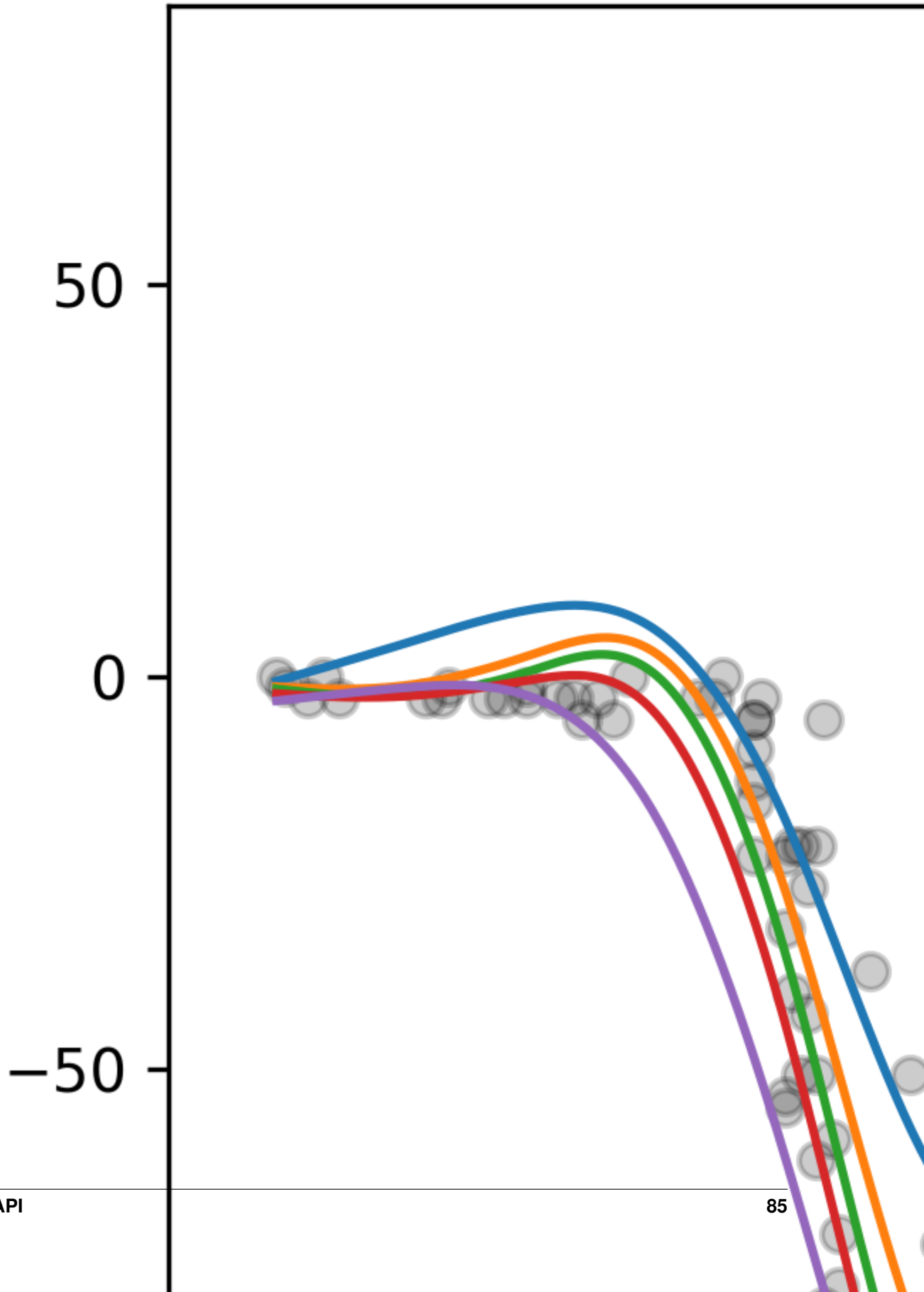
Returns

Return type None

ExpectileGAM

```
from pygam import ExpectileGAM
from pygam.datasets import mcycle

X, y = mcycle(return_X_y=True)

# lets fit the mean model first by CV
gam50 = ExpectileGAM(expectile=0.5).gridsearch(X, y)

# and copy the smoothing to the other models
lam = gam50.lam

# now fit a few more models
gam95 = ExpectileGAM(expectile=0.95, lam=lam).fit(X, y)
gam75 = ExpectileGAM(expectile=0.75, lam=lam).fit(X, y)
gam25 = ExpectileGAM(expectile=0.25, lam=lam).fit(X, y)
gam05 = ExpectileGAM(expectile=0.05, lam=lam).fit(X, y)
```

```
from matplotlib import pyplot as plt

XX = gam50.generate_X_grid(term=0, n=500)

plt.scatter(X, y, c='k', alpha=0.2)
plt.plot(XX, gam95.predict(XX), label='0.95')
plt.plot(XX, gam75.predict(XX), label='0.75')
plt.plot(XX, gam50.predict(XX), label='0.50')
plt.plot(XX, gam25.predict(XX), label='0.25')
plt.plot(XX, gam05.predict(XX), label='0.05')
plt.legend()
```


50
0
−50

class pygam.pygam.**ExpectileGAM**(*terms='auto', max_iter=100, tol=0.0001, scale=None, callbacks=['deviance', 'diffs'], fit_intercept=True, expectile=0.5, verbose=False, **kwargs*)

Bases: *pygam.pygam.GAM*

Expectile GAM

This is a GAM with a Normal distribution and an Identity Link, but minimizing the Least Asymmetrically Weighted Squares

Parameters

- **terms** (*expression specifying terms to model, optional.*) – By default a univariate spline term will be allocated for each feature.

  For example:

  ```
  >>> GAM(s(0) + l(1) + f(2) + te(3, 4))
  ```

  will fit a spline term on feature 0, a linear term on feature 1, a factor term on feature 2, and a tensor term on features 3 and 4.

- **callbacks** (*list of str or list of CallBack objects, optional*) – Names of callback objects to call during the optimization loop.
- **fit_intercept** (*bool, optional*) – Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function. Note: the intercept receives no smoothing penalty.
- **max_iter** (*int, optional*) – Maximum number of iterations allowed for the solver to converge.
- **tol** (*float, optional*) – Tolerance for stopping criteria.
- **verbose** (*bool, optional*) – whether to show pyGAM warnings.

coef_
Coefficient of the features in the decision function. If fit_intercept is True, then self.coef_[0] will contain the bias.

Type array, shape (n_classes, m_features)

statistics_
Dictionary containing model statistics like GCV/UBRE scores, AIC/c, parameter covariances, estimated degrees of freedom, etc.

Type dict

logs_
Dictionary containing the outputs of any callbacks at each optimization loop.

The logs are structured as `{callback: [...]}`

Type dict

References

Simon N. Wood, 2006 Generalized Additive Models: an introduction with R

Hastie, Tibshirani, Friedman The Elements of Statistical Learning http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf

Paul Eilers & Brian Marx, 2015 International Biometric Society: A Crash Course on P-splines http://www.ibschannel2015.nl/project/userfiles/Crash_course_handout.pdf

confidence_intervals(*X*, *width=0.95*, *quantiles=None*)
estimate confidence intervals for the model.

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – Input data matrix
- **width** (*float on [0,1], optional*) –
- **quantiles** (*array-like of floats in (0, 1), optional*) – Instead of specifying the prediciton width, one can specify the quantiles. So `width=.95` is equivalent to `quantiles=[.025, .975]`

Returns **intervals**

Return type np.array of shape (n_samples, 2 or len(quantiles))

Notes

Wood 2006, section 4.9 Confidence intervals based on section 4.8 rely on large sample results to deal with non-Gaussian distributions, and treat the smoothing parameters as fixed, when in reality they are estimated from the data.

deviance_residuals(*X*, *y*, *weights=None*, *scaled=False*)
method to compute the deviance residuals of the model

these are analogous to the residuals of an OLS.

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones
- **scaled** (*bool, optional*) – whether to scale the deviance by the (estimated) distribution scale

Returns **deviance_residuals** – with shape (n_samples,)

Return type np.array

fit(*X*, *y*, *weights=None*)
Fit the generalized additive model.

Parameters

- **X** (*array-like, shape (n_samples, m_features)*) – Training vectors.
- **y** (*array-like, shape (n_samples,)*) – Target values, ie integers in classification, real numbers in regression)
- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **self** – Returns fitted GAM object

Return type object

fit_quantile(*X*, *y*, *quantile*, *max_iter=20*, *tol=0.01*, *weights=None*)
fit ExpectileGAM to a desired quantile via binary search

Parameters

- **X** (*array-like, shape (n_samples, m_features)*) – Training vectors, where n_samples is the number of samples and m_features is the number of features.
- **y** (*array-like, shape (n_samples,)*) – Target values (integers in classification, real numbers in regression) For classification, labels must correspond to classes.
- **quantile** (*float on (0, 1)*) – desired quantile to fit.
- **max_iter** (*int, default: 20*) – maximum number of binary search iterations to perform
- **tol** (*float > 0, default: 0.01*) – maximum distance between desired quantile and fitted quantile
- **weights** (*array-like shape (n_samples,) or None, default: None*) – containing sample weights if None, defaults to array of ones

Returns self

Return type fitted GAM object

generate_X_grid(*term*, *n=100*, *meshgrid=False*)

create a nice grid of X data

array is sorted by feature and uniformly spaced, so the marginal and joint distributions are likely wrong

if term is >= 0, we generate n samples per feature, which results in n^deg samples, where deg is the degree of the interaction of the term

Parameters

- **term** (*int,*) – Which term to process.
- **n** (*int, optional*) – number of data points to create
- **meshgrid** (*bool, optional*) – Whether to return a meshgrid (useful for 3d plotting) or a feature matrix (useful for inference like partial predictions)

Returns

- *if meshgrid is False* – np.array of shape (n, n_features) where m is the number of (sub)terms in the requested (tensor)term.
- *else* – tuple of len m, where m is the number of (sub)terms in the requested (tensor)term.

  each element in the tuple contains a np.ndarray of size (n)^m

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

get_params(*deep=False*)

returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

gridsearch(*X*, *y*, *weights=None*, *return_scores=False*, *keep_best=True*, *objective='auto'*, *progress=True*, ***param_grids*)

Performs a grid search over a space of parameters for a given objective

Warning: `gridsearch` is lazy and will not remove useless combinations from the search space, eg.

```
>>> n_splines=np.arange(5,10), fit_splines=[True, False]
```

will result in 10 loops, of which 5 are equivalent because `fit_splines = False`

Also, it is not recommended to search over a grid that alternates between known scales and unknown scales, as the scores of the candidate models will not be comparable.

Parameters

- **X** (*array-like*) – input data of shape (n_samples, m_features)
- **y** (*array-like*) – label data of shape (n_samples,)
- **weights** (*array-like shape (n_samples,), optional*) – sample weights
- **return_scores** (*boolean, optional*) – whether to return the hyperpamaters and score for each element in the grid
- **keep_best** (*boolean, optional*) – whether to keep the best GAM as self.
- **objective** (*{'auto', 'AIC', 'AICc', 'GCV', 'UBRE'}, optional*) – Metric to optimize. If *auto*, then grid search will optimize *GCV* for models with unknown scale and *UBRE* for models with known scale.
- **progress** (*bool, optional*) – whether to display a progress bar
- ****kwargs** – pairs of parameters and iterables of floats, or parameters and iterables of iterables of floats.

  If no parameter are specified, `lam=np.logspace(-3, 3, 11)` is used. This results in a 11 points, placed diagonally across lam space.

  If grid is iterable of iterables of floats, the outer iterable must have length `m_features`. the cartesian product of the subgrids in the grid will be tested.

  If grid is a 2d numpy array, each row of the array will be tested.

  The method will make a grid of all the combinations of the parameters and fit a GAM to each combination.

Returns

- if `return_scores=True` – model_scores: dict containing each fitted model as keys and corresponding objective scores as values
- *else* – self: ie possibly the newly fitted model

Examples

For a model with 4 terms, and where we expect 4 lam values, our search space for lam must have 4 dimensions.

We can search the space in 3 ways:

1. via cartesian product by specifying the grid as a list. our grid search will consider `11 ** 4` points:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = [lam] * 4
>>> gam.gridsearch(X, y, lam=lams)
```

2. directly by specifying the grid as a np.ndarray. This is useful for when the dimensionality of the search space is very large, and we would prefer to execute a randomized search:

```
>>> lams = np.exp(np.random.random(50, 4) * 6 - 3)
>>> gam.gridsearch(X, y, lam=lams)
```

3. copying grids for parameters with multiple dimensions. if we specify a 1D np.ndarray for lam, we are implicitly testing the space where all points have the same value

```
>>> gam.gridsearch(lam=np.logspace(-3, 3, 11))
```

is equivalent to:

```
>>> lam = np.logspace(-3, 3, 11)
>>> lams = np.array([lam] * 4)
>>> gam.gridsearch(X, y, lam=lams)
```

loglikelihood(*X*, *y*, *weights=None*)
compute the log-likelihood of the dataset using the current model

Parameters

- **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset
- **y** (*array-like of shape (n,)*) – containing target values
- **weights** (*array-like of shape (n,), optional*) – containing sample weights

Returns **log-likelihood** – containing log-likelihood scores

Return type np.array of shape (n,)

partial_dependence(*term*, *X=None*, *width=None*, *quantiles=None*, *meshgrid=False*)
Computes the term functions for the GAM and possibly their confidence intervals.

if both width=None and quantiles=None, then no confidence intervals are computed

Parameters

- **term** (*int, optional*) – Term for which to compute the partial dependence functions.
- **X** (*array-like with input data, optional*) – if *meshgrid=False*, then *X* should be an array-like of shape (n_samples, m_features).

  if *meshgrid=True*, then *X* should be a tuple containing an array for each feature in the term.

  if None, an equally spaced grid of points is generated.
- **width** (*float on (0, 1), optional*) – Width of the confidence interval.
- **quantiles** (*array-like of floats on (0, 1), optional*) – instead of specifying the prediciton width, one can specify the quantiles. so width=.95 is equivalent to quantiles=[.025, .975]. if None, defaults to width.
- **meshgrid** (*bool, whether to return and accept meshgrids.*) – Useful for creating outputs that are suitable for 3D plotting.

  Note, for simple terms with no interactions, the output of this function will be the same for `meshgrid=True` and `meshgrid=False`, but the inputs will need to be different.

Returns

- **pdeps** (*np.array of shape (n_samples,)*)
- **conf_intervals** (*list of length len(term)*) – containing np.arrays of shape (n_samples, 2 or len(quantiles))

Raises *ValueError :* – If the term requested is an intercept since it does not make sense to process the intercept term.

See also:

generate_X_grid() for help creating meshgrids.

predict(*X*)

preduct expected value of target given model and input X often this is done via expected value of GAM given input X

Parameters **X** (*array-like of shape (n_samples, m_features)*) – containing the input dataset

Returns **y** – containing predicted values under the model

Return type np.array of shape (n_samples,)

predict_mu(*X*)

preduct expected value of target given model and input X

Parameters **X** (*array-like of shape (n_samples, m_features),*) – containing the input dataset

Returns **y** – containing expected values under the model

Return type np.array of shape (n_samples,)

sample(*X*, *y*, *quantity='y'*, *sample_at_X=None*, *weights=None*, *n_draws=100*, *n_bootstraps=5*, *objective='auto'*)

Simulate from the posterior of the coefficients and smoothing params.

Samples are drawn from the posterior of the coefficients and smoothing parameters given the response in an approximate way. The GAM must already be fitted before calling this method; if the model has not been fitted, then an exception is raised. Moreover, it is recommended that the model and its hyperparameters be chosen with *gridsearch* (with the parameter *keep_best=True*) before calling *sample*, so that the result of that gridsearch can be used to generate useful response data and so that the model's coefficients (and their covariance matrix) can be used as the first bootstrap sample.

These samples are drawn as follows. Details are in the reference below.

1. `n_bootstraps` many "bootstrap samples" of the response (`y`) are simulated by drawing random samples from the model's distribution evaluated at the expected values (`mu`) for each sample in `X`.

2. A copy of the model is fitted to each of those bootstrap samples of the response. The result is an approximation of the distribution over the smoothing parameter `lam` given the response data `y`.

3. Samples of the coefficients are simulated from a multivariate normal using the bootstrap samples of the coefficients and their covariance matrices.

Notes

A `gridsearch` is done `n_bootstraps` many times, so keep `n_bootstraps` small. Make `n_bootstraps < n_draws` to take advantage of the expensive bootstrap samples of the smoothing parameters.

Parameters

- **X** (*array of shape (n_samples, m_features)*) – empirical input data
- **y** (*array of shape (n_samples,)*) – empirical response vector
- **quantity** (*{'y', 'coef', 'mu'}, default: 'y'*) – What quantity to return pseudorandom samples of. If *sample_at_X* is not None and *quantity* is either *'y'* or *'mu'*, then samples are drawn at the values of *X* specified in *sample_at_X*.
- **sample_at_X** (*array of shape (n_samples_to_simulate, m_features) or*) –
- **optional** (*None,*) – Input data at which to draw new samples.

  Only applies for *quantity* equal to *'y'* or to *'mu*'. If *None*, then *sample_at_X* is replaced by *X*.
- **weights** (*np.array of shape (n_samples,)*) – sample weights
- **n_draws** (*positive int, optional (default=100)*) – The number of samples to draw from the posterior distribution of the coefficients and smoothing parameters
- **n_bootstraps** (*positive int, optional (default=5)*) – The number of bootstrap samples to draw from simulations of the response (from the already fitted model) to estimate the distribution of the smoothing parameters given the response data. If *n_bootstraps* is 1, then only the already fitted model's smoothing parameter is used, and the distribution over the smoothing parameters is not estimated using bootstrap sampling.
- **objective** (*string, optional (default='auto'*) – metric to optimize in grid search. must be in ['AIC', 'AICc', 'GCV', 'UBRE', 'auto'] if 'auto', then grid search will optimize GCV for models with unknown scale and UBRE for models with known scale.

Returns

draws – Simulations of the given *quantity* using samples from the posterior distribution of the coefficients and smoothing parameter given the response data. Each row is a pseudorandom sample.

If *quantity == 'coef'*, then the number of columns of *draws* is the number of coefficients (*len(self.coef_)*).

Otherwise, the number of columns of *draws* is the number of rows of *sample_at_X* if *sample_at_X* is not *None* or else the number of rows of *X*.

Return type 2D array of length n_draws

References

Simon N. Wood, 2006. Generalized Additive Models: an introduction with R. Section 4.9.3 (pages 198–199) and Section 5.4.2 (page 256–257).

score (*X*, *y*, *weights=None*)
method to compute the explained deviance for a trained model for a given X data and y labels

Parameters

- **X** (*array-like*) – Input data array of shape (n_samples, m_features)
- **y** (*array-like*) – Output data vector of shape (n_samples,)

- **weights** (*array-like shape (n_samples,) or None, optional*) – Sample weights. if None, defaults to array of ones

Returns **explained deviancce score**

Return type np.array() (n_samples,)

set_params(*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters
- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have
- ****parameters** (*paramters to set*) –

Returns

Return type self

summary()
produce a summary of the model statistics

Parameters **None** –

Returns

Return type None

6.3.2 Terms

Linear Term

pygam.terms.**l**(*feature*, *lam=0.6*, *penalties='auto'*, *verbose=False*)
creates an instance of a LinearTerm

feature [int] Index of the feature to use for the feature function.

lam [float or iterable of floats] Strength of smoothing penalty. Must be a positive float. Larger values enforce stronger smoothing.

If single value is passed, it will be repeated for every penalty.

If iterable is passed, the length of *lam* must be equal to the length of *penalties*

penalties [{'auto', 'derivative', 'l2', None} or callable or iterable] Type of smoothing penalty to apply to the term.

If an iterable is used, multiple penalties are applied to the term. The length of the iterable must match the length of *lam*.

If 'auto', then 2nd derivative smoothing for 'numerical' dtypes, and L2/ridge smoothing for 'categorical' dtypes.

Custom penalties can be passed as a callable.

n_coefs [int] Number of coefficients contributed by the term to the model

istensor [bool] whether the term is a tensor product of sub-terms

isintercept [bool] whether the term is an intercept

hasconstraint [bool] whether the term has any constraints

info [dict] contains dict with the sufficient information to duplicate the term

See also:

LinearTerm() for developer details

Spline Term

pygam.terms.**s**(*feature*, *n_splines=20*, *spline_order=3*, *lam=0.6*, *penalties='auto'*, *constraints=None*, *dtype='numerical'*, *basis='ps'*, *by=None*, *edge_knots=None*, *verbose=False*)

creates an instance of a SplineTerm

feature [int] Index of the feature to use for the feature function.

n_splines [int] Number of splines to use for the feature function. Must be non-negative.

spline_order [int] Order of spline to use for the feature function. Must be non-negative.

lam [float or iterable of floats] Strength of smoothing penalty. Must be a positive float. Larger values enforce stronger smoothing.

If single value is passed, it will be repeated for every penalty.

If iterable is passed, the length of *lam* must be equal to the length of *penalties*

penalties [{'auto', 'derivative', 'l2', None} or callable or iterable] Type of smoothing penalty to apply to the term.

If an iterable is used, multiple penalties are applied to the term. The length of the iterable must match the length of *lam*.

If 'auto', then 2nd derivative smoothing for 'numerical' dtypes, and L2/ridge smoothing for 'categorical' dtypes.

Custom penalties can be passed as a callable.

constraints [{None, 'convex', 'concave', 'monotonic_inc', 'monotonic_dec'}] or callable or iterable

Type of constraint to apply to the term.

If an iterable is used, multiple penalties are applied to the term.

dtype [{'numerical', 'categorical'}] String describing the data-type of the feature.

basis [{'ps', 'cp'}] Type of basis function to use in the term.

'ps' : p-spline basis

'cp' [cyclic p-spline basis, useful for building periodic functions.] by default, the maximum and minimum of the feature values are used to determine the function's period.

to specify a custom period use argument *edge_knots*

edge_knots : optional, array-like of floats of length 2

these values specify minimum and maximum domain of the spline function.

in the case that *spline_basis="cp"*, *edge_knots* determines the period of the cyclic function.

when *edge_knots=None* these values are inferred from the data.

default: None

by [int, optional] Feature to use as a by-variable in the term.

For example, if *feature* = 2 *by* = 0, then the term will produce: x0 * f(x2)

n_coefs [int] Number of coefficients contributed by the term to the model

istensor [bool] whether the term is a tensor product of sub-terms

isintercept [bool] whether the term is an intercept

hasconstraint [bool] whether the term has any constraints

info [dict] contains dict with the sufficient information to duplicate the term

See also:

SplineTerm() for developer details

Factor Term

pygam.terms.**f**(*feature*, *lam=0.6*, *penalties='auto'*, *coding='one-hot'*, *verbose=False*)

creates an instance of a FactorTerm

feature [int] Index of the feature to use for the feature function.

lam [float or iterable of floats] Strength of smoothing penalty. Must be a positive float. Larger values enforce stronger smoothing.

If single value is passed, it will be repeated for every penalty.

If iterable is passed, the length of *lam* must be equal to the length of *penalties*

penalties [{'auto', 'derivative', 'l2', None} or callable or iterable] Type of smoothing penalty to apply to the term.

If an iterable is used, multiple penalties are applied to the term. The length of the iterable must match the length of *lam*.

If 'auto', then 2nd derivative smoothing for 'numerical' dtypes, and L2/ridge smoothing for 'categorical' dtypes.

Custom penalties can be passed as a callable.

coding [{'one-hot'} type of contrast encoding to use.] currently, only 'one-hot' encoding has been developed. this means that we fit one coefficient per category.

n_coefs [int] Number of coefficients contributed by the term to the model

istensor [bool] whether the term is a tensor product of sub-terms

isintercept [bool] whether the term is an intercept

hasconstraint [bool] whether the term has any constraints

info [dict] contains dict with the sufficient information to duplicate the term

See also:

FactorTerm() for developer details

Tensor Term

pygam.terms.**te**(**args*, ***kwargs*)

creates an instance of a TensorTerm

This is useful for creating interactions between features, or other terms.

*args : marginal Terms to combine into a tensor product

feature [list of integers] Indices of the features to use for the marginal terms.

n_splines [list of integers] Number of splines to use for each marginal term. Must be of same length as *feature*.

spline_order [list of integers] Order of spline to use for the feature function. Must be of same length as *feature*.

lam [float or iterable of floats] Strength of smoothing penalty. Must be a positive float. Larger values enforce stronger smoothing.

If single value is passed, it will be repeated for every penalty.

If iterable is passed, the length of *lam* must be equal to the length of *penalties*

penalties [{'auto', 'derivative', 'l2', None} or callable or iterable] Type of smoothing penalty to apply to the term.

If an iterable is used, multiple penalties are applied to the term. The length of the iterable must match the length of *lam*.

If 'auto', then 2nd derivative smoothing for 'numerical' dtypes, and L2/ridge smoothing for 'categorical' dtypes.

Custom penalties can be passed as a callable.

constraints [{None, 'convex', 'concave', 'monotonic_inc', 'monotonic_dec'}] or callable or iterable

Type of constraint to apply to the term.

If an iterable is used, multiple penalties are applied to the term.

dtype [list of {'numerical', 'categorical'}] String describing the data-type of the feature.

Must be of same length as *feature*.

basis [list of {'ps'}] Type of basis function to use in the term.

'ps' : p-spline basis

NotImplemented: 'cp' : cyclic p-spline basis

Must be of same length as *feature*.

by [int, optional] Feature to use as a by-variable in the term.

For example, if *feature* = [1, 2] *by* = 0, then the term will produce: x0 * te(x1, x2)

n_coefs [int] Number of coefficients contributed by the term to the model

istensor [bool] whether the term is a tensor product of sub-terms

isintercept [bool] whether the term is an intercept

hasconstraint [bool] whether the term has any constraints

info [dict] contains dict with the sufficient information to duplicate the term

See also:

TensorTerm() for developer details

6.4 Developer API

6.4.1 Terms

class pygam.terms.**Term**(*feature*, *lam=0.6*, *dtype='numerical'*, *fit_linear=False*, *fit_splines=True*, *penalties='auto'*, *constraints=None*, *verbose=False*)

Bases: pygam.core.Core

build_columns(*X*, *verbose=False*)

construct the model matrix columns for the term

Parameters

- **X** (*array-like*) – Input dataset with n rows
- **verbose** (*bool*) – whether to show warnings

Returns

Return type scipy sparse array with n rows

build_constraints(*coef*, *constraint_lam*, *constraint_l2*)

builds the GAM block-diagonal constraint matrix in quadratic form out of constraint matrices specified for each feature.

behaves like a penalty, but with a very large lambda value, ie 1e6.

Parameters

- **coefs** (*array-like containing the coefficients of a term*) –
- **constraint_lam** (*float,*) – penalty to impose on the constraint.

  typically this is a very large number.
- **constraint_l2** (*float,*) – loading to improve the numerical conditioning of the constraint matrix.

  typically this is a very small number.

Returns **C**

Return type sparse CSC matrix containing the model constraints in quadratic form

classmethod build_from_info(*info*)

build a Term instance from a dict

Parameters

- **cls** (*class*) –
- **info** (*dict*) – contains all information needed to build the term

Returns

Return type Term instance

build_penalties(*verbose=False*)
builds the GAM block-diagonal penalty matrix in quadratic form out of penalty matrices specified for each feature.

each feature penalty matrix is multiplied by a lambda for that feature.

so for m features: P = block_diag[lam0 * P0, lam1 * P1, lam2 * P2, ... , lamm * Pm]

Parameters **None** –

Returns **P**

Return type sparse CSC matrix containing the model penalties in quadratic form

compile(*X*, *verbose=False*)
method to validate and prepare data-dependent parameters

Parameters

- **X** (*array-like*) – Input dataset
- **verbose** (*bool*) – whether to show warnings

Returns

Return type None

hasconstraint
bool, whether the term has any constraints

info
get information about this term

Returns

Return type dict containing information to duplicate this term

isintercept

istensor

n_coefs
Number of coefficients contributed by the term to the model

class pygam.terms.**LinearTerm**(*feature*, *lam=0.6*, *penalties='auto'*, *verbose=False*)
Bases: *pygam.terms.Term*

build_columns(*X*, *verbose=False*)
construct the model matrix columns for the term

Parameters

- **X** (*array-like*) – Input dataset with n rows
- **verbose** (*bool*) – whether to show warnings

Returns

Return type scipy sparse array with n rows

build_constraints(*coef*, *constraint_lam*, *constraint_l2*)
builds the GAM block-diagonal constraint matrix in quadratic form out of constraint matrices specified for each feature.

behaves like a penalty, but with a very large lambda value, ie 1e6.

Parameters

- **coefs** (*array-like containing the coefficients of a term*) –
- **constraint_lam** (*float,*) – penalty to impose on the constraint.

  typically this is a very large number.
- **constraint_l2** (*float,*) – loading to improve the numerical conditioning of the constraint matrix.

  typically this is a very small number.

Returns **C**

Return type sparse CSC matrix containing the model constraints in quadratic form

classmethod build_from_info(*info*)
build a Term instance from a dict

Parameters

- **cls** (*class*) –
- **info** (*dict*) – contains all information needed to build the term

Returns

Return type Term instance

build_penalties(*verbose=False*)
builds the GAM block-diagonal penalty matrix in quadratic form out of penalty matrices specified for each feature.

each feature penalty matrix is multiplied by a lambda for that feature.

so for m features: P = block_diag[lam0 * P0, lam1 * P1, lam2 * P2, ... , lamm * Pm]

Parameters **None** –

Returns **P**

Return type sparse CSC matrix containing the model penalties in quadratic form

compile(*X*, *verbose=False*)
method to validate and prepare data-dependent parameters

Parameters

- **X** (*array-like*) – Input dataset
- **verbose** (*bool*) – whether to show warnings

Returns

Return type None

get_params(*deep=False*)
returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

hasconstraint
bool, whether the term has any constraints

info
get information about this term

Returns

Return type dict containing information to duplicate this term

isintercept

istensor

n_coefs
Number of coefficients contributed by the term to the model

set_params(*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters
- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have
- ****parameters** (*paramters to set*) –

Returns

Return type self

class pygam.terms.**SplineTerm**(*feature*, *n_splines=20*, *spline_order=3*, *lam=0.6*, *penalties='auto'*, *constraints=None*, *dtype='numerical'*, *basis='ps'*, *by=None*, *edge_knots=None*, *verbose=False*)
Bases: *pygam.terms.Term*

build_columns(*X*, *verbose=False*)
construct the model matrix columns for the term

Parameters

- **X** (*array-like*) – Input dataset with n rows
- **verbose** (*bool*) – whether to show warnings

Returns

Return type scipy sparse array with n rows

build_constraints(*coef*, *constraint_lam*, *constraint_l2*)
builds the GAM block-diagonal constraint matrix in quadratic form out of constraint matrices specified for each feature.

behaves like a penalty, but with a very large lambda value, ie 1e6.

Parameters

- **coefs** (*array-like containing the coefficients of a term*) –
- **constraint_lam** (*float,*) – penalty to impose on the constraint.

  typically this is a very large number.
- **constraint_l2** (*float,*) – loading to improve the numerical conditioning of the constraint matrix.

  typically this is a very small number.

Returns C

Return type sparse CSC matrix containing the model constraints in quadratic form

classmethod build_from_info(*info*)
build a Term instance from a dict

Parameters

- **cls** (*class*) –
- **info** (*dict*) – contains all information needed to build the term

Returns

Return type Term instance

build_penalties(*verbose=False*)
builds the GAM block-diagonal penalty matrix in quadratic form out of penalty matrices specified for each feature.

each feature penalty matrix is multiplied by a lambda for that feature.

so for m features: P = block_diag[lam0 * P0, lam1 * P1, lam2 * P2, ... , lamm * Pm]

Parameters **None** –

Returns P

Return type sparse CSC matrix containing the model penalties in quadratic form

compile(*X*, *verbose=False*)
method to validate and prepare data-dependent parameters

Parameters

- **X** (*array-like*) – Input dataset
- **verbose** (*bool*) – whether to show warnings

Returns

Return type None

get_params(*deep=False*)
returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

hasconstraint
bool, whether the term has any constraints

info
get information about this term

Returns

Return type dict containing information to duplicate this term

isintercept

istensor

n_coefs
Number of coefficients contributed by the term to the model

set_params(*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters
- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have
- ****parameters** (*paramters to set*) –

Returns

Return type self

class pygam.terms.**FactorTerm**(*feature*, *lam=0.6*, *penalties='auto'*, *coding='one-hot'*, *verbose=False*)
Bases: *pygam.terms.SplineTerm*

build_columns(*X*, *verbose=False*)
construct the model matrix columns for the term

Parameters

- **X** (*array-like*) – Input dataset with n rows
- **verbose** (*bool*) – whether to show warnings

Returns

Return type scipy sparse array with n rows

build_constraints(*coef*, *constraint_lam*, *constraint_l2*)
builds the GAM block-diagonal constraint matrix in quadratic form out of constraint matrices specified for each feature.

behaves like a penalty, but with a very large lambda value, ie 1e6.

Parameters

- **coefs** (*array-like containing the coefficients of a term*) –
- **constraint_lam** (*float,*) – penalty to impose on the constraint.

  typically this is a very large number.
- **constraint_l2** (*float,*) – loading to improve the numerical conditioning of the constraint matrix.

  typically this is a very small number.

Returns **C**

Return type sparse CSC matrix containing the model constraints in quadratic form

classmethod build_from_info(*info*)
build a Term instance from a dict

Parameters

- **cls** (*class*) –
- **info** (*dict*) – contains all information needed to build the term

Returns

Return type Term instance

build_penalties(*verbose=False*)
builds the GAM block-diagonal penalty matrix in quadratic form out of penalty matrices specified for each feature.

each feature penalty matrix is multiplied by a lambda for that feature.

so for m features: P = block_diag[lam0 * P0, lam1 * P1, lam2 * P2, ... , lamm * Pm]

Parameters **None** –

Returns **P**

Return type sparse CSC matrix containing the model penalties in quadratic form

compile(*X*, *verbose=False*)
method to validate and prepare data-dependent parameters

Parameters

- **X** (*array-like*) – Input dataset
- **verbose** (*bool*) – whether to show warnings

Returns

Return type None

get_params(*deep=False*)
returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

hasconstraint
bool, whether the term has any constraints

info
get information about this term

Returns

Return type dict containing information to duplicate this term

isintercept

istensor

n_coefs
Number of coefficients contributed by the term to the model

set_params(*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters
- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have

- ****parameters** (*paramters to set*) –

Returns

Return type self

class pygam.terms.**TensorTerm**(**args*, ***kwargs*)

Bases: *pygam.terms.SplineTerm*, pygam.terms.MetaTermMixin

build_columns(*X*, *verbose=False*)

construct the model matrix columns for the term

Parameters

- **X** (*array-like*) – Input dataset with n rows
- **verbose** (*bool*) – whether to show warnings

Returns

Return type scipy sparse array with n rows

build_constraints(*coef*, *constraint_lam*, *constraint_l2*)

builds the GAM block-diagonal constraint matrix in quadratic form out of constraint matrices specified for each feature.

Parameters

- **coefs** (*array-like containing the coefficients of a term*) –
- **constraint_lam** (*float,*) – penalty to impose on the constraint.

  typically this is a very large number.
- **constraint_l2** (*float,*) – loading to improve the numerical conditioning of the constraint matrix.

  typically this is a very small number.

Returns C

Return type sparse CSC matrix containing the model constraints in quadratic form

classmethod build_from_info(*info*)

build a TensorTerm instance from a dict

Parameters

- **cls** (*class*) –
- **info** (*dict*) – contains all information needed to build the term

Returns

Return type TensorTerm instance

build_penalties()

builds the GAM block-diagonal penalty matrix in quadratic form out of penalty matrices specified for each feature.

each feature penalty matrix is multiplied by a lambda for that feature.

so for m features: P = block_diag[lam0 * P0, lam1 * P1, lam2 * P2, ... , lamm * Pm]

Parameters **None** –

Returns P

Return type sparse CSC matrix containing the model penalties in quadratic form

compile(*X*, *verbose=False*)
method to validate and prepare data-dependent parameters

Parameters

- **X** (*array-like*) – Input dataset
- **verbose** (*bool*) – whether to show warnings

Returns

Return type None

get_params(*deep=False*)
returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

hasconstraint
bool, whether the term has any constraints

info
get information about this term

Returns

Return type dict containing information to duplicate this term

isintercept

istensor

n_coefs
Number of coefficients contributed by the term to the model

set_params(*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters
- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have
- ****parameters** (*paramters to set*) –

Returns

Return type self

class pygam.terms.**TermList**(**terms*, ***kwargs*)
Bases: pygam.core.Core, pygam.terms.MetaTermMixin

build_columns(*X*, *term=-1*, *verbose=False*)
construct the model matrix columns for the term

Parameters

- **X** (*array-like*) – Input dataset with n rows
- **verbose** (*bool*) – whether to show warnings

Returns

Return type scipy sparse array with n rows

build_constraints(*coefs*, *constraint_lam*, *constraint_l2*)
builds the GAM block-diagonal constraint matrix in quadratic form out of constraint matrices specified for each feature.

behaves like a penalty, but with a very large lambda value, ie 1e6.

Parameters

- **coefs** (*array-like containing the coefficients of a term*) –
- **constraint_lam** (*float,*) – penalty to impose on the constraint.

  typically this is a very large number.
- **constraint_l2** (*float,*) – loading to improve the numerical conditioning of the constraint matrix.

  typically this is a very small number.

Returns C

Return type sparse CSC matrix containing the model constraints in quadratic form

classmethod build_from_info(*info*)
build a TermList instance from a dict

Parameters

- **cls** (*class*) –
- **info** (*dict*) – contains all information needed to build the term

Returns

Return type TermList instance

build_penalties()
builds the GAM block-diagonal penalty matrix in quadratic form out of penalty matrices specified for each feature.

each feature penalty matrix is multiplied by a lambda for that feature.

so for m features: P = block_diag[lam0 * P0, lam1 * P1, lam2 * P2, . . . , lamm * Pm]

Parameters **None** –

Returns P

Return type sparse CSC matrix containing the model penalties in quadratic form

compile(*X*, *verbose=False*)
method to validate and prepare data-dependent parameters

Parameters

- **X** (*array-like*) – Input dataset
- **verbose** (*bool*) – whether to show warnings

Returns

Return type None

get_coef_indices(*i=-1*)
get the indices for the coefficients of a term in the term list

Parameters **i** (*int*) – by default *int=-1*, meaning that coefficient indices are returned for all terms in the term list

Returns

Return type list of integers

get_params(*deep=False*)
returns a dict of all of the object's user-facing parameters

Parameters **deep** (*boolean, default: False*) – when True, also gets non-user-facing paramters

Returns

Return type dict

hasconstraint
bool, whether the term has any constraints

info
get information about the terms in the term list

Returns

Return type dict containing information to duplicate the term list

n_coefs
Total number of coefficients contributed by the terms in the model

pop(*i=None*)
remove the ith term from the term list

Parameters **i** (*int, optional*) – term to remove from term list

by default the last term is popped.

Returns **term**

Return type *Term*

set_params(*deep=False*, *force=False*, ***parameters*)
sets an object's paramters

Parameters

- **deep** (*boolean, default: False*) – when True, also sets non-user-facing paramters
- **force** (*boolean, default: False*) – when True, also sets parameters that the object does not already have
- ****parameters** (*paramters to set*) –

Returns

Return type self

6.4.2 Distributions

Distributions

class pygam.distributions.**BinomialDist**(*levels=1*)
Bases: *pygam.distributions.Distribution*

Binomial Distribution

V(*mu*)
glm Variance function

computes the variance of the distribution

Parameters **mu** (*array-like of length n*) – expected values

Returns **variance**

Return type np.array of length n

deviance(*y*, *mu*, *scaled=True*)
model deviance

for a bernoulli logistic model, this is equal to the twice the negative loglikelihod.

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **scaled** (*boolean, default: True*) – whether to divide the deviance by the distribution scaled

Returns **deviances**

Return type np.array of length n

log_pdf(*y*, *mu*, *weights=None*)
computes the log of the pdf or pmf of the values under the current distribution

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **weights** (*array-like shape (n,) or None, default: None*) – sample weights if None, defaults to array of ones

Returns **pdf/pmf**

Return type np.array of length n

sample(*mu*)
Return random samples from this Normal distribution.

Parameters **mu** (*array-like of shape n_samples or shape (n_simulations, n_samples)*) – expected values

Returns **random_samples**

Return type np.array of same shape as mu

class pygam.distributions.**Distribution**(*name=None*, *scale=None*)
Bases: pygam.core.Core

phi(*y*, *mu*, *edof*, *weights*)
GLM scale parameter. for Binomial and Poisson families this is unity for Normal family this is variance

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **edof** (*float*) – estimated degrees of freedom

- **weights** (*array-like shape (n,) or None, default: None*) – sample weights if None, defaults to array of ones

Returns **scale**

Return type estimated model scale

sample(*mu*)
Return random samples from this distribution.

Parameters **mu** (*array-like of shape n_samples or shape (n_simulations, n_samples)*) – expected values

Returns **random_samples**

Return type np.array of same shape as mu

class pygam.distributions.**GammaDist**(*scale=None*)
Bases: *pygam.distributions.Distribution*

Gamma Distribution

V(*mu*)
glm Variance function

computes the variance of the distribution

Parameters **mu** (*array-like of length n*) – expected values

Returns **variance**

Return type np.array of length n

deviance(*y*, *mu*, *scaled=True*)
model deviance

for a bernoulli logistic model, this is equal to the twice the negative loglikelihod.

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **scaled** (*boolean, default: True*) – whether to divide the deviance by the distribution scaled

Returns **deviances**

Return type np.array of length n

log_pdf(*y*, *mu*, *weights=None*)
computes the log of the pdf or pmf of the values under the current distribution

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **weights** (*array-like shape (n,) or None, default: None*) – containing sample weights if None, defaults to array of ones

Returns **pdf/pmf**

Return type np.array of length n

sample(*mu*)
Return random samples from this Gamma distribution.

Parameters **mu** (*array-like of shape n_samples or shape (n_simulations, n_samples)*) – expected values

Returns **random_samples**

Return type np.array of same shape as mu

class pygam.distributions.**InvGaussDist**(*scale=None*)
Bases: *pygam.distributions.Distribution*

Inverse Gaussian (Wald) Distribution

V(*mu*)
glm Variance function

computes the variance of the distribution

Parameters **mu** (*array-like of length n*) – expected values

Returns **variance**

Return type np.array of length n

deviance(*y*, *mu*, *scaled=True*)
model deviance

for a bernoulli logistic model, this is equal to the twice the negative loglikelihod.

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **scaled** (*boolean, default: True*) – whether to divide the deviance by the distribution scaled

Returns **deviances**

Return type np.array of length n

log_pdf(*y*, *mu*, *weights=None*)
computes the log of the pdf or pmf of the values under the current distribution

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **weights** (*array-like shape (n,) or None, default: None*) – containing sample weights if None, defaults to array of ones

Returns **pdf/pmf**

Return type np.array of length n

sample(*mu*)
Return random samples from this Inverse Gaussian (Wald) distribution.

Parameters **mu** (*array-like of shape n_samples or shape (n_simulations, n_samples)*) – expected values

Returns **random_samples**

Return type np.array of same shape as mu

class pygam.distributions.**NormalDist**(*scale=None*)

Bases: *pygam.distributions.Distribution*

Normal Distribution

V(*mu*)

glm Variance function.

if Y ~ ExpFam(theta, scale=phi)

such that E[Y] = mu = b'(theta)

and Var[Y] = b''(theta) * phi / w

then we seek V(mu) such that we can represent Var[y] as a fn of mu: Var[Y] = V(mu) * phi

ie V(mu) = b''(theta) / w

Parameters **mu** (*array-like of length n*) – expected values

Returns **V(mu)**

Return type np.array of length n

deviance(*y*, *mu*, *scaled=True*)

model deviance

for a gaussian linear model, this is equal to the SSE

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **scaled** (*boolean, default: True*) – whether to divide the deviance by the distribution scaled

Returns **deviances**

Return type np.array of length n

log_pdf(*y*, *mu*, *weights=None*)

computes the log of the pdf or pmf of the values under the current distribution

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **weights** (*array-like shape (n,) or None, default: None*) – sample weights if None, defaults to array of ones

Returns **pdf/pmf**

Return type np.array of length n

sample(*mu*)

Return random samples from this Normal distribution.

Samples are drawn independently from univariate normal distributions with means given by the values in *mu* and with standard deviations equal to the *scale* attribute if it exists otherwise 1.0.

Parameters **mu** (*array-like of shape n_samples or shape (n_simulations, n_samples)*) – expected values

Returns **random_samples**

Return type np.array of same shape as mu

class pygam.distributions.**PoissonDist**

Bases: *pygam.distributions.Distribution*

Poisson Distribution

V(*mu*)

glm Variance function

computes the variance of the distribution

Parameters **mu** (*array-like of length n*) – expected values

Returns **variance**

Return type np.array of length n

deviance(*y*, *mu*, *scaled=True*)

model deviance

for a bernoulli logistic model, this is equal to the twice the negative loglikelihod.

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **scaled** (*boolean, default: True*) – whether to divide the deviance by the distribution scaled

Returns **deviances**

Return type np.array of length n

log_pdf(*y*, *mu*, *weights=None*)

computes the log of the pdf or pmf of the values under the current distribution

Parameters

- **y** (*array-like of length n*) – target values
- **mu** (*array-like of length n*) – expected values
- **weights** (*array-like shape (n,) or None, default: None*) – containing sample weights if None, defaults to array of ones

Returns **pdf/pmf**

Return type np.array of length n

sample(*mu*)

Return random samples from this Poisson distribution.

Parameters **mu** (*array-like of shape n_samples or shape (n_simulations, n_samples)*) – expected values

Returns **random_samples**

Return type np.array of same shape as mu

pygam.distributions.**divide_weights**(*V*)

pygam.distributions.**multiply_weights**(*deviance*)

6.4.3 Links

class pygam.links.**Link**(*name=None*)
Bases: pygam.core.Core

class pygam.links.**IdentityLink**
Bases: *pygam.links.Link*

gradient(*mu*, *dist*)
derivative of the link function wrt mu

Parameters

- **mu** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **grad**

Return type np.array of length n

link(*mu*, *dist*)
glm link function this is useful for going from mu to the linear prediction

Parameters

- **mu** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **lp**

Return type np.array of length n

mu(*lp*, *dist*)
glm mean function, ie inverse of link function this is useful for going from the linear prediction to mu

Parameters

- **lp** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **mu**

Return type np.array of length n

class pygam.links.**InvSquaredLink**
Bases: *pygam.links.Link*

gradient(*mu*, *dist*)
derivative of the link function wrt mu

Parameters

- **mu** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **grad**

Return type np.array of length n

link(*mu*, *dist*)
glm link function this is useful for going from mu to the linear prediction

Parameters

- **mu** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **lp**

Return type np.array of length n

mu(*lp*, *dist*)
glm mean function, ie inverse of link function this is useful for going from the linear prediction to mu

Parameters

- **lp** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **mu**

Return type np.array of length n

class pygam.links.**LogitLink**
Bases: *pygam.links.Link*

gradient(*mu*, *dist*)
derivative of the link function wrt mu

Parameters

- **mu** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **grad**

Return type np.array of length n

link(*mu*, *dist*)
glm link function this is useful for going from mu to the linear prediction

Parameters

- **mu** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **lp**

Return type np.array of length n

mu(*lp*, *dist*)
glm mean function, ie inverse of link function this is useful for going from the linear prediction to mu

Parameters

- **lp** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **mu**

Return type np.array of length n

class pygam.links.**LogLink**
Bases: *pygam.links.Link*

gradient(*mu*, *dist*)
derivative of the link function wrt mu

Parameters

- **mu** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **grad**

Return type np.array of length n

link(*mu*, *dist*)
glm link function this is useful for going from mu to the linear prediction

Parameters

- **mu** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **lp**

Return type np.array of length n

mu(*lp*, *dist*)
glm mean function, ie inverse of link function this is useful for going from the linear prediction to mu

Parameters

- **lp** (*array-like of legth n*) –
- **dist** (*Distribution instance*) –

Returns **mu**

Return type np.array of length n

6.4.4 Callbacks

class pygam.callbacks.**CallBack**(*name=None*)
Bases: pygam.core.Core

CallBack class

class pygam.callbacks.**Accuracy**
Bases: *pygam.callbacks.CallBack*

on_loop_start(*y*, *mu*)
runs the method at start of each optimization loop

Parameters

- **y** (*array-like of length n*) – target data
- **mu** (*array-like of length n*) – expected value data

Returns **accuracy**

Return type np.array of length n

class pygam.callbacks.**Coef**
Bases: *pygam.callbacks.CallBack*

on_loop_start(*gam*)
runs the method at start of each optimization loop

Parameters **gam** (*float*) –

Returns **coef_**

Return type list of floats

class pygam.callbacks.**Deviance**
Bases: *pygam.callbacks.CallBack*

Deviance CallBack class

on_loop_start(*gam*, *y*, *mu*)
runs the method at loop start

Parameters

- **gam** (*GAM instance*) –
- **y** (*array-like of length n*) – target data
- **mu** (*array-like of length n*) – expected value data

Returns **deviance**

Return type np.array of length n

class pygam.callbacks.**Diffs**
Bases: *pygam.callbacks.CallBack*

on_loop_end(*diff*)
runs the method at end of each optimization loop

Parameters **diff** (*float*) –

Returns **diff**

Return type float

pygam.callbacks.**validate_callback**(*callback*)
validates a callback's on_loop_start and on_loop_end methods

Parameters **callback** (*Callback object*) –

Returns

Return type validated callback

pygam.callbacks.**validate_callback_data**(*method*)
wraps a callback's method to pull the desired arguments from the vars dict also checks to ensure the method's arguments are in the vars dict

Parameters **method** (*callable*) –

Returns

Return type validated callable

6.4.5 Penalties

Penalty matrix generators

pygam.penalties.**concave**(*n*, *coef*)
Builds a penalty matrix for P-Splines with continuous features. Penalizes violation of a concave feature function.

Parameters

- **n** (*int*) – number of splines

- **coef** (*array-like*) – coefficients of the feature function

Returns penalty matrix

Return type sparse csc matrix of shape (n,n)

pygam.penalties.**convex**(*n*, *coef*)

Builds a penalty matrix for P-Splines with continuous features. Penalizes violation of a convex feature function.

Parameters

- **n** (*int*) – number of splines
- **coef** (*array-like*) – coefficients of the feature function

Returns penalty matrix

Return type sparse csc matrix of shape (n,n)

pygam.penalties.**convexity_**(*n*, *coef*, *convex=True*)

Builds a penalty matrix for P-Splines with continuous features. Penalizes violation of convexity in the feature function.

Parameters

- **n** (*int*) – number of splines
- **coef** (*array-like*) – coefficients of the feature function
- **convex** (*bool, default: True*) – whether to enforce convex, or concave functions

Returns penalty matrix

Return type sparse csc matrix of shape (n,n)

pygam.penalties.**derivative**(*n*, *coef*, *derivative=2*, *periodic=False*)

Builds a penalty matrix for P-Splines with continuous features. Penalizes the squared differences between basis coefficients.

Parameters

- **n** (*int*) – number of splines
- **coef** (*unused*) – for compatibility with constraints
- **derivative** (*int, default: 2*) – which derivative do we penalize. derivative is 1, we penalize 1st order derivatives, derivative is 2, we penalize 2nd order derivatives, etc

Returns penalty matrix

Return type sparse csc matrix of shape (n,n)

pygam.penalties.**l2**(*n*, *coef*)

Builds a penalty matrix for P-Splines with categorical features. Penalizes the squared value of each basis coefficient.

Parameters

- **n** (*int*) – number of splines
- **coef** (*unused*) – for compatibility with constraints

Returns penalty matrix

Return type sparse csc matrix of shape (n,n)

pygam.penalties.**monotonic_dec**(*n*, *coef*)
Builds a penalty matrix for P-Splines with continuous features. Penalizes violation of a monotonic decreasing feature function.

Parameters

- **n** (*int*) – number of splines
- **coef** (*array-like*) – coefficients of the feature function

Returns penalty matrix

Return type sparse csc matrix of shape (n,n)

pygam.penalties.**monotonic_inc**(*n*, *coef*)
Builds a penalty matrix for P-Splines with continuous features. Penalizes violation of a monotonic increasing feature function.

Parameters

- **n** (*int*) – number of splines
- **coef** (*array-like, coefficients of the feature function*) –

Returns penalty matrix

Return type sparse csc matrix of shape (n,n)

pygam.penalties.**monotonicity_**(*n*, *coef*, *increasing=True*)
Builds a penalty matrix for P-Splines with continuous features. Penalizes violation of monotonicity in the feature function.

Parameters

- **n** (*int*) – number of splines
- **coef** (*array-like*) – coefficients of the feature function
- **increasing** (*bool, default: True*) – whether to enforce monotic increasing, or decreasing functions

Returns penalty matrix

Return type sparse csc matrix of shape (n,n)

pygam.penalties.**none**(*n*, *coef*)
Build a matrix of zeros for features that should go unpenalized

Parameters

- **n** (*int*) – number of splines
- **coef** (*unused*) – for compatibility with constraints

Returns penalty matrix

Return type sparse csc matrix of shape (n,n)

pygam.penalties.**periodic**(*n*, *coef*, *derivative=2*, *_penalty=<function derivative>*)

pygam.penalties.**sparse_diff**(*array*, *n=1*, *axis=-1*)
A ported sparse version of np.diff. Uses recursion to compute higher order differences

Parameters

- **array** (*sparse array*) –
- **n** (*int, default: 1*) – differencing order

- **axis** (*int, default: -1*) – axis along which differences are computed

Returns **diff_array** – same shape as input array, but 'axis' dimension is smaller by 'n'.

Return type sparse array

pygam.penalties.**wrap_penalty**(*p*, *fit_linear*, *linear_penalty=0.0*)

tool to account for unity penalty on the linear term of any feature.

Example

p = wrap_penalty(derivative, fit_linear=True)(n, coef)

Parameters

- **p** (*callable.*) – penalty-matrix-generating function.
- **fit_linear** (*boolean.*) – whether the current feature has a linear term or not.
- **linear_penalty** (*float, default: 0.*) – penalty on the linear term

Returns **wrapped_p** – modified penalty-matrix-generating function

Return type callable

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

D

E

F

G

H

I

L

M

N

O

P

S

T

V

W