glum

QuantCo Inc.

May 06, 2024

CONTENTS

1 Installation 3

2 Getting Started: fitting a Lasso model 5

3 Yet another GLM package? 9

4 Benchmarks against glmnet and H2O 13

5 Tutorials 19

6 Contributing and Development 67

7 An introduction to the algorithms used in `glum` 73

8 glum package 77

9 Changelog 141

Bibliography 153

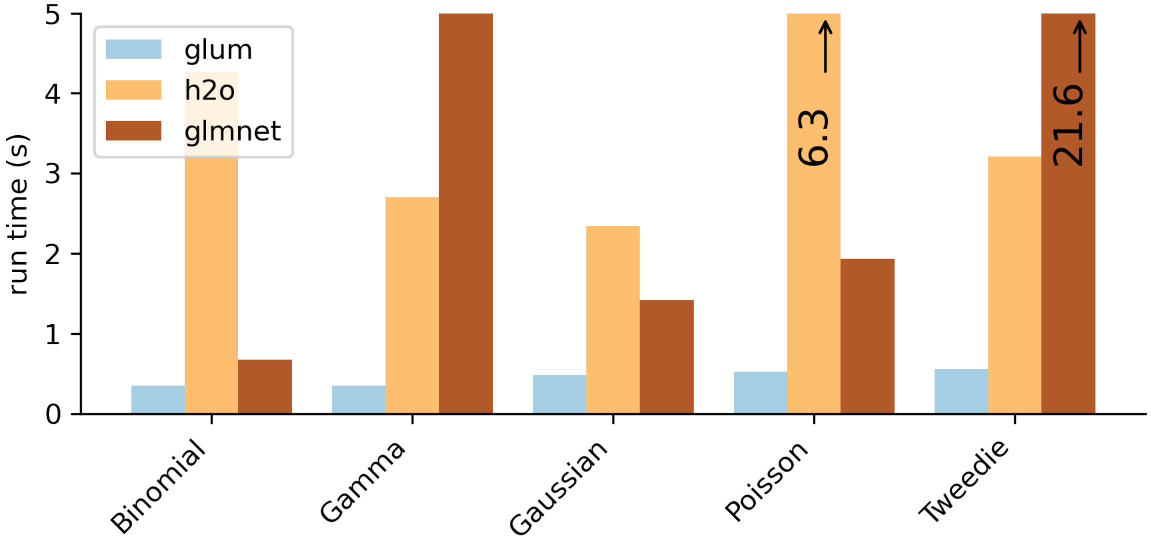
Python Module Index 155

Index 157

glum is a fast, modern, Python-first GLM estimation library. Generalized linear modeling (GLM) is a core statistical tool that includes many common methods like least-squares regression, Poisson regression and logistic regression as special cases. In addition to fitting basic GLMs, glum supports a wide range of features. These include:

- Built-in cross validation for optimal regularization, efficiently exploiting a "regularization path"
- L1 and elastic net regularization, which produce sparse and easily interpretable solutions
- L2 regularization, including variable matrix-valued (Tikhonov) penalties, which are useful in modeling correlated effects
- Normal, Poisson, logistic, gamma, and Tweedie distributions, plus varied and customizable link functions
- Dispersion and standard errors
- Box and linear inequality constraints, sample weights, offsets.
- A scikit-learn-like API to fit smoothly into existing workflows.

glum was also built with performance in mind. The following figure shows the runtime of a realistic example using an insurance dataset. For more details and other benchmarks, see the *Benchmarks* section.



We suggest visiting the *Installation* and *Getting Started* sections first.

CHAPTER ONE

INSTALLATION

You can install the package through conda:

```
conda install glum -c conda-forge
```

Head onwards to *Getting Started* to try it out!

CHAPTER

TWO

GETTING STARTED: FITTING A LASSO MODEL

The purpose of this tutorial is to show the basics of `glum`. It assumes a working knowledge of python, regularized linear models, and machine learning. The API is very similar to scikit-learn. After all, `glum` is based on a fork of scikit-learn.

If you have not done so already, please refer to our *installation instructions* for installing `glum`.

```
[1]: import pandas as pd
     import sklearn
     from sklearn.datasets import fetch_openml
     from glum import GeneralizedLinearRegressor, GeneralizedLinearRegressorCV
```

2.1 Data

We start by loading the King County housing dataset from openML and splitting it into training and test sets. For simplicity, we don't go into any details regarding exploration or data cleaning.

```
[2]: house_data = fetch_openml(name="house_sales", version=3, as_frame=True)

     # Use only select features
     X = house_data.data[
         [
             "bedrooms",
             "bathrooms",
             "sqft_living",
             "floors",
             "waterfront",
             "view",
             "condition",
             "grade",
             "yr_built",
         ]
     ].copy()

     # Targets
     y = house_data.target
```

```
[3]: X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(
         X, y, test_size = 0.3, random_state=5
     )
```

2.2 GLM basics: fitting and predicting using the normal family

We'll use `glum.GeneralizedLinearRegressor` to predict the house prices using the available predictors.

We set three key parameters:

- `family`: the family parameter specifies the distributional assumption of the GLM and, as a consequence, the loss function to be minimized. Accepted strings are 'normal', 'poisson', 'gamma', 'inverse.gaussian', and 'binomial'. You can also pass in an instantiated `glum` distribution (e.g. `glum.TweedieDistribution(1.5)`)
- `alpha`: the constant multiplying the penalty term that determines regularization strength. (*Note*: `GeneralizedLinearRegressor` also has an alpha-search option. See the `GeneralizedLinearRegressorCV` example below for details on how alpha-search works).
- `l1_ratio`: the elastic net mixing parameter (`0 <= l1_ratio <= 1`). For `l1_ratio = 0`, the penalty is the L2 penalty (ridge). `For l1_ratio = 1`, it is an L1 penalty (lasso). For `0 < l1_ratio < 1`, the penalty is a combination of L1 and L2.

To be precise, we will be minimizing the function with respect to the parameters, β:

$$\frac{1}{N}(\mathbf{X}\beta - y)^2 + \alpha\|\beta\|_1 \tag{2.1}$$

```
[4]: glm = GeneralizedLinearRegressor(family="normal", alpha=0.1, l1_ratio=1)
```

The `GeneralizedLinearRegressor.fit()` method follows typical sklearn API style and accepts two primary inputs:

1. `X`: the design matrix with shape `(n_samples, n_features)`.
2. `y`: the `n_samples` length array of target data.

```
[5]: glm.fit(X_train, y_train)
```

```
[5]: GeneralizedLinearRegressor(alpha=0.1, l1_ratio=1)
```

Once the model has been estimated, we can retrieve useful information using an sklearn-style syntax.

```
[6]: # retrieve the coefficients and the intercept
     coefs = glm.coef_
     intercept = glm.intercept_

     # use the model to predict on our test data
     preds = glm.predict(X_test)

     preds[0:5]
```

```
[6]: array([ 482648.22861066,  142902.68859995,  539452.61266391,
             569693.78048569, 1042446.90903451])
```

2.3 Fitting a GLM with cross validation

Now, we fit using automatic cross validation with `glum.GeneralizedLinearRegressorCV`. This mirrors the commonly used `cv.glmnet` function.

Some important parameters:

- `alphas`: for `GeneralizedLinearRegressorCV`, the best `alpha` will be found by searching along the regularization path. The regularization path is determined as follows:
    1. If `alpha` is an iterable, use it directly. All other parameters governing the regularization path are ignored.
    2. If `min_alpha` is set, create a path from `min_alpha` to the lowest alpha such that all coefficients are zero.
    3. If `min_alpha_ratio` is set, create a path where the ratio of `min_alpha / max_alpha = min_alpha_ratio`.
    4. If none of the above parameters are set, use a `min_alpha_ratio` of 1e-6.
- `l1_ratio`: for `GeneralizedLinearRegressorCV`, if you pass `l1_ratio` as an array, the `fit` method will choose the best value of `l1_ratio` and store it as `self.l1_ratio_`.

[7]:
```
glmcv = GeneralizedLinearRegressorCV(
    family="normal",
    alphas=None,  # default
    min_alpha=None,  # default
    min_alpha_ratio=None,  # default
    l1_ratio=[0, 0.5, 1.0],
    fit_intercept=True,
    max_iter=150
)
glmcv.fit(X_train, y_train)
print(f"Chosen alpha:    {glmcv.alpha_}")
print(f"Chosen l1 ratio: {glmcv.l1_ratio_}")
```

```
Chosen alpha:    0.00032745491628777320
Chosen l1 ratio: 0.0
```

Congratulations! You have finished our getting started tutorial. If you wish to learn more, please see our other tutorials for more advanced topics like Poisson, Gamma, and Tweedie regression, high dimensional fixed effects, and spatial smoothing using Tikhonov regularization.

[ ]:
```

```

CHAPTER

THREE

YET ANOTHER GLM PACKAGE?

glum was inspired by a desire to have a fast, maintainable, Python-first library for fitting GLMs with an extensive feature set.

At the beginning, we thoroughly examined all the existing contenders. The two mostly feature-complete options were glmnet and H2O. In many ways, the R package "glmnet" is the gold standard for regularized glm implementations. However, it is missing several useful features like built-in support for Tweedie and Gamma distributions. It also suffers from impossible-to-maintain source and thus has frequent bugs and segfaults. Although Python-to-glmnet interfaces exist, none is complete and well maintained. We also looked into the H2O implementation. It's more feature-complete than glmnet, but there are serious integration issues with Python. As we discovered, there is also substantial room to improve performance beyond the level of glmnet or H2O.

So we decided to improve an existing package. Which one? To be a bit more precise, the question we wanted to answer was "Which library will be the least work to make feature-complete, high performance and correct?" To decide, we began by building a suite of benchmarks to compare the different libraries, and compared the libraries in terms of speed, the number of benchmarks that ran successfully, and code quality. In the end, we went with the code from an sklearn pull request. We called it "sklearn-fork" and actually gave our code that name for quite a while too. sklearn-fork had decent, understandable code, converged in most situations, and included many of the features that we wanted. But it was slow. We figured it would be easier to speed up a functioning library than fix a broken but fast library. So we decided to start improving sklearn-fork. As a side note, a huge thank you to Christian for producing the baseline code for glum.

Ultimately, improving sklearn-fork seems to have been the right choice. We feel we have achieved our goals and glum is now *feature-complete*, *high-performance* and correct. However, over time, we uncovered more flaws in the optimizer than expected and, like most projects, building sklearn-fork into a feature-complete, fast, GLM library was a harder task than we predicted. When we started, sklearn-fork successfully converged for most problems. But, it was slow, taking hundreds or thousands of iteratively reweighted least squares (IRLS) iterations, many more than other similar libraries. Overall, the improvements we've made separate into three categories: algorithmic improvements, detailed software optimizations, and new features.

3.1 Algorithmic improvements

At the beginning, the lowest-hanging fruit came from debugging the implementation of IRLS and coordinate descent (CD) because those components were quite buggy and suboptimal. The algorithm we use is from [Yuan2012]. We started by understanding the paper and relating it back to the code. This led to a circuitous chase around the code base, an effort that paid off when we noticed a hard-coded value in the optimizer was far too high. Fixing this was a one-line change that gave us 2-4X faster convergence!

Another large algorithmic improvement to the optimizer came from centering the predictor matrix to have mean zero. Coordinate descent cycles through one feature at a time, which is a strategy that works poorly with non-centered predictors because changing any coefficient changes the mean. In several cases, zero-centering reduced the total number of IRLS iterations by a factor of two, while leaving solutions unchanged. As we discuss below, centering is nontrivial in the case of a sparse matrix because we don't want to modify the zero entries and destroy the sparsity. This was a major impetus for starting a tabular matrix handling library, tabmat, as an extension of `glum`.

Much later on, we made major improvements to the quality of the quadratic approximations for binomial, gamma, and Tweedie distributions, where the the original Hessian approximations turned out to be suboptimal. For the first couple months, we took for granted that the quadratic log-likelihood approximations from sklearn-fork were correct. However, after substantial investigation, it turned out that we were using a Fisher information matrix-based approximation to the hessian rather than the true Hessian. This was done in sklearn-fork because the Fisher information matrix (FIM) is guaranteed to be positive definite for any link function or distribution, a necessary condition for guaranteed convergence. However, in cases where the true Hessian is also positive definite, using it will result in much faster convergence. It turned out that switching to using the true Hessian for these special cases (linear, Poisson, gamma, logistic regression and Tweedie regression for $1 < p < 2$) gave huge reductions in the number of IRLS iterations. Some gamma regression problems dropped from taking 50-100 iterations to taking just 5-10 iterations.

Other important improvements:

- Using numerically stable log-likelihood, gradient and hessian formulas for the binomial distribution. In the naive version, we encounter floating point infinities for large parameter values in intermediate calculations.
- Exploring the use of an ADMM iterative L1 solver compared to our current CD solver. We ended up sticking with CD. This helped identify some crucial differences between glum and H2O, which uses an ADMM solver.
- Active set iteration where we use heuristics to improve performance in L1-regularized problems by predicting, at the beginning of each iteration, which coefficients are likely to remain zero. This effectively reduces the set of predictors and significantly improves performance in severely L1-regularized problems.
- Making sure that we could correctly compare objective functions between libraries. The meaning of the regularization strength varies depending on the constant factors that multiply the log-likelihood.

3.2 Software optimizations

Substantial performance improvements came from many places.

- Removing redundant calculations and storing intermediate results to re-use later. The line search step had a particularly large number of such optimization opportunities.
- Cython-izing the coordinate descent implementation based on a version from sklearn's Lasso implementation. Several optimizations were possible even beyond the sklearn Lasso implementation and we hope to contribute some of these upstream.
- Hand-optimizing the formulas and Cython-izing the code for common distributions' log likelihood, gradients, and hessians. We did this for normal, Poisson, gamma, Tweedie, binomial distributions.

The largest performance improvements have come from better tabular matrix handling. Initially, we were only handling uniformly dense or sparse matrices and using numpy and scipy.sparse to perform matrix operation. Now, we handle general "split" matrices that can be represented by a combination of dense, sparse, and categorical subcomponents. In addition, we built a `StandardizedMatrix` which handles the offsetting and multiplication needed to standardize a matrix to have mean zero and standard deviation one. We store the offsets and multipliers to perform this operation without modifying the underlying matrices.

We took our first step into developing custom matrix classes when we realized that even the pure dense and sparse matrix implementations were suboptimal. The default scipy.sparse matrix-multiply and matrix-vector product implementations are not parallel. Furthermore, many matrix-vector products only involve a small subset of rows or columns. As a result, we now have custom implementations of these operations that are parallelized and allow operating on a restricted set of rows and columns.

Before continuing, a quick summary of the only three matrix operations that we care about for GLM estimation:

- Matrix-vector products. `X.dot(v)` in numpy notation

- Transpose-matrix-vector products. `X.T.dot(v)`
- Sandwich products. `X.T @ diag(d) @ X`

As a matrix multiplication, the sandwich products are higher-dimensional operations than the matrix-vector products and, as such, are particularly expensive. Not only that, but the default implementation in numpy or scipy.sparse is going to be very inefficient. With dense numpy arrays, if we perform `X.T @ diag(d)`, that will allocate and create a whole new matrix that's just as large as the original `X` matrix. Then, we still need to perform a matrix multiply! As a result, we implemented a parallelized, cache-friendly, SIMD-optimized sandwich product operation that avoids the copy and performs the operation as a single matrix-multiply-like operation. We are in the process of contributing an implementation to the BLIS library.

The next big matrix optimization came from realizing that most data matrices are neither fully dense nor fully sparse. Some columns will be very sparse (e.g. number of parrots owned), some columns will be one-hot encoded categoricals (e.g. preferred parrot species) while other columns will be dense (e.g. volume in liters of the most recently seen parrot). So we built a SplitMatrix class that splits a matrix into dense and sparse subcomponents. A threshold of around 90% sparsity seems to be about the level at which it is beneficial to use a simple CSR sparse matrix instead of a dense matrix. The benefit of this split matrix was large, improving performance across all the matrix operations by 2-5x.

Later on, we also added categorical matrix handling to the mix. Many categorical columns will be very sparse. If there are 100 evenly distributed categories, each column will have 99% sparse. However, simply treating them as a general sparse matrix is leaving a lot on the table. Beyond just being sparse, we know that every non-zero entry is a one and that every row has only a single non-zero column. This is particularly beneficial for sandwich products where the output ends up being diagonal. But, despite the clear gains, adding categorical matrices was quite a large undertaking. We needed to modify our data generation process to produce categoricals instead of one-hot-encoded columns, add and optimize each of our matrix operations for categoricals, and specify "sandwich" interactions between categorical matrices, sparse matrices, and dense matrices. The result was a large improvement in runtime, with some sandwich and matrix-transpose-dot operations sped up by more than an order of magnitude.

The end result of all these matrix optimizations is that we now have a fairly complete library for handling simple sandwich, dot and transpose-dot operations on a mix of dense, sparse and categorical matrices. This is perfect for most tabular data! So, we've split this component off into its own library, tabmat.

3.3 New Features

In addition to the heavy focus on optimization and algorithmic correctness, we've also added a few important features to glum beyond what was already available in sklearn-fork.

- Automatic cross validation and regularization path handling similar in behavior to glmnet.
- Linear inequality constraints on coefficients.
- A step size convergence criterion in addition to the typical gradient-norm based criterion.
- The binomial distribution, and as a result, L1 and L2-regularized logistic regression.
- Standard errors.

3.4 References

CHAPTER

FOUR

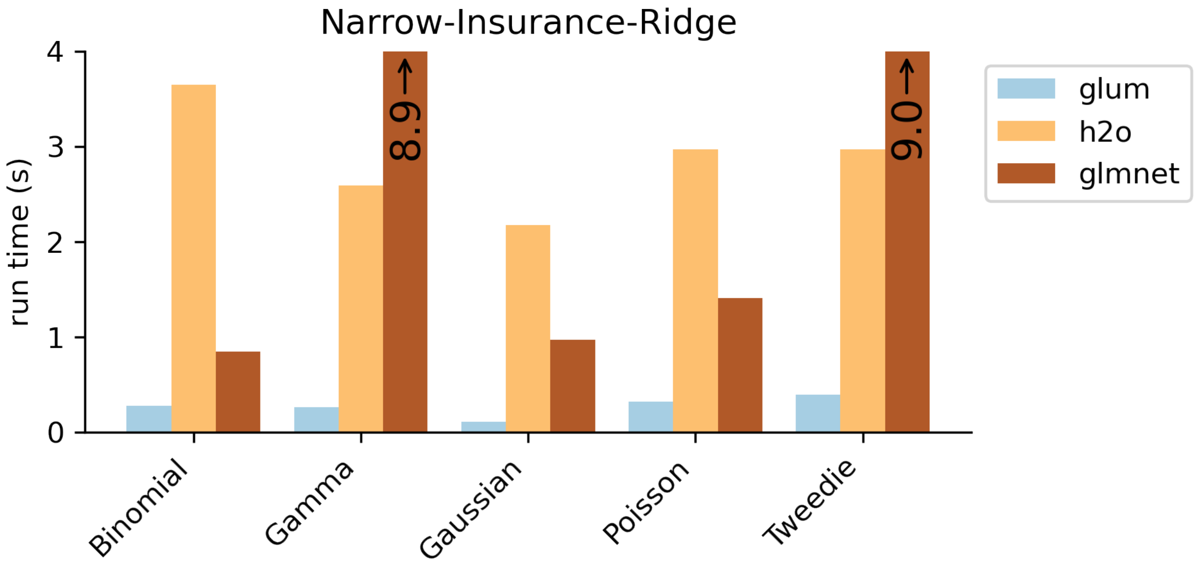
BENCHMARKS AGAINST GLMNET AND H2O

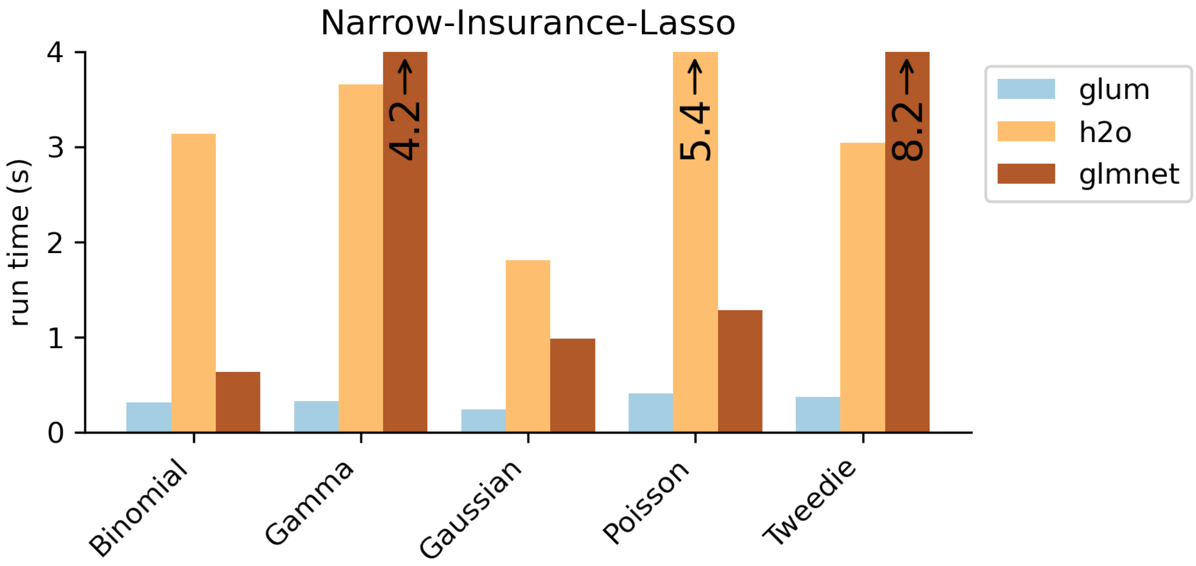
The following benchmarks were run on a MacBook Pro laptop with a quad-core Intel Core i5.

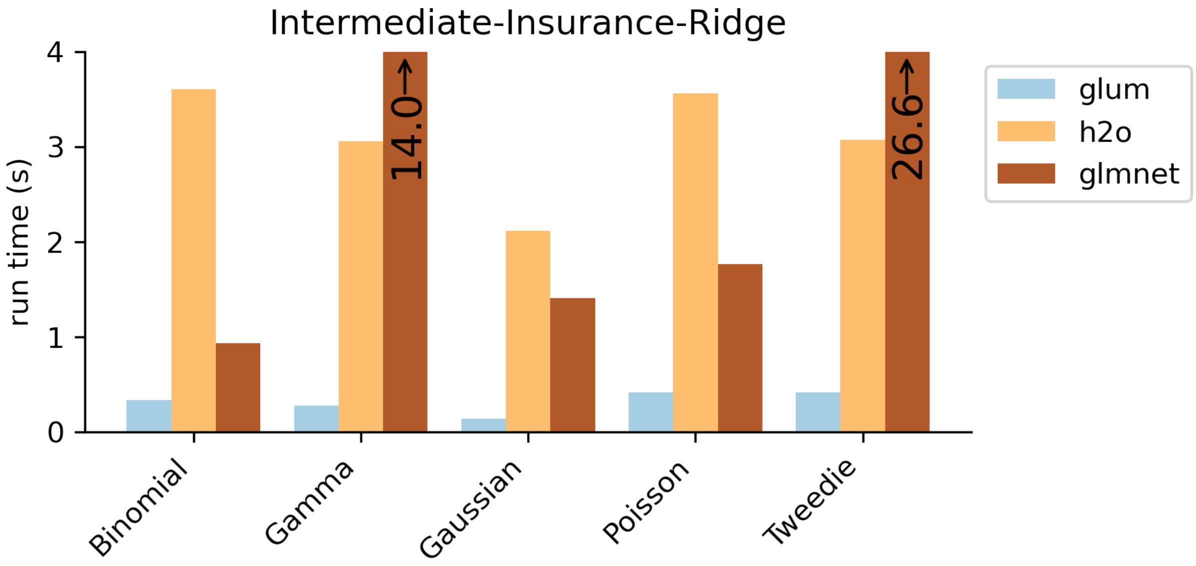
The title of each plot refers to both which dataset the benchmark was run on and whether a L2 ridge regression penalty or an L1 lasso penalty was included. For example "Narrow-Insurance-Ridge" was run on the `narrow-insurance` dataset with a ridge regression penalty. Each dataset/penalty pair is tested on five distributions that cover most of the common GLM types. The outcome variable is modified appropriately so that the behavior is similar to that expected for the distribution. For example, for the Poisson regression, we predict the number of claims per person. And for the binomial regression, we predict whether any given individual has ever had a claim. For the `housing` dataset, we only test three distributions because it does not contain count data that can be used as an outcome.

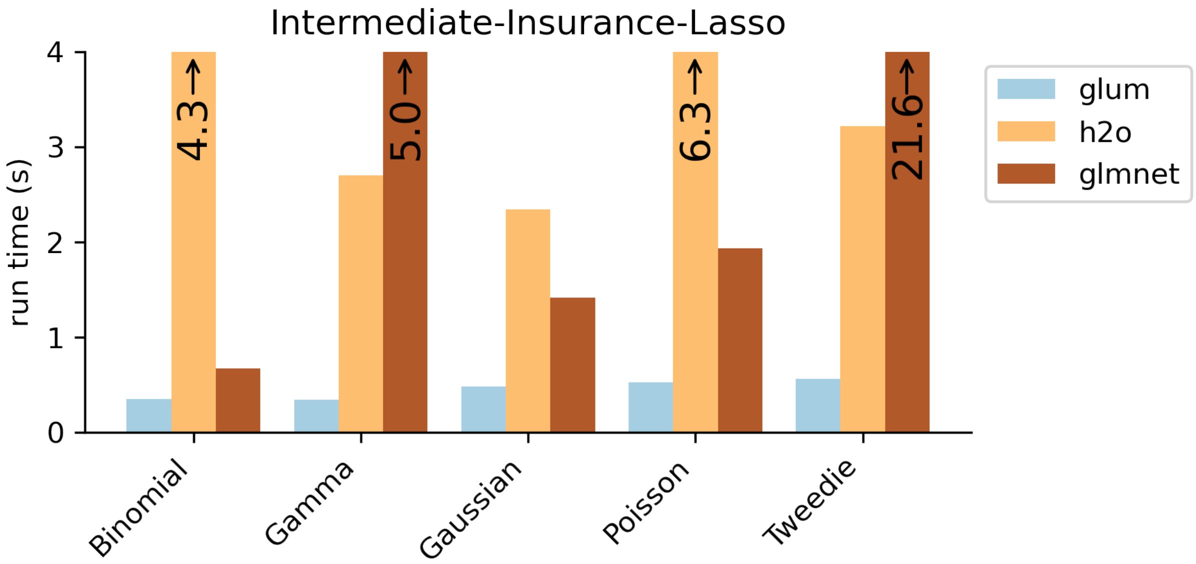
Note that glum was originally developed to solve problems where N >> K (number of observations is larger than the number of predictors), which is the case for the following benchmarks.

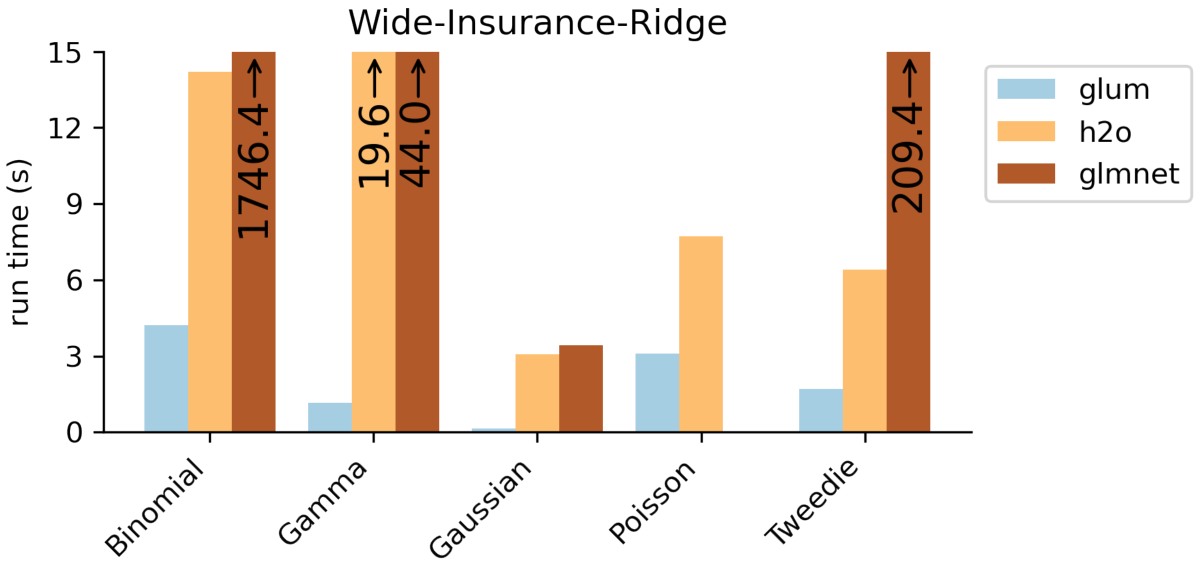
If a bar goes out of the range of the chart, the exact runtime is printed on the bar with an arrow indicating that the bar is truncated.



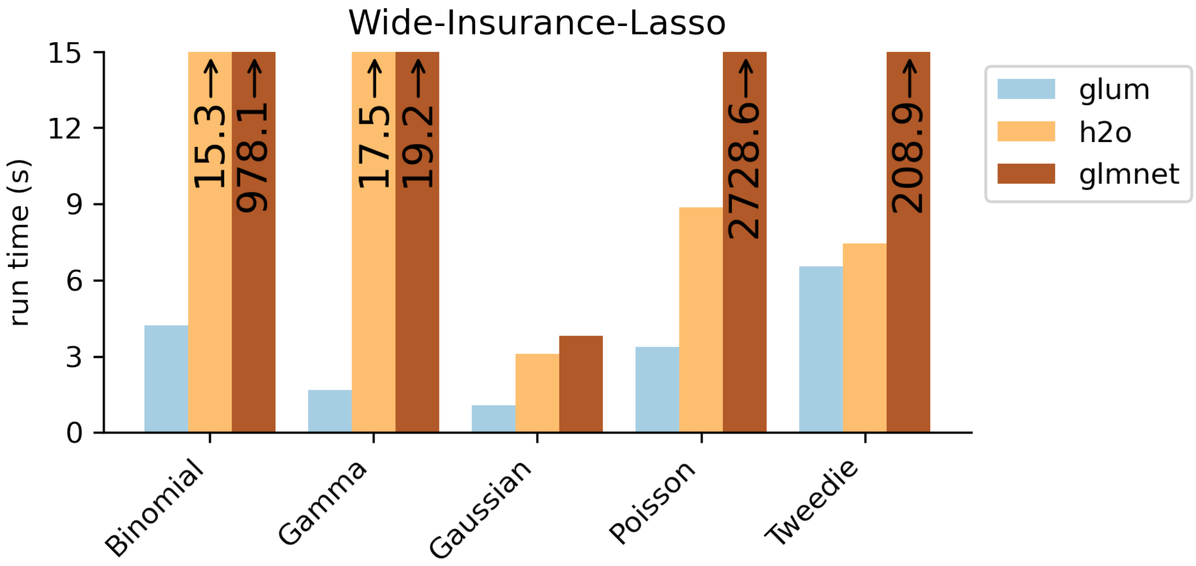

Narrow-Insurance-Lasso
run time (s)
0
1
2
3
4
4.2→
5.4→
8.2→
Binomial
Gamma
Gaussian
Poisson
Tweedie
glum
h2o
glmnet

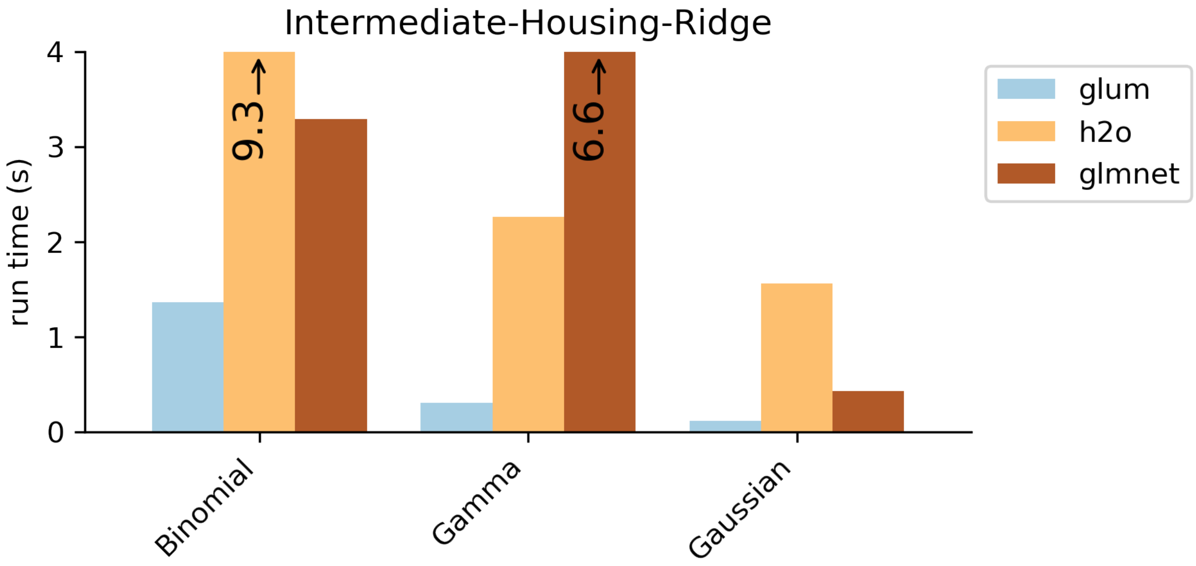

Intermediate-Insurance-Ridge
run time (s)
0
1
2
3
4
14.0→
26.6→
Binomial
Gamma
Gaussian
Poisson
Tweedie
glum
h2o
glmnet

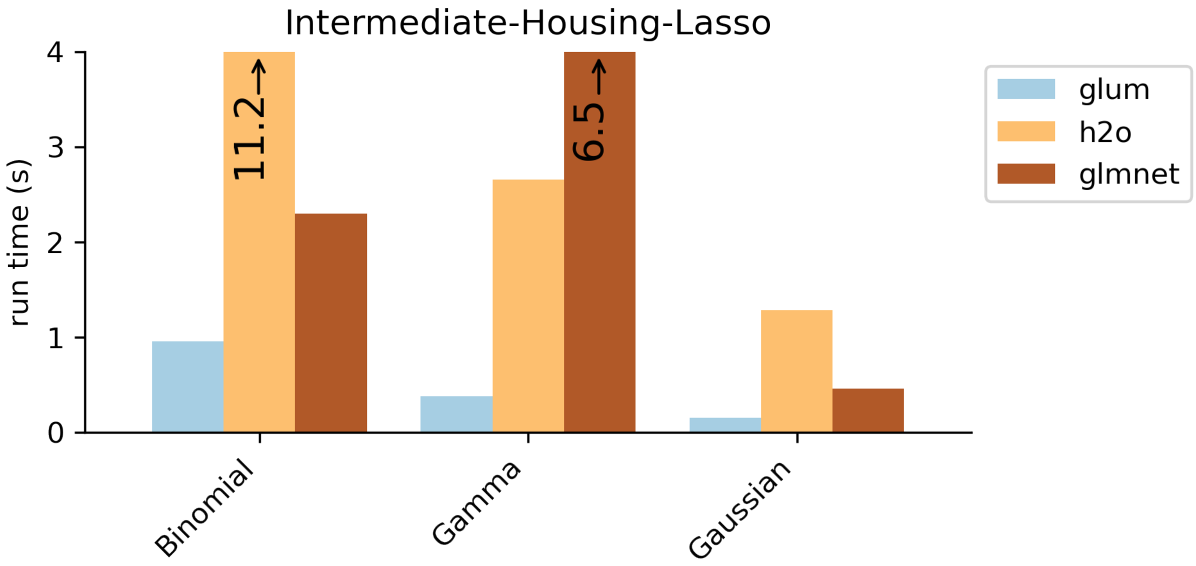




Note that the `r-glmnet` result for the `wide-insurance-ridge` Poisson benchmark is missing because `glmnet` did not converge after several hours of runtime.


Wide-Insurance-Lasso
run time (s)
0
3
6
9
12
15
15.3→
978.1→
17.5→
19.2→
2728.6→
208.9→
Binomial
Gamma
Gaussian
Poisson
Tweedie
glum
h2o
glmnet


Intermediate-Housing-Ridge
run time (s)
0
1
2
3
4
9.3→
6.6→
Binomial
Gamma
Gaussian
glum
h2o
glmnet


Intermediate-Housing-Lasso
run time (s)
0
1
2
3
4
11.2→
6.5→
Binomial
Gamma
Gaussian
glum
h2o
glmnet

CHAPTER

FIVE

TUTORIALS

5.1 GLM Tutorial: Poisson, Gamma, and Tweedie with French Motor Third-Party Liability Claims

Intro

This tutorial shows why and how to use Poisson, Gamma, and Tweedie GLMs on an insurance claims dataset using `glum`. It was inspired by, and closely mirrors, two other GLM tutorials that used this dataset:

1. An sklearn-learn tutorial, Tweedie regression on insurance claims, which was created for this (partially merged) sklearn PR that we based glum on
2. An R tutorial, Case Study: French Motor Third-Party Liability Claims with R code.

Background

Insurance claims are requests made by a policy holder to an insurance company for compensation in the event of a covered loss. When modeling these claims, the goal is often to estimate, per policy, the total claim amount per exposure unit. (i.e. number of claims × average amount per claim per year). This amount is also referred to as the pure premium.

Two approaches for modeling this value are:

1. Modeling the total claim amount per exposure directly
2. Modeling number of claims and claim amount separately with a frequency and a severity model

In this tutorial, we demonstrate both approaches. We start with the second option as it shows how to use two different families/distributions (Poisson and Gamma) within a GLM on a single dataset. We then show the first approach using a single poison-gamma Tweedie regressor (i.e. a Tweedie with power $p \in (1, 2)$)

5.1.1 Table of Contents

- 1. Load and Prepare Datasets from Openml.org
- *2. Frequency GLM - Poisson Distribution*
- *3. Severity GLM - Gamma Distribution*
- *4. Combined GLM - Tweedie Distribution*

[1]:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy.optimize as optimize
```

(continues on next page)

(continued from previous page)

```
import scipy.stats
from dask_ml.preprocessing import Categorizer
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import ShuffleSplit
from glum import GeneralizedLinearRegressor
from glum import TweedieDistribution

from load_transform import load_transform
```

5.1.2 1. Load and prepare datasets from Openml

back to table of contents

First, we load in our *dataset from openML* and apply several transformations. In the interest of simplicity, we do not include the data loading and preparation code in this notebook. Below is a list of further resources if you wish to explore further:

1. If you want to run the same code yourself, please see the helper functions here.
2. For a detailed description of the data, see here.
3. For an excellent exploratory data analysis, see the case study paper linked above.

Some important notes about the dataset post-transformation:

- Total claim amounts are aggregated per policy
- For ClaimAmountCut, the claim amounts (pre-aggregation) were cut at 100,000 per single claim. We choose to use this amount rather than the raw ClaimAmount. (100,000 is the 0.9984 quantile but claims > 100,000 account for 25% of the overall claim amount)
- We aggregate the total claim amounts per policy
- ClaimNb is the total number of claims per policy with claim amount greater zero
- VehPower, VehAge, and DrivAge are clipped and/or digitized into bins so that they can be used as categoricals later on

[2]:
```
df = load_transform()
with pd.option_context('display.max_rows', 10):
    display(df)
```

```
         ClaimNb  Exposure Area  VehPower  VehAge  DrivAge  BonusMalus  \
IDpol
1              0   0.10000    D         5       0        5          50
3              0   0.77000    D         5       0        5          50
5              0   0.75000    B         6       1        5          50
10             0   0.09000    B         7       0        4          50
11             0   0.84000    B         7       0        4          50
...          ...       ...  ...       ...     ...      ...         ...
6114326        0   0.00274    E         4       0        5          50
6114327        0   0.00274    E         4       0        4          95
6114328        0   0.00274    D         6       1        4          50
6114329        0   0.00274    B         4       0        5          50
6114330        0   0.00274    B         7       1        2          54
```

(continues on next page)

(continued from previous page)

```
        VehBrand   VehGas  Density Region  ClaimAmount  ClaimAmountCut
IDpol
1            B12  Regular     1217    R82          0.0             0.0
3            B12  Regular     1217    R82          0.0             0.0
5            B12   Diesel       54    R22          0.0             0.0
10           B12   Diesel       76    R72          0.0             0.0
11           B12   Diesel       76    R72          0.0             0.0
...          ...      ...      ...    ...          ...             ...
6114326      B12  Regular     3317    R93          0.0             0.0
6114327      B12  Regular     9850    R11          0.0             0.0
6114328      B12   Diesel     1323    R82          0.0             0.0
6114329      B12  Regular       95    R26          0.0             0.0
6114330      B12   Diesel       65    R72          0.0             0.0

[678013 rows x 13 columns]
```

5.1.3 2. Frequency GLM - Poisson distribution

back to Table of Contents

We start with the first part of our two part GLM - modeling the frequency of claims using a Poisson regression. Below, we give some background on why the Poisson family makes the most sense in this context.

2.1 Why Poisson distributions?

Poisson distributions are typically used to model the number of events occurring in a fixed period of time when the events occur independently at a constant rate. In our case, we can think of motor insurance claims as the events, and a unit of exposure (i.e. a year) as the fixed period of time.

To get more technical:

We define:

- z: number of claims
- w: exposure (time in years under risk)
- $y = \frac{z}{w}$: claim frequency per year
- X: feature matrix

The number of claims z is an integer, $z \in [0, 1, 2, 3, \ldots]$. Theoretically, a policy could have an arbitrarily large number of claims—very unlikely but possible. The simplest distribution for this range is a Poisson distribution $z \sim Poisson$. However, instead of z, we will model the frequency y. Nonetheless, this is still (scaled) Poisson distributed with variance inverse proportional to w, cf. wikipedia:Reproductive_EDM.

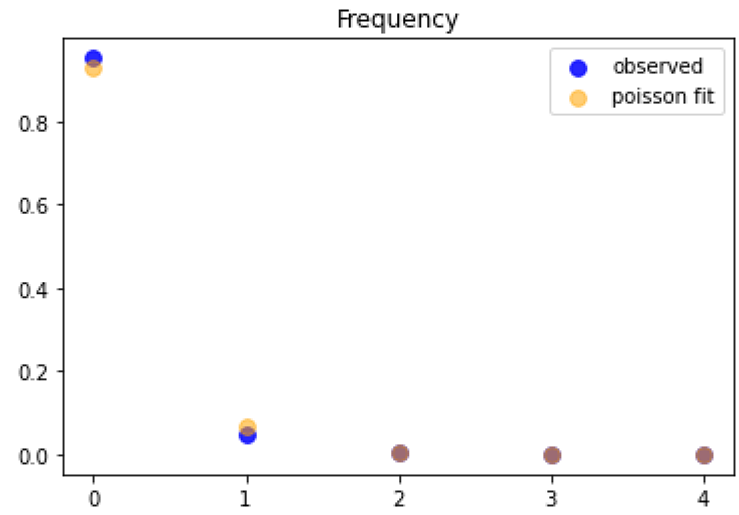
To verify our assumptions, we start by plotting the observed frequencies and a fitted Poisson distribution (Poisson regression with intercept only).

[3]:
```
# plt.subplots(figsize=(10, 7))
df_plot = (
    df.loc[:, ['ClaimNb', 'Exposure']].groupby('ClaimNb').sum()
    .assign(Frequency_Observed = lambda x: x.Exposure / df['Exposure'].sum())
)
```

(continues on next page)

(continued from previous page)

```
mean = df['ClaimNb'].sum() / df['Exposure'].sum()

x = range(5)
plt.scatter(x, df_plot['Frequency_Observed'].values, color="blue", alpha=0.85, s=60,␣
↪label='observed')
plt.scatter(x, scipy.stats.poisson.pmf(x, mean), color="orange", alpha=0.55, s=60, label=
↪"poisson fit")
plt.xticks(x)
plt.legend()
plt.title("Frequency");
```



This is a strong confirmation for the use of a Poisson when fitting!

2.2 Train and test frequency GLM

Now, we start fitting our model. We use claims frequency = claim number/exposure as our outcome variable. We then divide the dataset into training set and test set with a 9:1 random split.

Also, notice that we do not one hot encode our columns. Rather, we take advantage of `glum`'s integration with `tabmat`, which allows us to pass in categorical columns directly! `tabmat` will handle the encoding for us and even includes a handful of helpful matrix operation optimizations. We use the `Categorizer` from dask_ml to set our categorical columns as categorical dtypes and to ensure that the categories align in fitting and predicting.

[4]:
```
z = df['ClaimNb'].values
weight = df['Exposure'].values
y = z / weight # claims frequency

ss = ShuffleSplit(n_splits=1, test_size=0.1, random_state=42)
train, test = next(ss.split(y))

categoricals = ["VehBrand", "VehGas", "Region", "Area", "DrivAge", "VehAge", "VehPower"]
predictors = categoricals + ["BonusMalus", "Density"]
glm_categorizer = Categorizer(columns=categoricals)

X_train_p = glm_categorizer.fit_transform(df[predictors].iloc[train])
```

(continues on next page)

(continued from previous page)

```
X_test_p = glm_categorizer.transform(df[predictors].iloc[test])
y_train_p, y_test_p = y[train], y[test]
w_train_p, w_test_p = weight[train], weight[test]
z_train_p, z_test_p = z[train], z[test]
```

Now, we define our GLM using the `GeneralizedLinearRegressor` class from `glum`.

- `family='poisson'`: creates a Poisson regressor
- `alpha_search=True`: tells the GLM to search along the regularization path for the best alpha
- `l1_ratio = 1` tells the GLM to only use l1 penalty (not l2). `l1_ratio` is the elastic net mixing parameter. For `l1_ratio = 0`, the penalty is an L2 penalty. `For l1_ratio = 1`, it is an L1 penalty. For `0 < l1_ratio <` 1, the penalty is a combination of L1 and L2.

See the `GeneralizedLinearRegressor` class API documentation for more details.

Note: `glum` also supported a cross validation model GeneralizedLinearRegressorCV. However, because cross validation requires fitting many models, it is much slower and we don't demonstrate it in this tutorial.

[5]:

```
f_glm1 = GeneralizedLinearRegressor(family='poisson', alpha_search=True, l1_ratio=1, fit_
↪intercept=True)

f_glm1.fit(
    X_train_p,
    y_train_p,
    sample_weight=w_train_p
);

pd.DataFrame({'coefficient': np.concatenate(([f_glm1.intercept_], f_glm1.coef_))},
             index=['intercept'] + f_glm1.feature_names_).T
```

[5]:

```
             intercept  VehBrand__B1  VehBrand__B10  VehBrand__B11  \
coefficient  -4.269268     -0.003721      -0.010846       0.138466

             VehBrand__B12  VehBrand__B13  VehBrand__B14  VehBrand__B2  \
coefficient      -0.259298            0.0      -0.110712     -0.003604

             VehBrand__B3  VehBrand__B4  ...  VehAge__1  VehAge__2  \
coefficient      0.044075           0.0  ...   0.045494  -0.139428

             VehPower__4  VehPower__5  VehPower__6  VehPower__7  VehPower__8  \
coefficient    -0.070054    -0.028142          0.0          0.0     0.016531

             VehPower__9  BonusMalus   Density
coefficient     0.164711    0.026764  0.000004

[1 rows x 60 columns]
```

To measure our model's test and train performance, we use the deviance function for the Poisson family. We can get the total deviance function directly from `glum`'s distribution classes and divide it by the sum of our sample weight.

Note: a Poisson distribution is equivalent to a Tweedie distribution with power = 1.

[6]:

```
PoissonDist = TweedieDistribution(1)
print('training loss f_glm1: {}'.format(
```

(continues on next page)

(continued from previous page)

```
    PoissonDist.deviance(y_train_p, f_glm1.predict(X_train_p), sample_weight=w_train_p)/
↪np.sum(w_train_p)
))

print('test loss f_glm1: {}'.format(
      PoissonDist.deviance(y_test_p, f_glm1.predict(X_test_p), sample_weight=w_test_p)/
↪np.sum(w_test_p)))
```

```
training loss f_glm1: 0.45704947333555146
test loss f_glm1: 0.45793061314157685
```

A GLM with canonical link function (Normal - identity, Poisson - log, Gamma - 1/x, Binomial - logit) with an intercept term has the so called **balance property**. Neglecting small deviations due to an imperfect fit, on the training sample the results satisfy the equality:

$$\sum_{i \in training} w_i y_i = \sum_{i \in training} w_i \hat{\mu}_i$$

As expected, this property holds in our real data:

[7]:
```
# balance property of GLM with canonical link, like log-link for Poisson:
z_train_p.sum(), (f_glm1.predict(X_train_p) * w_train_p).sum()
```

[7]:
```
(23785, 23785.198509368805)
```

5.1.4 3. Severity GLM - Gamma distribution

back to Table of Contents

Now, we fit a GLM for the severity with the same features as the frequency model. The severity y is the average claim size. We define:

- z: total claim amount, single claims cut at 100,000
- w: number of claims (with positive claim amount!)
- $y = \frac{z}{w}$: severity

3.1 Why Gamma distributions

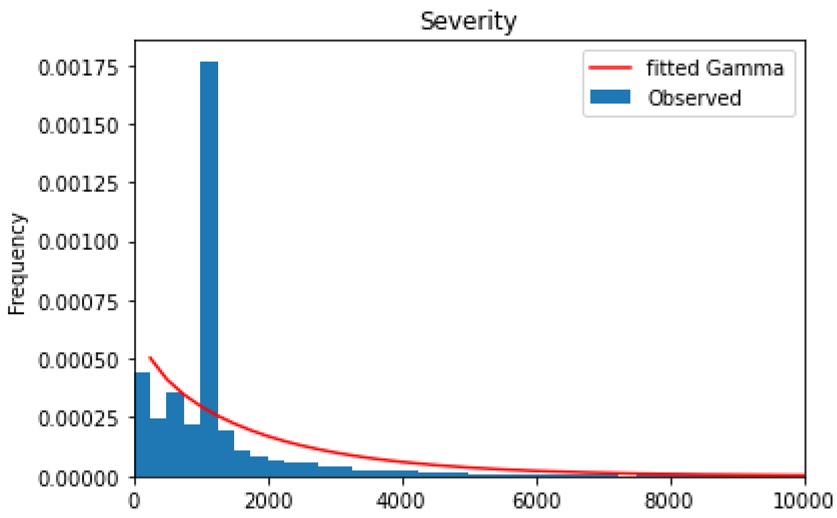
The severity y is a positive, real number, $y \in (0, \infty)$. Theoretically, especially for liability claims, one could have arbitrary large numbers—very unlikely but possible. A very simple distribution for this range is an Exponential distribution, or its generalization, a Gamma distribution $y \sim Gamma$. In the insurance industry, it is well known that the severity might be skewed by a few very large losses. It's common to model these tail losses separately so here we cut out claims larger than 100,000 to focus on modeling small and moderate claims.

[8]:
```
df_plot = (
    df.loc[:, ['ClaimAmountCut', 'ClaimNb']]
    .query('ClaimNb > 0')
    .assign(Severity_Observed = lambda x: x['ClaimAmountCut'] / df['ClaimNb'])
)

df_plot['Severity_Observed'].plot.hist(bins=400, density=True, label='Observed', )
```

(continues on next page)

(continued from previous page)

```
x = np.linspace(0, 1e5, num=400)
plt.plot(x,
         scipy.stats.gamma.pdf(x, *scipy.stats.gamma.fit(df_plot['Severity_Observed'],␣
↪floc=0)),
         'r-', label='fitted Gamma')
plt.legend()
plt.title("Severity");
plt.xlim(left=0, right = 1e4);
#plt.xticks(x);
```



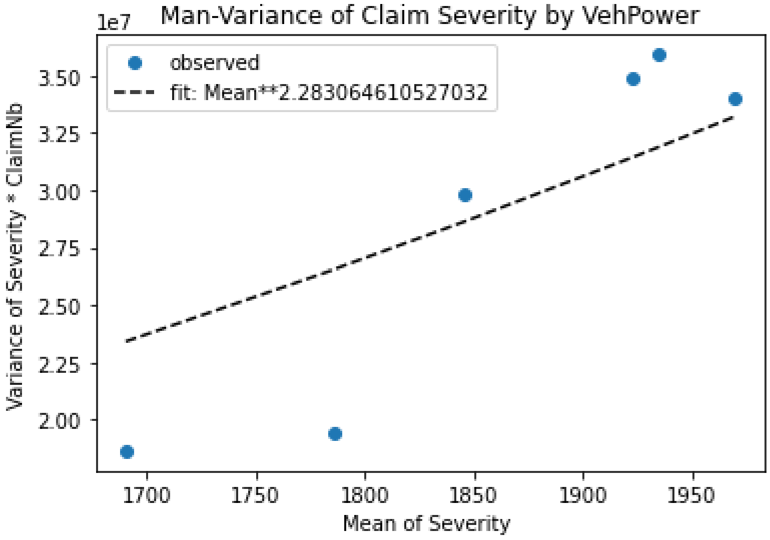
[9]:

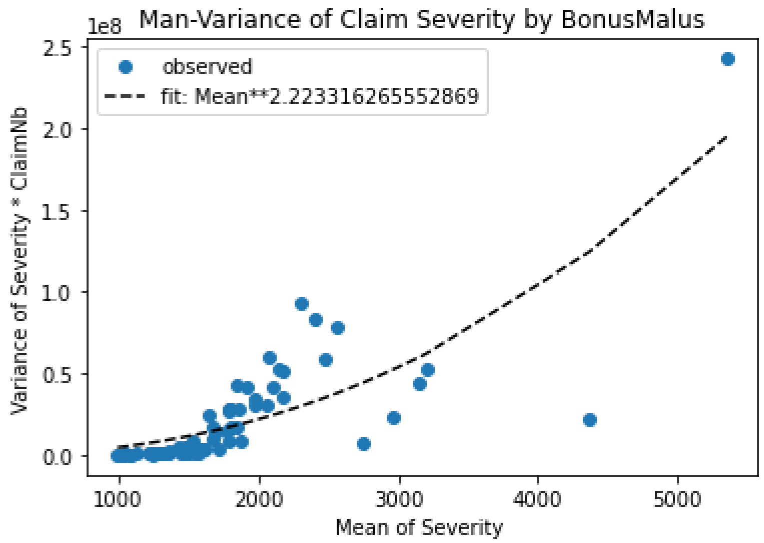
```
# Check mean-variance relationship for Gamma: Var[Y] = E[Y]^2 / Exposure
# Estimate Var[Y] and E[Y]
# Plot estimates Var[Y] vs E[Y]^s/Exposure
# Note: We group by VehPower and BonusMalus in order to have different E[Y].

def my_agg(x):
    """See https://stackoverflow.com/q/44635626"""
    x_sev = x['Sev']
    x_cnb = x['ClaimNb']
    n = x_sev.shape[0]
    names = {
        'Sev_mean': np.average(x_sev, sample_weight=x_cnb),
        'Sev_var': 1/(n-1) * np.sum((x_cnb/np.sum(x_cnb)) * (x_sev-np.average(x_sev,␣
↪sample_weight=x_cnb))**2),
        'ClaimNb_sum': x_cnb.sum()
    }
    return pd.Series(names, index=['Sev_mean', 'Sev_var', 'ClaimNb_sum'])

for col in ['VehPower', 'BonusMalus']:
    claims = df.groupby(col)['ClaimNb'].sum()
    df_plot = (df.loc[df[col].isin(claims[claims >= 4].index), :]
               .query('ClaimNb > 0')
               .assign(Sev = lambda x: x['ClaimAmountCut']/x['ClaimNb'])
               .groupby(col)
```

(continues on next page)

(continued from previous page)

```
               .apply(my_agg)
              )

    plt.plot(df_plot['Sev_mean'], df_plot['Sev_var'] * df_plot['ClaimNb_sum'], '.',
             markersize=12, label='observed')

    # fit: mean**p/claims
    p = optimize.curve_fit(lambda x, p: np.power(x, p),
                           df_plot['Sev_mean'].values,
                           df_plot['Sev_var'] * df_plot['ClaimNb_sum'],
                           p0 = [2])[0][0]
    df_fit = pd.DataFrame({'x': df_plot['Sev_mean'],
                           'y': np.power(df_plot['Sev_mean'], p)})
    df_fit = df_fit.sort_values('x')

    plt.plot(df_fit.x, df_fit.y,
             'k--', label='fit: Mean**{}'.format(p))
    plt.xlabel('Mean of Severity ')
    plt.ylabel('Variance of Severity * ClaimNb')
    plt.legend()
    plt.title('Man-Variance of Claim Severity by {}'.format(col))
    plt.show()
```





Great! A Gamma distribution seems to be an empirically reasonable assumption for this dataset.

Hint: If Y were normal distributed, one should see a horizontal line, because $Var[Y] = constant/Exposure$ and the fit should give $p \approx 0$.

3.2 Severity GLM with train and test data

We fit a GLM for the severity with the same features as the frequency model. We use the same categorizer as before.

Note:

- We filter out ClaimAmount == 0. The severity problem is to model claim amounts conditional on a claim having already been submitted. It seems reasonable to treat a claim of zero as equivalent to no claim at all. Additionally, zero is not included in the open interval $(0, \infty)$ support of the Gamma distribution.
- We use ClaimNb as sample weights.
- We use the same split in train and test data such that we can predict the final claim amount on the test set as the product of our Poisson claim number and Gamma claim severity GLMs.

[10]:

```
idx = df['ClaimAmountCut'].values > 0

z = df['ClaimAmountCut'].values
weight = df['ClaimNb'].values
# y = claims severity
y = np.zeros_like(z)  # zeros will never be used
y[idx] = z[idx] / weight[idx]

# we also need to represent train and test as boolean indices
itrain = np.zeros(y.shape, dtype='bool')
itest = np.zeros(y.shape, dtype='bool')
itrain[train] = True
itest[test] = True
# simplify life
itrain = idx & itrain
itest = idx & itest
```

(continues on next page)

(continued from previous page)

```
X_train_g = glm_categorizer.fit_transform(df[predictors].iloc[itrain])
X_test_g = glm_categorizer.transform(df[predictors].iloc[itest])
y_train_g, y_test_g = y[itrain], y[itest]
w_train_g, w_test_g = weight[itrain], weight[itest]
z_train_g, z_test_g = z[itrain], z[itest]
```

We fit our model with the same parameters before, but of course, this time we use `family=gamma`.

[11]:

```
s_glm1 = GeneralizedLinearRegressor(family='gamma', alpha_search=True, l1_ratio=1, fit_
↪intercept=True)
s_glm1.fit(X_train_g, y_train_g, sample_weight=weight[itrain])

pd.DataFrame({'coefficient': np.concatenate(([s_glm1.intercept_], s_glm1.coef_))},
             index=['intercept'] + s_glm1.feature_names_).T
```

[11]:

```
             intercept  VehBrand__B1  VehBrand__B10  VehBrand__B11  \
coefficient     7.3389     -0.034591       0.040528        0.13116

             VehBrand__B12  VehBrand__B13  VehBrand__B14  VehBrand__B2  \
coefficient       0.035838       0.100753      -0.073995     -0.033196

             VehBrand__B3  VehBrand__B4  ...  VehAge__1  VehAge__2  \
coefficient           0.0      0.049078  ...        0.0  -0.024827

             VehPower__4  VehPower__5  VehPower__6  VehPower__7  VehPower__8  \
coefficient    -0.009537    -0.089972     0.071376     0.009361    -0.042491

             VehPower__9  BonusMalus   Density
coefficient     0.051636    0.002365 -0.000001

[1 rows x 60 columns]
```

Again, we measure performance with the deviance of the distribution. We also compare against the simple arithmetic mean and include the mean absolute error to help understand the actual scale of our results.

Note: a Gamma distribution is equivalent to a Tweedie distribution with power = 2.

[26]:

```
GammaDist = TweedieDistribution(2)
print('training loss (deviance) s_glm1:     {}'.format(
    GammaDist.deviance(
        y_train_g, s_glm1.predict(X_train_g), sample_weight=w_train_g
    )/np.sum(w_train_g)
))
print('training mean absolute error s_glm1: {}'.format(
    mean_absolute_error(y_train_g, s_glm1.predict(X_train_g))
))

print('\ntesting loss s_glm1 (deviance):      {}'.format(
    GammaDist.deviance(
        y_test_g, s_glm1.predict(X_test_g), sample_weight=w_test_g
    )/np.sum(w_test_g)
))
```

(continues on next page)

(continued from previous page)

```
print('testing mean absolute error s_glm1:  {}'.format(
    mean_absolute_error(y_test_g, s_glm1.predict(X_test_g))
))

print('\ntesting loss Mean (deviance):        {}'.format(
    GammaDist.deviance(
        y_test_g, np.average(z_train_g, sample_weight=w_train_g)*np.ones_like(z_test_g),␣
↪sample_weight=w_test_g
    )/np.sum(w_test_g)
))
print('testing mean absolute error Mean:    {}'.format(
    mean_absolute_error(y_test_g, np.average(z_train_g, sample_weight=w_train_g)*np.ones_
↪like(z_test_g))
))
```

```
training loss (deviance) s_glm1:     1.29010461534461
training mean absolute error s_glm1: 1566.1785138646032

testing loss s_glm1 (deviance):      1.2975718597070154
testing mean absolute error s_glm1:  1504.4458958597086

testing loss Mean (deviance):        1.3115309309577132
testing mean absolute error Mean:    1689.205530922944
```

Even though the deviance improvement seems small, the improvement in mean absolute error is not! (In the insurance world, this will make a significant difference when aggregated over all claims).

3.3 Combined frequency and severity results

We put together the prediction of frequency and severity to get the predictions of the total claim amount per policy.

[13]:
```
#Put together freq * sev together
print("Total claim amount on train set, observed = {}, predicted = {}".
     format(df['ClaimAmountCut'].values[train].sum(),
            np.sum(df['Exposure'].values[train] * f_glm1.predict(X_train_p) * s_glm1.
↪predict(X_train_p)))
     )

print("Total claim amount on test set, observed = {}, predicted = {}".
     format(df['ClaimAmountCut'].values[test].sum(),
            np.sum(df['Exposure'].values[test] * f_glm1.predict(X_test_p) * s_glm1.
↪predict(X_test_p)))
     )
```

```
Total claim amount on train set, observed = 44594644.68, predicted = 44549152.42247057
Total claim amount on test set, observed = 4707551.37, predicted = 4946960.354743531
```

5.1.5 4. Combined GLM - Tweedie distribution

back to Table of Contents

Finally, to demonstrate an alternate approach to the combined frequency severity model, we show how we can model pure premium directly using a Tweedie regressor. Any Tweedie distribution with power $p \in (1, 2)$ is known as compound Poisson Gamma distribution

```
[14]: weight = df['Exposure'].values
df["PurePremium"] = df["ClaimAmountCut"] / df["Exposure"]
y = df["PurePremium"]

X_train_t = glm_categorizer.fit_transform(df[predictors].iloc[train])
X_test_t = glm_categorizer.transform(df[predictors].iloc[test])
y_train_t, y_test_t = y.iloc[train], y.iloc[test]
w_train_t, w_test_t = weight[train], weight[test]
```

For now, we just arbitrarily select 1.5 as the power parameter for our Tweedie model. However for a better fit we could include the power parameter in the optimization/fitting process, possibly via a simple grid search.

Note: notice how we pass a `TweedieDistribution` object in directly for the family parameter. While `glum` supports strings for common families, it is also possible to pass in a glum distribution directly.

```
[15]: TweedieDist = TweedieDistribution(1.5)
t_glm1 = GeneralizedLinearRegressor(family=TweedieDist, alpha_search=True, l1_ratio=1,␣
↪fit_intercept=True)
t_glm1.fit(X_train_t, y_train_t, sample_weight=w_train_t)


pd.DataFrame({'coefficient': np.concatenate(([t_glm1.intercept_], t_glm1.coef_))},
             index=['intercept'] + t_glm1.feature_names_).T
```

```
[15]:             intercept  VehBrand__B1  VehBrand__B10  VehBrand__B11  \
coefficient    2.88667     -0.064157            0.0       0.231868

             VehBrand__B12  VehBrand__B13  VehBrand__B14  VehBrand__B2  \
coefficient      -0.211061       0.054979      -0.270346     -0.071453

             VehBrand__B3  VehBrand__B4  ...  VehAge__1  VehAge__2  \
coefficient       0.00291      0.059324  ...   0.008117  -0.229906

             VehPower__4  VehPower__5  VehPower__6  VehPower__7  VehPower__8  \
coefficient    -0.111796    -0.123388     0.060757     0.005179    -0.021832

             VehPower__9  BonusMalus   Density
coefficient     0.208158    0.032508  0.000002

[1 rows x 60 columns]
```

Again, we use the distribution's deviance to measure model performance

```
[16]: print('training loss s_glm1: {}'.format(
    TweedieDist.deviance(y_train_t, t_glm1.predict(X_train_t), sample_weight=w_train_t)/
↪np.sum(w_train_t)))
```

(continues on next page)

(continued from previous page)

```
print('testing loss s_glm1:  {}'.format(
    TweedieDist.deviance(y_test_t, t_glm1.predict(X_test_t), sample_weight=w_test_t)/np.
↪sum(w_test_t)))
```

```
training loss s_glm1: 73.91371104577475
testing loss s_glm1:  72.35318912371723
```

Finally, we again show the total predicted vs. true claim amount on the training and test set

```
[17]: #Put together freq * sev together
print("Total claim amount on train set, observed = {}, predicted = {}".
     format(df['ClaimAmountCut'].values[train].sum(),
            np.sum(df['Exposure'].values[train] * t_glm1.predict(X_train_p)))
     )

print("Total claim amount on test set, observed = {}, predicted = {}".
     format(df['ClaimAmountCut'].values[test].sum(),
            np.sum(df['Exposure'].values[test] * t_glm1.predict(X_test_p)))
     )
```

```
Total claim amount on train set, observed = 44594644.68, predicted = 45027861.66007367
Total claim amount on test set, observed = 4707551.37, predicted = 4999381.03386664
```

In terms of the combined proximity to the true total claim amounts, the frequency severity model performed a bit better than Tweedie model. However, both approaches ultimately prove to be effective.

5.2 High Dimensional Fixed Effects with Rossman Sales Data

Intro

This tutorial demonstrates how to create models with high dimensional fixed effects using `glum`. Using `tabmat`, we can pass categorical variables with a large range of values. `glum` and `tabmat` will handle the creation of the one-hot-encoded design matrix.

In some real-world problems, we have used millions of categories. This would be impossible with a dense matrix. General-purpose sparse matrices like compressed sparse row (CSR) matrices help but still leave a lot on the table. For a categorical matrix, we know that each row has only a single non-zero value and that value is 1. These optimizations are implemented in `tabmat.CategoricalMatrix`.

Background

For this tutorial, we will be predicting sales for the European drug store chain Rossman. Specifically, we are tasked with predicting daily sales for future dates. Ideally, we want a model that can capture the many factors that influence stores sales – promotions, competition, school, holidays, seasonality, etc. As a baseline, we will start with a simple model that only uses a few basic predictors. Then, we will fit a model with a large number of fixed effects. For both models, we will use OLS with L2 regularization.

We will use a gamma distribution for our model. This choice is motivated by two main factors. First, our target variable, sales, is a positive real number, which matches the support of the gamma distribution. Second, it is expected that factors influencing sales are multiplicative rather than additive, which is better captured with a gamma regression than say, OLS.

Note: a few parts of this tutorial utilize local helper functions outside this notebook. If you wish to run the notebook on your own, you can find the rest of the code here.

5.2.1 Table of Contents

- *1. Data Loading and Feature Engineering*
- *2. Fit Baseline GLM*
- *3. GLM with High Dimensional Fixed Effects*
- *4. Plot Results*

[1]:
```
import os
from pathlib import Path

import altair as alt
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from dask_ml.impute import SimpleImputer
from dask_ml.preprocessing import Categorizer
from glum import GeneralizedLinearRegressor
from sklearn.pipeline import Pipeline

from feature_engineering import apply_all_transformations
from process_data import load_test, load_train, process_data

import sys
sys.path.append("../")
from metrics import root_mean_squared_percentage_error

pd.set_option("display.float_format", lambda x: "%.3f" % x)
pd.set_option('display.max_columns', None)
alt.data_transformers.enable("json")  # to allow for large plots
```

[1]:
```
DataTransformerRegistry.enable('json')
```

5.2.2 1. Data loading and feature engineering

back to table of contents

We start by loading in the raw data. If you have not yet processed the raw data, it will be done below. (Initial processing consists of some basic cleaning and renaming of columns.

Note: if you wish to run this notebook on your own, and have not done so already, please download the data from the Rossman Kaggle Challenge. This tutorial expects that it in a folder named “raw_data” under the same directory as the notebook.

1.1 Load

[2]:
```
if not all(Path(p).exists() for p in ["raw_data/train.csv", "raw_data/test.csv", "raw_
↪data/store.csv"]):
    raise Exception("Please download raw data into 'raw_data' folder")

if not all(Path(p).exists() for p in ["processed_data/train.parquet", "processed_data/
↪test.parquet"]):
    "Processed data not found. Processing data from raw data..."
    process_data()
    "Done"

df = load_train().sort_values(["store", "date"])
df = df.iloc[:int(.1*len(df))]
df.head()
```

[2]:
```
         store  day_of_week        date  sales  customers   open  promo  \
1016095      1            2 2013-01-01      0          0  False      0
1014980      1            3 2013-01-02   5530        668   True      0
1013865      1            4 2013-01-03   4327        578   True      0
1012750      1            5 2013-01-04   4486        619   True      0
1011635      1            6 2013-01-05   4997        635   True      0

        state_holiday  school_holiday  year  month store_type assortment  \
1016095             a               1  2013      1          c          a
1014980             0               1  2013      1          c          a
1013865             0               1  2013      1          c          a
1012750             0               1  2013      1          c          a
1011635             0               1  2013      1          c          a

         competition_distance  competition_open_since_month  \
1016095             1270.000                         9.000
1014980             1270.000                         9.000
1013865             1270.000                         9.000
1012750             1270.000                         9.000
1011635             1270.000                         9.000

         competition_open_since_year  promo2  promo2_since_week  \
1016095                     2008.000       0                NaN
1014980                     2008.000       0                NaN
1013865                     2008.000       0                NaN
1012750                     2008.000       0                NaN
1011635                     2008.000       0                NaN

         promo2_since_year promo_interval
1016095                NaN           None
1014980                NaN           None
1013865                NaN           None
1012750                NaN           None
1011635                NaN           None
```

1.2 Feature engineering

As mentioned earlier, we want our model to incorporate many factors that could influence store sales. We create a number of fixed effects to capture this information. These include fixed effects for:

- A certain number days before a school or state holiday
- A certain number days after a school or state holiday
- A certain number days before a promo
- A certain number days after a promo
- A certain number days before the store is open or closed
- A certain number days after the store is open or closed
- Each month for each store
- Each year for each store
- Each day of the week for each store

We also do several other transformations like computing the z score to eliminate outliers (in the next step)

[3]:
```
df = apply_all_transformations(df)
df.head()
```

[3]:
```
         store  day_of_week       date  sales  customers   open  promo  \
1016095      1            2 2013-01-01      0          0  False      0
1014980      1            3 2013-01-02   5530        668   True      0
1013865      1            4 2013-01-03   4327        578   True      0
1012750      1            5 2013-01-04   4486        619   True      0
1011635      1            6 2013-01-05   4997        635   True      0

        state_holiday  school_holiday  year  month store_type assortment  \
1016095             a               1  2013      1          c          a
1014980             0               1  2013      1          c          a
1013865             0               1  2013      1          c          a
1012750             0               1  2013      1          c          a
1011635             0               1  2013      1          c          a

         competition_distance  competition_open_since_month  \
1016095              1270.000                         9.000
1014980              1270.000                         9.000
1013865              1270.000                         9.000
1012750              1270.000                         9.000
1011635              1270.000                         9.000

         competition_open_since_year  promo2  promo2_since_week  \
1016095                     2008.000       0                NaN
1014980                     2008.000       0                NaN
1013865                     2008.000       0                NaN
1012750                     2008.000       0                NaN
1011635                     2008.000       0                NaN

         promo2_since_year promo_interval  age_quantile competition_open  \
1016095                NaN           None            -1            1.000
1014980                NaN           None            -1            1.000
```

(continues on next page)

(continued from previous page)

```
1013865                NaN           None            -1            1.000
1012750                NaN           None            -1            1.000
1011635                NaN           None            -1            1.000

         count open_lag_1 open_lag_2 open_lag_3 open_lead_1 open_lead_2  \
1016095      0        1.0        1.0        1.0        True        True
1014980      1      False        1.0        1.0        True        True
1013865      2       True      False        1.0        True        True
1012750      3       True       True      False        True       False
1011635      4       True       True       True       False        True

        open_lead_3  promo_lag_1  promo_lag_2  promo_lag_3  promo_lead_1  \
1016095        True        0.000        0.000        0.000         0.000
1014980        True        0.000        0.000        0.000         0.000
1013865       False        0.000        0.000        0.000         0.000
1012750        True        0.000        0.000        0.000         0.000
1011635        True        0.000        0.000        0.000         0.000

         promo_lead_2  promo_lead_3  school_holiday_lag_1  \
1016095         0.000         0.000                 0.000
1014980         0.000         0.000                 1.000
1013865         0.000         0.000                 1.000
1012750         0.000         1.000                 1.000
1011635         1.000         1.000                 1.000

         school_holiday_lag_2  school_holiday_lag_3  school_holiday_lead_1  \
1016095                 0.000                 0.000                  1.000
1014980                 0.000                 0.000                  1.000
1013865                 1.000                 0.000                  1.000
1012750                 1.000                 1.000                  1.000
1011635                 1.000                 1.000                  1.000

         school_holiday_lead_2  school_holiday_lead_3 state_holiday_lag_1  \
1016095                  1.000                  1.000                   0
1014980                  1.000                  1.000                   a
1013865                  1.000                  1.000                   0
1012750                  1.000                  1.000                   0
1011635                  1.000                  1.000                   0

        state_holiday_lag_2 state_holiday_lag_3 state_holiday_lead_1  \
1016095                   0                   0                    0
1014980                   0                   0                    0
1013865                   a                   0                    0
1012750                   0                   a                    0
1011635                   0                   0                    0

        state_holiday_lead_2 state_holiday_lead_3 store_day_of_week  \
1016095                    0                    0               1_2
1014980                    0                    0               1_3
1013865                    0                    0               1_4
1012750                    0                    0               1_5
1011635                    0                    0               1_6
```

(continues on next page)

(continued from previous page)

```
        store_month store_school_holiday store_state_holiday store_year  \
1016095         1_1                  1_1              1_True     1_2013
1014980         1_1                  1_1             1_False     1_2013
1013865         1_1                  1_1             1_False     1_2013
1012750         1_1                  1_1             1_False     1_2013
1011635         1_1                  1_1             1_False     1_2013

         zscore
1016095     NaN
1014980     NaN
1013865     NaN
1012750     NaN
1011635     NaN
```

1.3 Train vs. validation selection

Lastly, we split our data into training and validation sets. Kaggle provides a test set for the Rossman challenge, but it does not directly include outcome data (sales), so we do not use it for our tutorial. Instead, we simulate predicting future sales by taking the last 5 months of our training data as our validation set.

[4]:

```
validation_window = [pd.to_datetime("2015-03-15"), pd.to_datetime("2015-07-31")]
select_train = (df["sales"].gt(0) & df["date"].lt(validation_window[0]) & df["zscore"].
↪abs().lt(5)).to_numpy()

select_val = (
    df["sales"].gt(0)
    & df["date"].ge(validation_window[0])
    & df["date"].lt(validation_window[1])
).to_numpy()


(select_train.sum(), select_val.sum())
```

[4]:

```
(57876, 12502)
```

5.2.3 2. Fit baseline GLM

back to table of contents

We start with a simple model that uses only year, month, day of the week, and store as predictors. Even with these variables alone, we should still be able to capture a lot of valuable information. Year can capture overall sales trends, month can capture seasonality, week day can capture the variation in sales across the week, and store can capture locality. We will treat these all as categorical variables.

With the `GeneralizedLinearRegressor()` class, we can pass in `pandas.Categorical` variables directly without having to encode them ourselves. This is convenient, especially when we start adding more fixed effects. But it is very important that the categories are aligned between calls to `fit` and `predict`. One way of achieving this alignment is with a `dask_ml.preprocessing.Categorizer`. Note, however, that the `Categorizer` class fails to enforce category alignment if the input column is already a categorical data type.

You can reference the pandas documentation on Categoricals to learn more about how these data types work.

[5]:

```
baseline_features = ["year", "month", "day_of_week", "store"]
baseline_categorizer = Categorizer(columns=baseline_features)
baseline_glm = GeneralizedLinearRegressor(
    family="gamma",
    scale_predictors=True,
    l1_ratio=0.0,
    alphas=1e-1,
)
```

Fit the model making sure to process the data frame with the Categorizer first and inspect the coefficients.

[6]:

```
baseline_glm.fit(
    baseline_categorizer.fit_transform(df[select_train][baseline_features]),
    df.loc[select_train, "sales"]
)

pd.DataFrame(
    {'coefficient': np.concatenate(([baseline_glm.intercept_], baseline_glm.coef_))},
    index=['intercept'] + baseline_glm.feature_names_
).T
```

[6]:

```
             intercept  year__2013  year__2014  year__2015  month__1  \
coefficient      8.781      -0.010       0.008       0.003    -0.015   

             month__2  month__3  month__4  month__5  month__6  month__7  \
coefficient    -0.021    -0.019     0.020     0.008    -0.006    -0.002   

             month__8  month__9  month__10  month__11  month__12  \
coefficient    -0.023    -0.034     -0.030      0.022      0.116   

             day_of_week__1  day_of_week__2  day_of_week__3  day_of_week__4  \
coefficient           0.098           0.010          -0.014          -0.014   

             day_of_week__5  day_of_week__6  day_of_week__7  store__1  \
coefficient           0.017          -0.100           0.285    -0.156   

             store__2  store__3  store__4  store__5  store__6  store__7  \
coefficient    -0.136     0.024     0.202    -0.161    -0.087     0.158   

             store__8  store__9  store__10  store__11  store__12  store__13  \
coefficient    -0.094     0.002     -0.081      0.110      0.074     -0.126   

             store__14  store__15  store__16  store__17  store__18  store__19  \
coefficient     -0.088      0.009      0.082     -0.020      0.001     -0.008   

             store__20  store__21  store__22  store__23  store__24  store__25  \
coefficient      0.085     -0.088     -0.178     -0.088      0.183      0.274   

             store__26  store__27  store__28  store__29  store__30  store__31  \
coefficient      0.008      0.192     -0.098      0.059     -0.101     -0.051   

             store__32  store__33  store__34  store__35  store__36  store__37  \
coefficient     -0.225      0.125      0.102      0.198      0.185      0.046   
```

(continues on next page)

(continued from previous page)

```
             store__38  store__39  store__40  store__41  store__42  store__43  \
coefficient     -0.048     -0.153     -0.128     -0.085      0.234      0.017

             store__44  store__45  store__46  store__47  store__48  store__49  \
coefficient     -0.086     -0.101     -0.094      0.039     -0.243      0.058

             store__50  store__51  store__52  store__53  store__54  store__55  \
coefficient     -0.208      0.012      0.079     -0.109      0.099     -0.180

             store__56  store__57  store__58  store__59  store__60  store__61  \
coefficient      0.047      0.259     -0.017     -0.091      0.102     -0.149

             store__62  store__63  store__64  store__65  store__66  store__67  \
coefficient     -0.009      0.034      0.251     -0.122     -0.035      0.086

             store__68  store__69  store__70  store__71  store__72  store__73  \
coefficient      0.081      0.190      0.002      0.171     -0.189     -0.173

             store__74  store__75  store__76  store__77  store__78  store__79  \
coefficient     -0.023     -0.038      0.172      0.077     -0.266     -0.091

             store__80  store__81  store__82  store__83  store__84  store__85  \
coefficient      0.077      0.038      0.163     -0.248      0.369      0.019

             store__86  store__87  store__88  store__89  store__90  store__91  \
coefficient     -0.155     -0.060     -0.048     -0.047      0.130     -0.059

             store__92  store__93  store__94  store__95  store__96  store__97  \
coefficient     -0.047     -0.052      0.038      0.084     -0.121      0.007

             store__98  store__99  store__100  store__101  store__102  \
coefficient     -0.119     -0.111       0.108       0.069       0.038

             store__103  store__104  store__105  store__106  store__107  \
coefficient      -0.171       0.233      -0.175       0.136       0.058

             store__108  store__109  store__110  store__111  store__112
coefficient       0.284      -0.006      -0.174       0.018      -0.039
```

And let's predict for our test set with the caveat that we will predict 0 for days when the stores are closed!

[7]:

```
df.loc[lambda x: x["open"], "predicted_sales_baseline"] = baseline_glm.predict(
    baseline_categorizer.fit_transform(df.loc[lambda x: x["open"]][baseline_features])
)

df["predicted_sales_baseline"] = df["predicted_sales_baseline"].fillna(0)
df["predicted_sales_baseline"] = df["predicted_sales_baseline"]
```

We use root mean squared percentage error (RMSPE) as our performance metric. (Useful for thinking about error relative to total sales of each store).

[8]:

```
train_err = root_mean_squared_percentage_error(
    df.loc[select_train, "sales"], df.loc[select_train, "predicted_sales_baseline"]
)
val_err = root_mean_squared_percentage_error(
    df.loc[select_val, "sales"], df.loc[select_val, "predicted_sales_baseline"]
)
print(f'Training Error: {round(train_err, 2)}%')
print(f'Validation Error: {round(val_err, 2)}%')
```

```
Training Error: 27.96%
Validation Error: 29.57%
```

The results aren't bad for a start, but we can do better :)

5.2.4 3. GLM with high dimensional fixed effects

back to table of contents

Now, we repeat a similar process to above, but, this time, we take advantage of the full range of categoricals we created in our data transformation step. Since we will create a very large number of fixed effects, we may run into cases where our validation data has categorical values not seen in our training data. In these cases, Dask-ML's `Categorizer` will output null values when transforming the validation columns to the categoricals that were created on the training set. To fix this, we add Dask-ML's SimpleImputer to our pipeline.

[9]:

```
highdim_features = [
    "age_quantile",
    "competition_open",
    "open_lag_1",
    "open_lag_2",
    "open_lag_3",
    "open_lead_1",
    "open_lead_2",
    "open_lead_3",
    "promo_lag_1",
    "promo_lag_2",
    "promo_lag_3",
    "promo_lead_1",
    "promo_lead_2",
    "promo_lead_3",
    "promo",
    "school_holiday_lag_1",
    "school_holiday_lag_2",
    "school_holiday_lag_3",
    "school_holiday_lead_1",
    "school_holiday_lead_2",
    "school_holiday_lead_3",
    "school_holiday",
    "state_holiday_lag_1",
    "state_holiday_lag_2",
    "state_holiday_lag_3",
    "state_holiday_lead_1",
    "state_holiday_lead_2",
    "state_holiday_lead_3",
```

(continues on next page)

(continued from previous page)

```
    "state_holiday",
    "store_day_of_week",
    "store_month",
    "store_school_holiday",
    "store_state_holiday",
    "store_year",
]
highdim_categorizer = Pipeline([
    ("categorize", Categorizer(columns=highdim_features)),
    ("impute", SimpleImputer(strategy="most_frequent"))
])
highdim_glm = GeneralizedLinearRegressor(
    family="gamma",
    scale_predictors=True,
    l1_ratio=0.0, # only ridge
    alpha=1e-1,
)
```

For reference, we output the total number of predictors after fitting the model. We can see that the number getting a bit larger, so we don't print out the coefficients this time.

[10]:

```
highdim_glm.fit(
    highdim_categorizer.fit_transform(df[select_train][highdim_features]),
    df.loc[select_train, "sales"]
)

print(f"Number of predictors: {len(highdim_glm.feature_names_)}")
```

```
Number of predictors: 2771
```

[11]:

```
df.loc[lambda x: x["open"], "predicted_sales_highdim"] = highdim_glm.predict(
    highdim_categorizer.transform(df.loc[lambda x: x["open"]][highdim_features]),
)

df["predicted_sales_highdim"] = df["predicted_sales_highdim"].fillna(0)
df["predicted_sales_highdim"] = df["predicted_sales_highdim"]


train_err = root_mean_squared_percentage_error(
    df.loc[select_train, "sales"], df.loc[select_train, "predicted_sales_highdim"]
)
val_err = root_mean_squared_percentage_error(
    df.loc[select_val, "sales"], df.loc[select_val, "predicted_sales_highdim"]
)
print(f'Training Error: {round(train_err, 2)}%')
print(f'Validation Error: {round(val_err, 2)}%')
```

```
Training Error: 12.54%
Validation Error: 13.28%
```

From just the RMSPE, we can see a clear improvement from our baseline model.

5.2.5 4. Plot results

back to table of contents

Finally, to get a better look at our results, we make some plots.

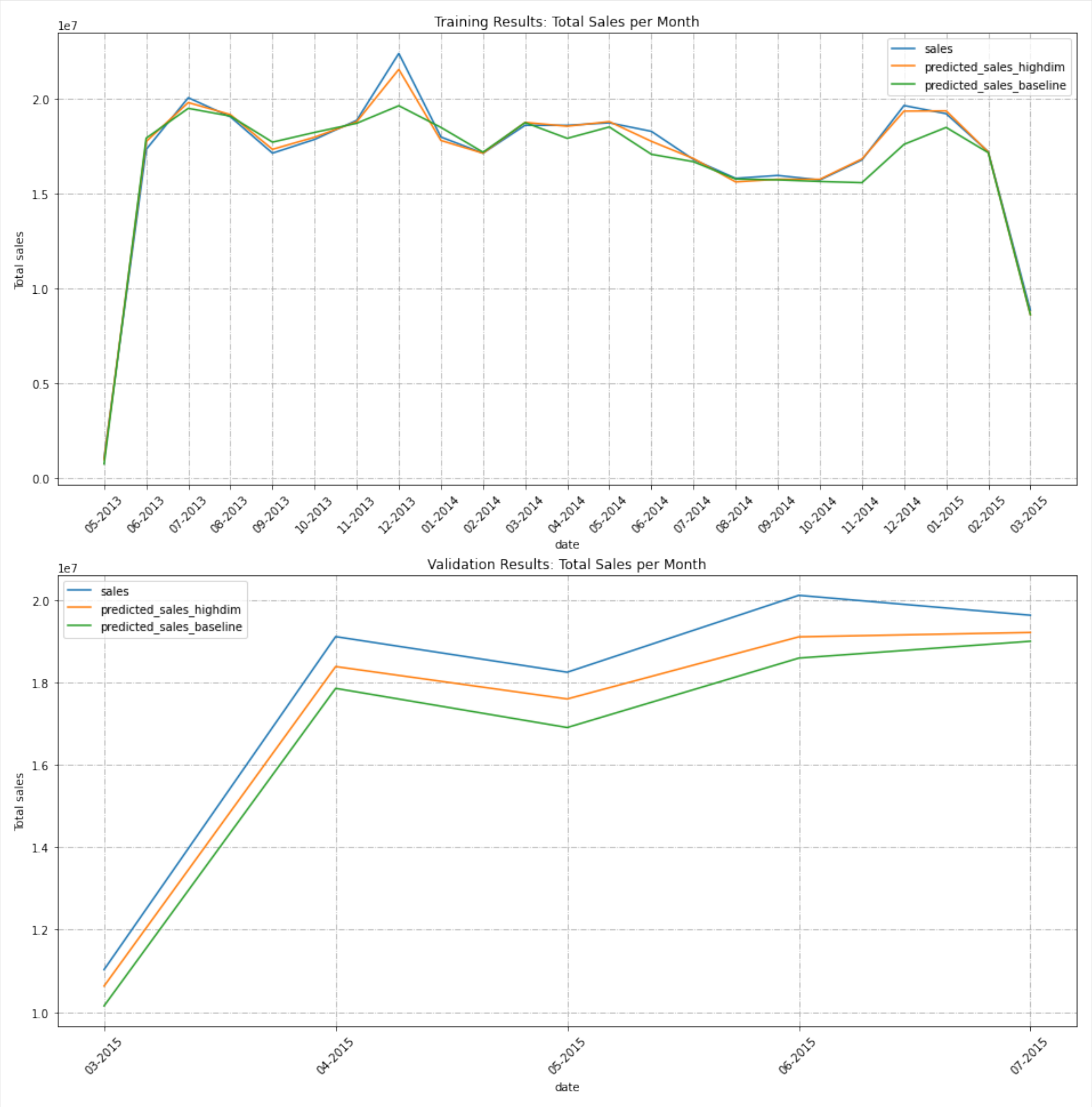
[12]:

```
sales_cols = ["sales", "predicted_sales_highdim", "predicted_sales_baseline"]
```

First, we plot true sales and the sales predictions from each model aggregated over month:

[13]:

```
_, axs = plt.subplots(2, 1, figsize=(16, 16))

for i, select in enumerate([select_train, select_val]):
    ax = axs[i]
    df_plot_date = df[select].groupby(
        ["year", "month"]
    ).agg("sum")[sales_cols].reset_index()

    year_month_date = df_plot_date['month'].map(str)+ '-' + df_plot_date['year'].map(str)
    df_plot_date['year_month'] = pd.to_datetime(year_month_date, format='%m-%Y').dt.
↪strftime('%m-%Y')
    df_plot_date.drop(columns= ["year", "month"], inplace=True)

    df_plot_date.plot(x="year_month", ax=ax)
    ax.set_xticks(range(len(df_plot_date)))
    ax.set_xticklabels(df_plot_date.year_month, rotation=45)
    ax.set_xlabel("date")
    ax.set_ylabel("Total sales")
    ax.grid(True, linestyle='-.')

axs[0].set_title("Training Results: Total Sales per Month")
axs[1].set_title("Validation Results: Total Sales per Month")
plt.show()
```



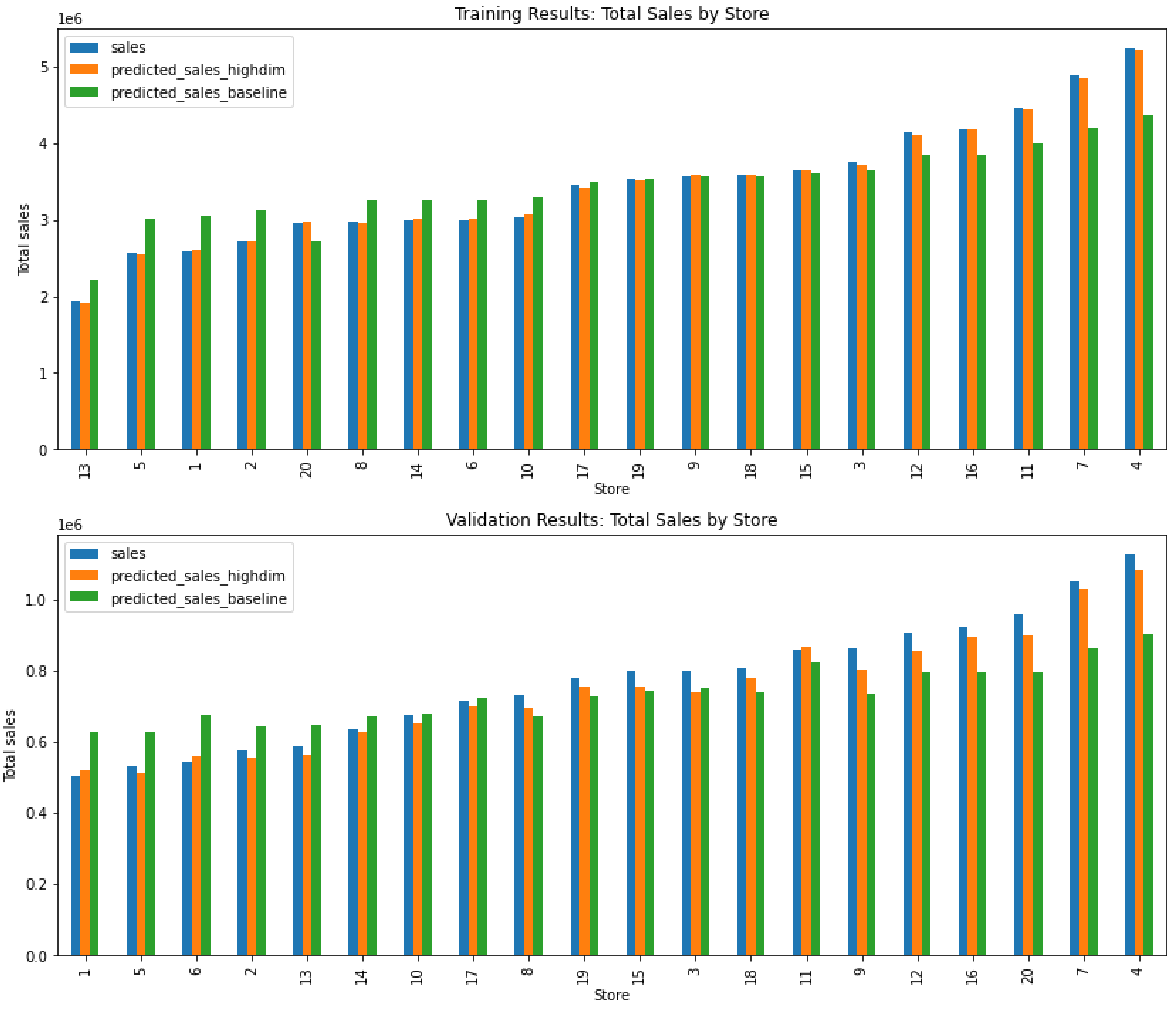
We can also look at aggregate sales for a subset of stores. We select the first 20 stores and plot in order of increasing sales.

[14]:
```
_, axs = plt.subplots(2, 1, figsize=(14, 12))
for i, select in enumerate([select_train, select_val]):
    ax = axs[i]
    df_plot_store = df[select].groupby(
        ["store"]
    ).agg("sum")[sales_cols].reset_index()[:20].sort_values(by="sales")

    df_plot_store.plot.bar(x="store", ax=ax)
    ax.set_xlabel("Store")
```

(continues on next page)

(continued from previous page)

```
    ax.set_ylabel("Total sales")

axs[0].set_title("Training Results: Total Sales by Store")
axs[1].set_title("Validation Results: Total Sales by Store")
plt.show()
```



We can see that the high dimensional model is much better at capturing the variation between months and individual stores!

5.3 Tikhonov Regularization Tutorial: Seattle-Tacoma Housing Data

Intro

This tutorial shows how to use variable L_2 regularization with glum. The `P2` parameter of the `GeneralizedLinearRegressor` class allows you to directly set the L_2 penalty matrix $w^T P_2 w$. If a 2d array is passed for the `P2` parameter, it is used directly, while if you pass a 1d array as `P2` it will be interpreted as the diagonal of P_2 and all other entries will be assumed to be zero.

Note: Variable L_1 regularization is also available by passing an array with length `n_features` to the `P1` parameter.

Background

For this tutorial, we will model the selling price of homes in King's County, Washington (Seattle-Tacoma Metro area) between May 2014 and May 2015. However, in order to demonstrate a Tikhonov regularization-based spatial smoothing technique, we will focus on a small, skewed data sample from that region in our training data. Specifically, we will show that when we have (a) a fixed effect for each postal code region and (b) only a select number of training observations in a certain region, we can improve the predictive power of our model by regularizing the difference between the coefficients of neighboring regions. While we are constructing a somewhat artificial example here in order to demonstrate the spatial smoothing technique, we have found similar techniques to be applicable to real-world problems.

We will use a gamma distribution for our model. This choice is motivated by two main factors. First, our target variable, home price, is a positive real number, which matches the support of the gamma distribution. Second, it is expected that factors influencing housing prices are multiplicative rather than additive, which is better captured with a gamma regression than say, OLS.

Note: a few parts of this tutorial utilize local helper functions outside this notebook. If you wish to run the notebook on your own, you can find the rest of the code here.

5.3.1 Table of Contents

- 1. Load and Prepare Datasets from Openml.org
- 2. Visualize Geographic Data with GIS Open Data
- *3. Feature Selection and Transformation*
- 4. Create P matrix
- *5. Fit Models*

[1]:

```
import itertools

import geopandas as geopd
import libpysal
import matplotlib.pyplot as plt
import numpy as np
import openml
import pandas as pd

from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from glum import GeneralizedLinearRegressor

import sys
sys.path.append("../")
```

(continues on next page)

(continued from previous page)

```
from metrics import root_mean_squared_percentage_error

import warnings
warnings.filterwarnings("ignore", message="The weights matrix is not fully connected")

import data_prep
import maps
```

5.3.2 1. Load and prepare datasets from Openml

back to table of contents

1.1. Download and transform

The main dataset is downloaded from OpenML. You can find the main page for the dataset here. It is also available through Kaggle here.

As part of data preparation, we also do some transformations to the data:

- We remove some outliers (homes over 1.5 million and under 100k).
- Since we want to focus on geographic features, we also remove a handful of the other features.

Below, you can see some example rows from the dataset.

[2]:

```
df = data_prep.download_and_transform()
df.head()
```

[2]:

```
   bedrooms  bathrooms  sqft_living  floors  waterfront  view  condition  \
0         3       1.00         1180     1.0           0     0          3
1         3       2.25         2570     2.0           0     0          3
2         2       1.00          770     1.0           0     0          3
3         4       3.00         1960     1.0           0     0          5
4         3       2.00         1680     1.0           0     0          3

   sqft_basement  yr_built  zipcode     price
0              0      1955    98178  221900.0
1            400      1951    98125  538000.0
2              0      1933    98028  180000.0
3            910      1965    98136  604000.0
4              0      1987    98074  510000.0
```

5.3.3 2. Visualize geographic data with GIS open

back to table of contents

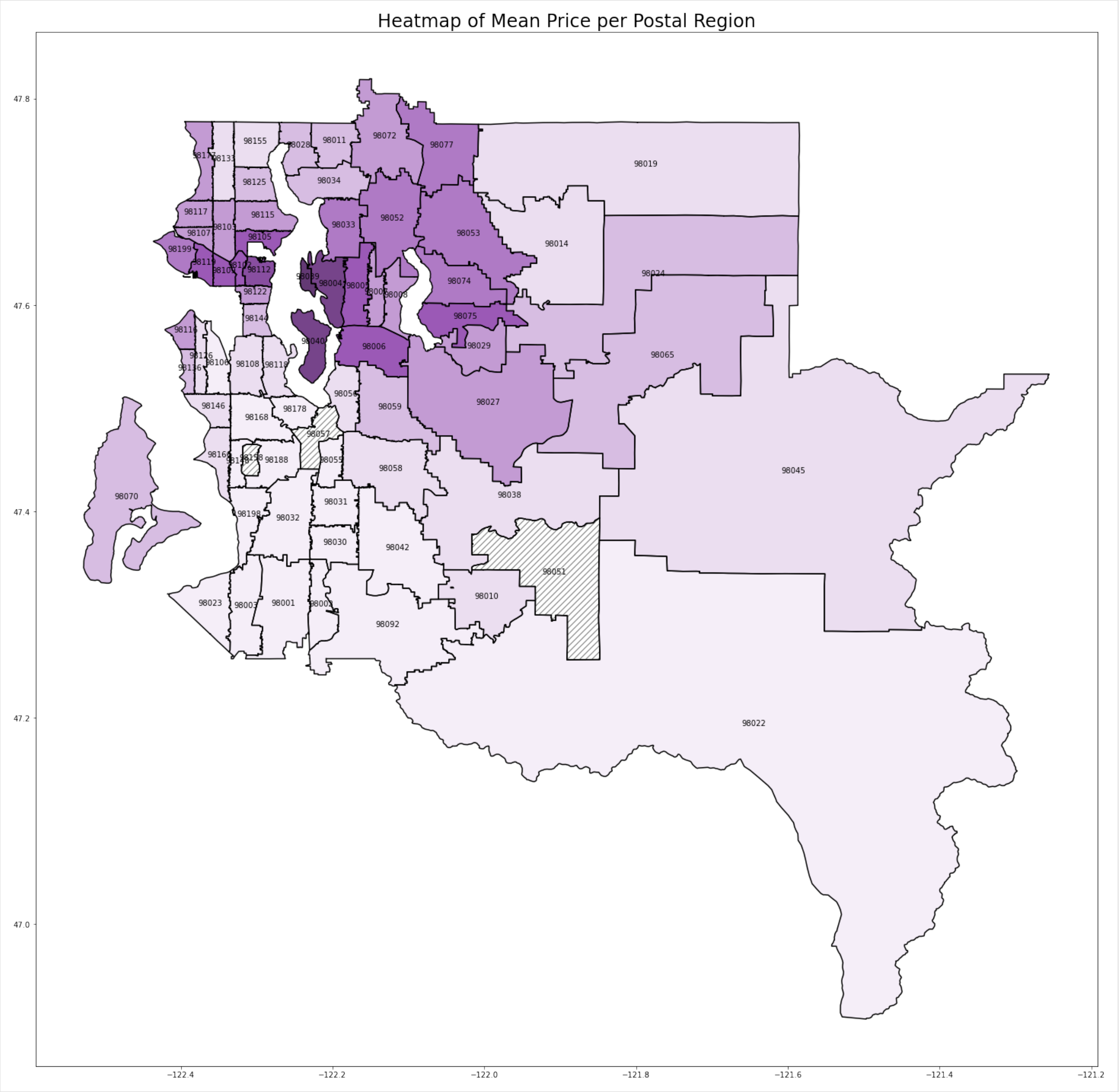
To help visualize the geographic data, we use geopandas and GIS Open Data to display price information on the King's county map. You can get the map data *here*.

To show the relatioship between home price and geography, we merge the map data with our sales data and use a heat map to plot mean home sale price for each postal code region.

```
[3]: maps.read_shapefile("Zip_Codes/Zip_Codes.shp")
```

```
[3]:      OBJECTID    ZIP ZIPCODE COUNTY     SHAPE_Leng    SHAPE_Area  \
     0           1  98031   98031    033  117508.211718  2.280129e+08
     1           2  98032   98032    033  166737.664791  4.826754e+08
     2           3  98033   98033    033  101363.840369  2.566747e+08
     3           4  98034   98034    033   98550.452509  2.725072e+08
     4           5  98030   98030    033   94351.264837  2.000954e+08
     ..        ...    ...     ...    ...            ...           ...
     199       200  98402   98402    053   30734.178112  2.612224e+07
     200       201  98403   98403    053   23495.038425  2.890938e+07
     201       202  98404   98404    053   61572.154365  2.160645e+08
     202       203  98405   98405    053   50261.100559  1.193118e+08
     203       204  98406   98406    053   74118.972418  1.088373e+08

                                                   geometry
     0    POLYGON ((-122.2184228967409 47.4375036485968,...
     1    (POLYGON ((-122.2418694980486 47.4412158004961...
     2    POLYGON ((-122.2057111926017 47.65169738162997...
     3    POLYGON ((-122.1755100327681 47.73706057280546...
     4    POLYGON ((-122.1674637459728 47.38548925033355...
     ..                                                 ...
     199  POLYGON ((-122.4427945843513 47.2647926142345,...
     200  POLYGON ((-122.4438167281511 47.26617469660845...
     201  POLYGON ((-122.3889999141967 47.23495303304902...
     202  POLYGON ((-122.4409198889526 47.23639133730699...
     203  (POLYGON ((-122.5212509005256 47.2712095490982...

     [204 rows x 7 columns]
```

```
[4]: df_shapefile = maps.read_shapefile("Zip_Codes/Zip_Codes.shp")
     df_map = maps.create_kings_county_map_df(df, df_shapefile)

     fix, ax = plt.subplots(figsize=(25, 25))
     maps.plot_heatmap(df=df_map, label_col="ZIP", data_col="price", ax=ax)
     ax.set_title("Heatmap of Mean Price per Postal Region", fontsize=24)
     plt.show()
```



We can see a clear relationship between postal code and home price. Seattle (98112, 98102, etc.) and the Bellevue/Mercer/Medina suburbs (98039, 98004, 98040) have the highest prices. As you get further from the city, the prices start to drop.

5.3.4 3. Feature selection and transformation

back to table of contents

3.1 Feature selection and one hot encoding

Since we want to focus on geographic data, we drop a number of columns below. We keep a handful of columns so that we can still create a reasonable model.

We then create a fixed effect for each of the postal code regions. We add the encoded postcode columns in numeric order to help us maintain the proper order of columns while building and training the model.

```
[5]: sorted_zips = sorted(list(df["zipcode"].unique()))
     one_hot = pd.get_dummies(df["zipcode"], dtype=float)
     one_hot = one_hot[sorted_zips]
     df = df.drop('zipcode', axis=1)
     df = one_hot.join(df)
     df.head()
```

```
[5]:    98001  98002  98003  98004  98005  98006  98007  98008  98010  98011  ...  \
     0      0      0      0      0      0      0      0      0      0      0  ...
     1      0      0      0      0      0      0      0      0      0      0  ...
     2      0      0      0      0      0      0      0      0      0      0  ...
     3      0      0      0      0      0      0      0      0      0      0  ...
     4      0      0      0      0      0      0      0      0      0      0  ...

        bedrooms  bathrooms  sqft_living  floors  waterfront  view  condition  \
     0         3       1.00         1180     1.0           0     0          3
     1         3       2.25         2570     2.0           0     0          3
     2         2       1.00          770     1.0           0     0          3
     3         4       3.00         1960     1.0           0     0          5
     4         3       2.00         1680     1.0           0     0          3

        sqft_basement  yr_built     price
     0              0      1955  221900.0
     1            400      1951  538000.0
     2              0      1933  180000.0
     3            910      1965  604000.0
     4              0      1987  510000.0

     [5 rows x 80 columns]
```

3.2 Test train split

As we mentioned in the introduction, we want to focus on modeling the selling price in a specific region while only using a very small, skewed data sample from that region in our training data. This scenario could arise if say, our task was to predict the sales prices for homes in Enumclaw (large region with zip code 98022 in the southeast corner of the map), but the only data we had from there was from a small luxury realtor.

To mimic this, instead creating a random split between our training and test data, we will intentionally create a highly skewed sample. For our test set, we will take all of the home sales in Enumclaw, *except* for the 15 highest priced homes.

Finally, we standardize our predictors.

[6]:
```
predictors = [c for c in df.columns if c != "price"]

test_region = "98022"
df_train = df[df[test_region] == 0]
df_test = df[df[test_region] == 1].sort_values(by="price", ascending=False)

test_to_train = df_test[:15]

df_train = pd.concat([df_train, test_to_train])
df_test = df_test.drop(test_to_train.index)

X_train = df_train[predictors]
y_train = df_train["price"]
X_test = df_test[predictors]
y_test = df_test["price"]

scaler = StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

5.3.5 4. Creating the penalty matrix

back to table of contents

To smooth the coefficients for neighboring regions, we will create a penalty matrix P such that we penalize the squared difference in coefficient values for neighbouring regions, e.g. for 98022 and 98045. For example, if 98022 and 98045 were the only region in question, we would need a 2×2 matrix P such that:

$$\left(\beta_{98022}, \beta_{98045}\right) P \begin{pmatrix} \beta_{98022} \\ \beta_{98045} \end{pmatrix} = (\beta_{98022} - \beta_{98045})^2$$

In this example, we would get this result with $P = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$.

Since we have 72 postal code regions, it would be rather annoying to construct this matrix by hand. Luckily, there are libraries that exist for this. We use pysal's pysal.lib.weights.Queen to retrieve a neighbor's matrix from our map data. The construction of the penalty matrix is rather straightforward once we have this information.

We leave the non-geographic features unregularized (all zeros in the P matrix).

[7]:
```
# format is {zip1: {neighbord1: 1, neighbor2: 1, ...}}
neighbor_matrix = libpysal.weights.Queen.from_dataframe(df_map, ids="ZIP")

n_features = X_train.shape[1]
P2 = np.zeros((n_features, n_features))

zip2index = dict(zip(sorted_zips, range(len(sorted_zips))))
for zip1 in sorted_zips:
    for zip2 in neighbor_matrix[zip1].keys():
        if zip1 in zip2index and zip2 in zip2index: # ignore regions w/o data
            if zip2index[zip1] < zip2index[zip2]: # don't repeat if already saw neighbor␣
↪pair in earlier iteration
                P2[zip2index[zip1], zip2index[zip1]] += 1
```

(continues on next page)

(continued from previous page)

```
                P2[zip2index[zip2], zip2index[zip2]] += 1
                P2[zip2index[zip1], zip2index[zip2]] -= 1
                P2[zip2index[zip2], zip2index[zip1]] -= 1
P2
```

[7]:
```
array([[ 3., -1., -1., ...,  0.,  0.,  0.],
       [-1.,  4.,  0., ...,  0.,  0.,  0.],
       [-1.,  0.,  4., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

5.3.6 5. Fit models

back to table of contents

Now, we will fit several L2 regularized OLS models using different levels of regularization. All will use the penalty matrix defined above, but the alpha parameter, the constant that multiplies the penalty terms and thus determines the regularization strength, will vary.

For each model, we will measure test performance using root mean squared percentage error (RMSPE), so that we can get a relaitve result. We will also plot a heatmat of the coefficient values over the regions.

Note: alpha=1e-12 is effectively no regularization. But we can't set alpha to zero because the unregularized problem has co-linear columns, resulting in a singular design matrix.

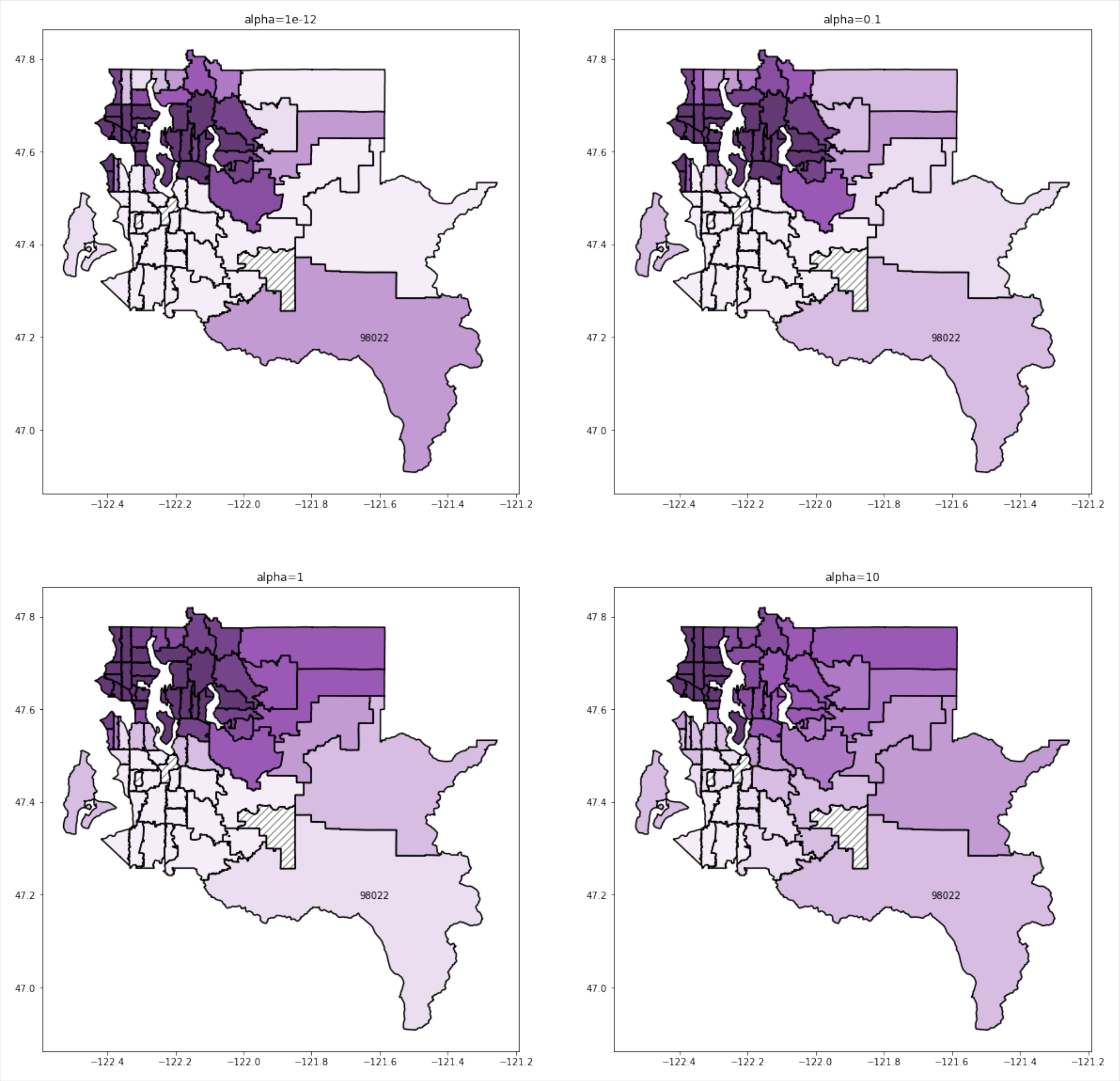
[8]:
```
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(20, 20))
for i, alpha in enumerate([1e-12, 1e-1, 1, 10]):

    glm = GeneralizedLinearRegressor(family='gamma', alpha=alpha, P2=P2, fit_
↪intercept=True)
    glm.fit(X_train, y_train)
    y_test_hat = glm.predict(X_test)

    coeffs = pd.DataFrame({'coefficient': np.concatenate(([glm.intercept_], glm.coef_))},
↪ ["intercept"]+predictors)

    print(f"alpha={alpha}")
    print(f"Test region coefficient: {coeffs.loc[test_region].values[0]}")
    print(f"Test RMSPE: {root_mean_squared_percentage_error(y_test_hat, y_test)}\n")

    df_map_coeffs = df_map.merge(
        coeffs.loc[sorted_zips],
        left_on="ZIP",
        right_index=True,
        how="outer"
    )

    ax = axs[i//2, i%2]
    df_map_coeffs["annotation"] = df_map_coeffs["ZIP"].apply(lambda x: "" if x!=test_
↪region else x)
    maps.plot_heatmap(
```

(continues on next page)

(continued from previous page)

```
        df=df_map_coeffs,
        label_col="annotation",
        data_col="coefficient",
        ax=ax,
        vmin=-0.015,
        vmax=0.025
    )
    ax.set_title(f"alpha={alpha}")

plt.show()
```

```
alpha=1e-12
Test region coefficient: 0.0010920106922960072
Test RMSPE: 72.65620542354644

alpha=0.1
Test region coefficient: -0.0036087215513505183
Test RMSPE: 43.926082004444204

alpha=1
Test region coefficient: -0.010413920757076630
Test RMSPE: 19.51113178158937

alpha=10
Test region coefficient: -0.0033476740903954213
Test RMSPE: 44.59786775358339
```



alpha=1 seems to recover the best results. Remember that our test dataset is just a small subset of the data in region 98022 and that the training data is skewed towards high sales prices. For alpha less than 1, we can see that the 98022 region coefficient is still much greater than its neighbors coefficients, which we can see is not accurate if we refer back to map we produced based on the raw data. For higher alpha levels, we start to see poor predictions resulting from regional coefficients that are too smooth between adjacent regions.

A test RMSPE of 19.5% is a surprisingly good result considering that we only had 10 highly skewed observations from our test region in our training data and is far better than the RMSPE of 67.5% from the unregularized case.

5.4 Formula Interface Tutorial: Revisiting French Motor Third-Party Liability Claims

Intro

This tutorial showcases the formula interface of `glum`. It allows for the specification of the design matrix and the response variable using so-called Wilkinson-formulas instead of constructing it by hand. This kind of model specification should be familiar to R users or those who have used the `statsmodels` or `linearmodels` Python packages before. This tutorial aims to introduce the basics of working with formulas to other users, as well as highlighting some important differences between `glums` and other packages' formula implementations.

For a more in-depth look at how formulas work, please take a look at the *documentation of ``formulaic`* <https://matthewwardrop.github.io/formulaic/>`__, the package on which `glum`'s formula interface is based.

Background

This tutorial reimplements and extends the combined frequency-severity model from Chapter 4 of the GLM tutorial. If you would like to know more about the setting, the data, or GLM modeling in general, please check that out first.

Sneak Peak

Formulas can provide a concise and convenient way to specify many of the usual pre-processing steps, such as converting to categorical types, creating interactions, applying transformations, or even spline interpolation. As an example, consider the following formula:

```
{ClaimAmountCut / Exposure} ~ C(DrivAge, missing_method='convert') * C(VehPower, missing_
↪method="zero") + bs(BonusMalus, 3)
```

Despite its brevity, it describes all of the following: - The outcome variable is the ratio of `ClaimAmountCut` and `Exposure`. - The predictors should include the interactions of the categorical variables `DrivAge` and `VehPower`, as well as those two variables themselves. (Even though they behave as such, neither the individual variables nor their interaction will be dummy-encoded by glum. For categoricals with many levels, this can lead to a substantial performance improvement over dummy encoding, especially for the interaction.) - If there are missing values in `DrivAge`, they should be treated as a separate category. - On the other hand, missing values in `VehPower` should be treated as all-zero indicators. - The predictors should also include a third degree B-spline interpolation of `BonusMalus`.

The following chapters demonstrate each of these features in some detail, as well as some additional advantages of using the formula interface.

5.4.1 Table of Contents

- *1. Load and Prepare Datasets from Openml*
- 2. Reproducing the model from the GLM Tutorial
- *3. Categorical Variables*
- *4. Interactions and Structural Full-rankness*
- *5. Fun with Functions*
- *6. Miscellaneous Features*

[1]:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pytest
```

(continues on next page)

(continued from previous page)

```
import scipy.optimize as optimize
import scipy.stats
from dask_ml.preprocessing import Categorizer
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import ShuffleSplit
from glum import GeneralizedLinearRegressor
from glum import TweedieDistribution

from load_transform_formula import load_transform
```

5.4.2 1. Load and Prepare Datasets from Openml

back to table of contents

First, we load in our *dataset from openML* and apply several transformations. In the interest of simplicity, we do not include the data loading and preparation code in this notebook.

[2]:
```
df = load_transform()
with pd.option_context('display.max_rows', 10):
    display(df)
```

```
         ClaimNb  Exposure Area  VehPower  VehAge  DrivAge  BonusMalus  \
IDpol
1              0   0.10000    D         5       0        5          50
3              0   0.77000    D         5       0        5          50
5              0   0.75000    B         6       1        5          50
10             0   0.09000    B         7       0        4          50
11             0   0.84000    B         7       0        4          50
...          ...       ...  ...       ...     ...      ...         ...
6114326        0   0.00274    E         4       0        5          50
6114327        0   0.00274    E         4       0        4          95
6114328        0   0.00274    D         6       1        4          50
6114329        0   0.00274    B         4       0        5          50
6114330        0   0.00274    B         7       1        2          54

        VehBrand   VehGas  Density Region  ClaimAmount  ClaimAmountCut
IDpol
1            B12  Regular     1217    R82          0.0             0.0
3            B12  Regular     1217    R82          0.0             0.0
5            B12   Diesel       54    R22          0.0             0.0
10           B12   Diesel       76    R72          0.0             0.0
11           B12   Diesel       76    R72          0.0             0.0
...          ...      ...      ...    ...          ...             ...
6114326      B12  Regular     3317    R93          0.0             0.0
6114327      B12  Regular     9850    R11          0.0             0.0
6114328      B12   Diesel     1323    R82          0.0             0.0
6114329      B12  Regular       95    R26          0.0             0.0
6114330      B12   Diesel       65    R72          0.0             0.0

[678013 rows x 13 columns]
```

5.4.3 2. Reproducing the Model From the GLM Turorial

Now, let us start by fitting a very simple model. As usual, let's divide our samples into a training and a test set so that we get valid out-of-sample goodness-of-fit measures. Perhaps less usually, we do not create separate `y` and `X` data frames for our label and features – the formula will take care of that for us.

We still have some preprocessing to do: - Many of the ordinal or nominal variables are encoded as integers, instead of as categoricals. We will need to convert these so that `glum` will know to estimate a separate coefficient for each of their levels. - The outcome variable is a transformation of other columns. We need to create it first.

As we will see later on, these steps can be incorporated into the formula itself, but let's not overcomplicate things at first.

[3]:

```
ss = ShuffleSplit(n_splits=1, test_size=0.1, random_state=42)
train, test = next(ss.split(df))

df = df.assign(PurePremium=lambda x: x["ClaimAmountCut"] / x["Exposure"])

glm_categorizer = Categorizer(
    columns=["VehBrand", "VehGas", "Region", "Area", "DrivAge", "VehAge", "VehPower"]
)
df_train = glm_categorizer.fit_transform(df.iloc[train])
df_test = glm_categorizer.transform(df.iloc[test])
```

This example demonstrates the basic idea behind formulas: the outcome variable and the predictors are separated by a tilde (~), and different prefictors are separated by plus signs (+). Thus, formulas provide a concise way of specifying a model without the need to create dataframes by hand.

[4]:

```
formula = "PurePremium ~ VehBrand + VehGas + Region + Area + DrivAge + VehAge + VehPower␣
↪+ BonusMalus + Density"

TweedieDist = TweedieDistribution(1.5)
t_glm1 = GeneralizedLinearRegressor(
    family=TweedieDist,
    alpha_search=True,
    l1_ratio=1,
    fit_intercept=True,
    formula=formula,
)
t_glm1.fit(df_train, sample_weight=df["Exposure"].values[train])

pd.DataFrame(
    {"coefficient": np.concatenate(([t_glm1.intercept_], t_glm1.coef_))},
    index=["intercept"] + t_glm1.feature_names_,
).T
```

[4]:

```
             intercept  VehBrand[B1]  VehBrand[B10]  VehBrand[B11]  \
coefficient    2.88667     -0.064157            0.0       0.231868

             VehBrand[B12]  VehBrand[B13]  VehBrand[B14]  VehBrand[B2]  \
coefficient      -0.211061       0.054979      -0.270346     -0.071453

             VehBrand[B3]  VehBrand[B4]  ...  VehAge[1]  VehAge[2]  \
coefficient       0.00291      0.059324  ...   0.008117  -0.229906
```

(continues on next page)

(continued from previous page)

```
             VehPower[4]  VehPower[5]  VehPower[6]  VehPower[7]  VehPower[8]  \
coefficient    -0.111796    -0.123388     0.060757     0.005179    -0.021832

             VehPower[9]  BonusMalus   Density
coefficient     0.208158    0.032508  0.000002

[1 rows x 60 columns]
```

5.4.4 3. Categorical Variables

glum also provides extensive support for categorical variables. The main function one needs to be aware of in the context of categoricals is simply called C(). A variable placed within it is always converted to a categorical, regardless of its type.

A huge part of tabmat's/glum's performance advantage is that categoricals need not be one-hot encoded, but are treated as if they were. For this reason, we do not support using other coding schemes within the formula interface. If one needs to use other categorical encodings than one-hot, they can always do so manually (or even using formulaic directly) before the estimation.

Let's try it out on our dataset!

```
[5]: df_train_noncat = df.iloc[train]
     df_test_noncat = df.iloc[test]

     df_train_noncat.dtypes
```

```
[5]: ClaimNb             int64
     Exposure          float64
     Area               object
     VehPower            int64
     VehAge              int64
     DrivAge             int64
     BonusMalus          int64
     VehBrand           object
     VehGas             object
     Density             int64
     Region             object
     ClaimAmount       float64
     ClaimAmountCut    float64
     PurePremium       float64
     dtype: object
```

Even though some of the variables are integers in this dataset, they are handled as categoricals thanks to the C() function. Strings, such as VehBrand or VehGas would have been handled as categorical by default anyway, but using the C() function never hurts: if applied to something that is already a caetgorical variable, it does not have any effect outside of the feature name.

```
[6]: formula_cat = (
         "PurePremium ~ C(VehBrand) + C(VehGas) + C(Region) + C(Area) "
         "+ C(DrivAge) + C(VehAge) + C(VehPower) + BonusMalus + Density"
     )
```

(continues on next page)

(continued from previous page)

```
t_glm3 = GeneralizedLinearRegressor(
    family=TweedieDist,
    alpha_search=True,
    l1_ratio=1,
    fit_intercept=True,
    formula=formula_cat,
)
t_glm3.fit(df_train_noncat, sample_weight=df["Exposure"].values[train])

pd.DataFrame(
    {"coefficient": np.concatenate(([t_glm3.intercept_], t_glm3.coef_))},
    index=["intercept"] + t_glm3.feature_names_,
).T
```

[6]:

```
             intercept  C(VehBrand)[B1]  C(VehBrand)[B10]  C(VehBrand)[B11]  \
coefficient    2.88667        -0.064157               0.0          0.231868

             C(VehBrand)[B12]  C(VehBrand)[B13]  C(VehBrand)[B14]  \
coefficient         -0.211061          0.054979         -0.270346

             C(VehBrand)[B2]  C(VehBrand)[B3]  C(VehBrand)[B4]  ...  \
coefficient        -0.071453          0.00291         0.059324  ...

             C(VehAge)[1]  C(VehAge)[2]  C(VehPower)[4]  C(VehPower)[5]  \
coefficient      0.008117     -0.229906       -0.111796       -0.123388

             C(VehPower)[6]  C(VehPower)[7]  C(VehPower)[8]  C(VehPower)[9]  \
coefficient        0.060757        0.005179       -0.021832        0.208158

             BonusMalus   Density
coefficient    0.032508  0.000002

[1 rows x 60 columns]
```

Finally, prediction works as expected with categorical variables. `glum` keeps track of the levels present in the training dataset, and makes sure that categorical variables in unseen datasets are also properly aligned, even if they have missing or unknown levels.3 Therefore, one can simply use predict, and `glum` does The Right Thing™ by default.

3: This is made possible due to `glum` saving a `` `ModelSpec `` object <https://matthewwardrop.github.io/formulaic/guides/model_specs/>`__, which contains any information necessary for reapplying the transitions that were done during the formula materialization process. It is especially relevant in the case of stateful transforms, such as creating categorical variables.

[7]:

```
t_glm3.predict(df_test_noncat)
```

[7]:

```
array([303.77443311, 548.47789523, 244.34438579, ..., 109.81572865,
        67.98332028, 297.21717383])
```

5.4.5 4. Interactions and Structural Full-Rankness

One of the biggest strengths of Wilkinson-formuals lie in their ability of concisely specifying interactions between terms. glum implements this as well, and in a very efficient way: the interactions of categorical features are encoded as a new categorical feature, making it possible to interact high-cardinality categoricals with each other. If this is not possible, because, for example, a categorical is interacted with a numeric variable, sparse representations are used when appropriate. In general, just as with glum's categorical handling in general, you can be assured that glum you don't have to worry too much about the actual implementation, and can expect that glum will do the most efficient thing behind the scenes.

Let's see how that looks like on the insurance example! Suppose that we expect VehPower to have a different effect depending on DrivAge (e.g. performance cars might not be great for new drivers, but may be less problematic for more experienced ones). We can include the interaction of these variables as follows.

[8]:
```
formula_int = (
    "PurePremium ~ C(VehBrand) + C(VehGas) + C(Region) + C(Area)"
    " + C(DrivAge) * C(VehPower) + C(VehAge) + BonusMalus + Density"
)

t_glm4 = GeneralizedLinearRegressor(
    family=TweedieDist,
    alpha_search=True,
    l1_ratio=1,
    fit_intercept=True,
    formula=formula_int,
)
t_glm4.fit(df_train, sample_weight=df["Exposure"].values[train])

pd.DataFrame(
    {"coefficient": np.concatenate(([t_glm4.intercept_], t_glm4.coef_))},
    index=["intercept"] + t_glm4.feature_names_,
).T
```

[8]:
```
             intercept  C(VehBrand)[B1]  C(VehBrand)[B10]  C(VehBrand)[B11]  \
coefficient    2.88023        -0.069076               0.0          0.221037

             C(VehBrand)[B12]  C(VehBrand)[B13]  C(VehBrand)[B14]  \
coefficient         -0.211854          0.052355         -0.272058

             C(VehBrand)[B2]  C(VehBrand)[B3]  C(VehBrand)[B4]  ...  \
coefficient        -0.074836              0.0         0.052523  ...

             C(DrivAge)[4]:C(VehPower)[8]  C(DrivAge)[5]:C(VehPower)[8]  \
coefficient                     -0.147844                      -0.03567

             C(DrivAge)[6]:C(VehPower)[8]  C(DrivAge)[0]:C(VehPower)[9]  \
coefficient                      0.504407                      0.682528

             C(DrivAge)[1]:C(VehPower)[9]  C(DrivAge)[2]:C(VehPower)[9]  \
coefficient                     -0.106569                     -0.308257

             C(DrivAge)[3]:C(VehPower)[9]  C(DrivAge)[4]:C(VehPower)[9]  \
coefficient                      0.173206                      0.010684
```

(continues on next page)

(continued from previous page)

```
             C(DrivAge)[5]:C(VehPower)[9]  C(DrivAge)[6]:C(VehPower)[9]
coefficient                     -0.220273                      0.070334

[1 rows x 102 columns]
```

Note that, in addition to the interactions, the non-interacted variants of `DrivAge` and `VehPower` are also included in the model. This is a result of using the `*` operator to interact the variables. Using `:` instead would only include the interactions, and not the marginals. (In short, `a * b` is equivalent to `a + b + a:b`.)

[9]:

```
[name for name in t_glm4.feature_names_ if "VehPower" in name]
```

[9]:

```
['C(VehPower)[4]',
 'C(VehPower)[5]',
 'C(VehPower)[6]',
 'C(VehPower)[7]',
 'C(VehPower)[8]',
 'C(VehPower)[9]',
 'C(DrivAge)[0]:C(VehPower)[4]',
 'C(DrivAge)[1]:C(VehPower)[4]',
 'C(DrivAge)[2]:C(VehPower)[4]',
 'C(DrivAge)[3]:C(VehPower)[4]',
 'C(DrivAge)[4]:C(VehPower)[4]',
 'C(DrivAge)[5]:C(VehPower)[4]',
 'C(DrivAge)[6]:C(VehPower)[4]',
 'C(DrivAge)[0]:C(VehPower)[5]',
 'C(DrivAge)[1]:C(VehPower)[5]',
 'C(DrivAge)[2]:C(VehPower)[5]',
 'C(DrivAge)[3]:C(VehPower)[5]',
 'C(DrivAge)[4]:C(VehPower)[5]',
 'C(DrivAge)[5]:C(VehPower)[5]',
 'C(DrivAge)[6]:C(VehPower)[5]',
 'C(DrivAge)[0]:C(VehPower)[6]',
 'C(DrivAge)[1]:C(VehPower)[6]',
 'C(DrivAge)[2]:C(VehPower)[6]',
 'C(DrivAge)[3]:C(VehPower)[6]',
 'C(DrivAge)[4]:C(VehPower)[6]',
 'C(DrivAge)[5]:C(VehPower)[6]',
 'C(DrivAge)[6]:C(VehPower)[6]',
 'C(DrivAge)[0]:C(VehPower)[7]',
 'C(DrivAge)[1]:C(VehPower)[7]',
 'C(DrivAge)[2]:C(VehPower)[7]',
 'C(DrivAge)[3]:C(VehPower)[7]',
 'C(DrivAge)[4]:C(VehPower)[7]',
 'C(DrivAge)[5]:C(VehPower)[7]',
 'C(DrivAge)[6]:C(VehPower)[7]',
 'C(DrivAge)[0]:C(VehPower)[8]',
 'C(DrivAge)[1]:C(VehPower)[8]',
 'C(DrivAge)[2]:C(VehPower)[8]',
 'C(DrivAge)[3]:C(VehPower)[8]',
 'C(DrivAge)[4]:C(VehPower)[8]',
 'C(DrivAge)[5]:C(VehPower)[8]',
 'C(DrivAge)[6]:C(VehPower)[8]',
 'C(DrivAge)[0]:C(VehPower)[9]',
```

(continues on next page)

(continued from previous page)

```
 'C(DrivAge)[1]:C(VehPower)[9]',
 'C(DrivAge)[2]:C(VehPower)[9]',
 'C(DrivAge)[3]:C(VehPower)[9]',
 'C(DrivAge)[4]:C(VehPower)[9]',
 'C(DrivAge)[5]:C(VehPower)[9]',
 'C(DrivAge)[6]:C(VehPower)[9]']
```

The attentive reader might have also noticed that the first level of each categorical variable is omitted from the model. This is a manifestation of the more general concept of ensuring structural full-rankedness4. By default, `glum` and `formulaic` will try to make sure that one does not fall into the Dummy Variable Trap. Moreover, it even does it in the case of (possibly multi-way) interactions involving categorical variables. It will always drop the necessary number of levels, and no more. If you want to opt out of this behavior (for example because you would like to penalize all levels equally), simply set the `drop_first` parameter during model initialization to `False`. If one only aims to include all levels of a certain variable, and not others, it is possible to do so by using the `spans_intercept` parameter (e.g. `C(VehPower, spans_intercept=False)` would include all levels of `VehPower` even if `drop_first` is set to `True`).

4: Note, that it does not guarantee that the design matrix is actually full rank. For example, two identical numerical variables will still lead to a rank-deficient design matrix.

5.4.6 5. Fun with Functions

The previous example is only scratching the surface of what formulas are capable of. For example, they are capable of evaluating arbitrary Python expressions, which act as if they saw the columns of the input data frame as local variables (`pandas.Series`). The way to tell `glum` that a part of the formula should be evaluated as a Python expression before applying the formula grammar to it is to enclose it in curly braces. As an example, we can easily do the following within the formula itself:

1. Create the outcome variable on the fly instead of doing it beforehand.
2. Include the logarithm of a certain variable in the model.
3. Include a basis spline interpolation of a variable to capture non-linearities in its effect.

1. works because because formulas can contain Python operations. 2. and 3. work because formulas are evaluated within a context that is aware of a number of transforms. To be precise, 2. is a regular transform and 3. is a stateful transform.

Let's try it out!

[10]:

```
formula_fun = (
    "{ClaimAmountCut / Exposure} ~ VehBrand + VehGas + Region + Area"
    " + DrivAge + VehAge + VehPower + bs(BonusMalus, 3) + np.log(Density)"
)

t_glm5 = GeneralizedLinearRegressor(
    family=TweedieDist,
    alpha_search=True,
    l1_ratio=1,
    fit_intercept=True,
    formula=formula_fun,
)
t_glm5.fit(df_train, sample_weight=df["Exposure"].values[train])

pd.DataFrame(
```

(continues on next page)

(continued from previous page)

```
    {"coefficient": np.concatenate(([t_glm5.intercept_], t_glm5.coef_))},
    index=["intercept"] + t_glm5.feature_names_,
).T
```

[10]:

```
             intercept  VehBrand[B1]  VehBrand[B10]  VehBrand[B11]  \
coefficient   3.808829     -0.060201            0.0       0.242194

             VehBrand[B12]  VehBrand[B13]  VehBrand[B14]  VehBrand[B2]  \
coefficient      -0.202517       0.063471      -0.345415     -0.072546

             VehBrand[B3]  VehBrand[B4]  ...  VehPower[4]  VehPower[5]  \
coefficient       0.00777      0.079391  ...    -0.113038    -0.127255

             VehPower[6]  VehPower[7]  VehPower[8]  VehPower[9]  \
coefficient     0.060209     0.005577    -0.032114     0.207355

             bs(BonusMalus, 3)[1]  bs(BonusMalus, 3)[2]  bs(BonusMalus, 3)[3]  \
coefficient              3.178178              0.361951              8.231846

             np.log(Density)
coefficient         0.121944

[1 rows x 62 columns]
```

To allow for even more flexibility, you can add custom transformations that are defined in the context from which the call is made. E.g., we can define a transformation that takes the logarithm of `VehAge + 1` after casting it to numeric. To make the formula recognize this transform, you need to explicitly set `context=0` when calling the fit method (note that this differs from `formulaic`'s default, which is already `context=0`).

[11]:

```
def _log_plus_one(x):
    return np.log(pd.to_numeric(x) + 1)

formula_custom_fun = (
    "{ClaimAmountCut / Exposure} ~ _log_plus_one(VehAge)"
)

t_glm6 = GeneralizedLinearRegressor(
    family=TweedieDist,
    alpha_search=True,
    l1_ratio=1,
    fit_intercept=True,
    formula=formula_custom_fun,
)
t_glm6.fit(df_train, sample_weight=df["Exposure"].values[train], context=0)

pd.DataFrame(
    {"coefficient": np.concatenate(([t_glm6.intercept_], t_glm6.coef_))},
    index=["intercept"] + t_glm6.feature_names_,
).T
```

[11]:

```
             intercept  _log_plus_one(VehAge)
coefficient   5.046712              -0.151043
```

5.4.7 6. Miscellaneous Features

Variable Names

glum's formula interface provides a lot of control over how the resulting features are named. By default, it follows formulaic's standards, but it can be customized by setting the interaction_separator and categorical_format paremeters.

```
[12]: formula_name = "PurePremium ~ DrivAge * VehPower"

t_glm7 = GeneralizedLinearRegressor(
    family=TweedieDist,
    alpha_search=True,
    l1_ratio=1,
    fit_intercept=True,
    formula=formula_name,
    interaction_separator="__x__",
    categorical_format="{name}__{category}",
)
t_glm7.fit(df_train, sample_weight=df["Exposure"].values[train])

pd.DataFrame(
    {"coefficient": np.concatenate(([t_glm7.intercept_], t_glm7.coef_))},
    index=["intercept"] + t_glm7.feature_names_,
).T
```

```
[12]:
             intercept  DrivAge__0  DrivAge__1  DrivAge__2  DrivAge__3  \
coefficient   5.007277    1.497079     0.53565         0.0   -0.152974

             DrivAge__4  DrivAge__5  DrivAge__6  VehPower__4  VehPower__5  \
coefficient   -0.210998   -0.205689    0.017896    -0.096153     -0.05484

             ...  DrivAge__4__x__VehPower__8  DrivAge__5__x__VehPower__8  \
coefficient  ...                   -0.143822                   -0.002094

             DrivAge__6__x__VehPower__8  DrivAge__0__x__VehPower__9  \
coefficient                    0.512258                    0.730534

             DrivAge__1__x__VehPower__9  DrivAge__2__x__VehPower__9  \
coefficient                   -0.280869                   -0.367669

             DrivAge__3__x__VehPower__9  DrivAge__4__x__VehPower__9  \
coefficient                    0.171063                    0.022052

             DrivAge__5__x__VehPower__9  DrivAge__6__x__VehPower__9
coefficient                   -0.270456                    0.119634

[1 rows x 56 columns]
```

Intercept Term

Just like in the case of the non-formula interface, the presence of an intercept is determined by the `fit_intercept` argument. In case that the formula specifies a different behavior (e.g., adding `+0` or `-1` while `fit_intercept=True`), an error will be raised.

[13]:
```
formula_noint = "PurePremium ~ DrivAge * VehPower - 1"

with pytest.raises(ValueError, match="The formula sets the intercept to False"):
    t_glm8 = GeneralizedLinearRegressor(
        family=TweedieDist,
        alpha_search=True,
        l1_ratio=1,
        fit_intercept=True,
        formula=formula_noint,
        interaction_separator="__x__",
        categorical_format="{name}__{category}",
    ).fit(df_train, sample_weight=df["Exposure"].values[train])
```

One-Sided Formulas

Even when using formulas, the outcome variable can be specified as a vector, as in the interface without formulas. In that case the supplied formula should be one-sided (not contain a `~`), and only describe the right-hand side of the regression.

[14]:
```
formula_onesie = "DrivAge * VehPower"

t_glm8 = GeneralizedLinearRegressor(
    family=TweedieDist,
    alpha_search=True,
    l1_ratio=1,
    fit_intercept=False,
    formula=formula_onesie,
    interaction_separator="__x__",
    categorical_format="{name}__{category}",
)
t_glm8.fit(
    X=df_train, y=df_train["PurePremium"], sample_weight=df["Exposure"].values[train]
)

pd.DataFrame(
    {"coefficient": np.concatenate(([t_glm8.intercept_], t_glm8.coef_))},
    index=["intercept"] + t_glm8.feature_names_,
).T
```

[14]:
```
             intercept  DrivAge__0  DrivAge__1  DrivAge__2  DrivAge__3  \
coefficient        0.0    1.713298    0.783505    0.205914    0.016085

             DrivAge__4  DrivAge__5  DrivAge__6  VehPower__4  VehPower__5  \
coefficient         0.0    0.000094    0.223685      4.66123     4.736272

             ...  DrivAge__4__x__VehPower__8  DrivAge__5__x__VehPower__8  \
coefficient  ...                   -0.144927                    0.001657
```

(continues on next page)

(continued from previous page)

```
             DrivAge__6__x__VehPower__8  DrivAge__0__x__VehPower__9  \
coefficient                    0.515373                    0.714834

             DrivAge__1__x__VehPower__9  DrivAge__2__x__VehPower__9  \
coefficient                   -0.325666                   -0.370935

             DrivAge__3__x__VehPower__9  DrivAge__4__x__VehPower__9  \
coefficient                     0.20417                    0.013222

             DrivAge__5__x__VehPower__9  DrivAge__6__x__VehPower__9
coefficient                   -0.273913                    0.115693

[1 rows x 56 columns]
```

Missing Values in Categorical Columns

By default, `glum` raises a `ValueError` when it encounters a missing value in a categorical variable (`"raise"` option). However, there are two other options for handling these cases. They can also be treated as if they represented all-zeros indicators (`"zero"` option, which is also the way `pandas.get_dummies` works) or missing values can be treated as their own separate category (`"convert"` option).

Similarly to the non-formula-based interface, `glum`'s behavior can be set globally using the `cat_missing_method` parameter during model initialization. However, formulas provide some additional flexibility: the `C` function has a `missing_method` parameter, with which users can select an option on a column-by-column basis. Here is an example of doing that (although our dataset does not have any missing values, so these options have no actual effect in this case):

```
[15]: formula_missing = "C(DrivAge, missing_method='zero') + C(VehPower, missing_method=
      ↪'convert')"

      t_glm9 = GeneralizedLinearRegressor(
          family=TweedieDist,
          alpha_search=True,
          l1_ratio=1,
          fit_intercept=False,
          formula=formula_missing,

      )
      t_glm9.fit(
          X=df_train, y=df_train["PurePremium"], sample_weight=df["Exposure"].values[train]
      )

      pd.DataFrame(
          {"coefficient": np.concatenate(([t_glm9.intercept_], t_glm9.coef_))},
          index=["intercept"] + t_glm9.feature_names_,
      ).T
```

```
[15]:              intercept  C(DrivAge, missing_method='zero')[0]  \
      coefficient        0.0                              1.786703

                   C(DrivAge, missing_method='zero')[1]  \
      coefficient                              0.742765
```

(continues on next page)

(continued from previous page)

```

             C(DrivAge, missing_method='zero')[2]  \
coefficient                              0.239528

             C(DrivAge, missing_method='zero')[3]  \
coefficient                              0.096531

             C(DrivAge, missing_method='zero')[4]  \
coefficient                              0.071118

             C(DrivAge, missing_method='zero')[5]  \
coefficient                                   0.0

             C(DrivAge, missing_method='zero')[6]  \
coefficient                              0.201078

             C(VehPower, missing_method='convert')[4]  \
coefficient                                  4.637267

             C(VehPower, missing_method='convert')[5]  \
coefficient                                  4.679391

             C(VehPower, missing_method='convert')[6]  \
coefficient                                  4.863387

             C(VehPower, missing_method='convert')[7]  \
coefficient                                   4.77263

             C(VehPower, missing_method='convert')[8]  \
coefficient                                  4.749673

             C(VehPower, missing_method='convert')[9]
coefficient                                  4.970188
```

CHAPTER

SIX

CONTRIBUTING AND DEVELOPMENT

Hello! And thanks for exploring glum more deeply. Please see the issue tracker and pull requests tabs on Github for information about what is currently happening. Feel free to post an issue if you'd like to get involved in development and don't really know where to start – we can give some advice.

We welcome contributions of any kind!

- New features
- Feature requests
- Bug reports
- Documentation
- Tests
- Questions

6.1 Pull request process

- Before working on a non-trivial PR, please first discuss the change you wish to make via issue, Slack, email or any other method with the owners of this repository. This is meant to prevent spending time on a feature that will not be merged.
- Please make sure that a new feature comes with adequate tests. If these require data, please check if any of our existing test data sets fits the bill.
- Please make sure that all functions come with proper docstrings. If you do extensive work on docstrings, please check if the Sphinx documentation renders them correctly. ReadTheDocs builds on every commit to an open pull request. You can see whether the documentation has successfully built in the "checks" section of the PR. Once the build finishes, your documentation should be accessible by clicking the "details" link next to the check in the GitHub interface and will appear at a URL like: `https://glum--###.org.readthedocs.build/en/###/` where `###` is the number of your PR.
- Please make sure you have our pre-commit hooks installed.
- If you fix a bug, please consider first contributing a test that _fails_ because of the bug and then adding the fix as a separate commit, so that the CI system picks it up.
- Please add an entry to the change log and increment the version number according to the type of change. We use semantic versioning. Update the major if you break the public API. Update the minor if you add new functionality. Update the patch if you fixed a bug. All changes that have not been released are collected under the date `UNRELEASED`.

6.2 Releases

- We make package releases infrequently, but usually any time a new non-trivial feature is contributed or a bug is fixed. To make a release, just open a PR that updates the change log with the current date. Once that PR is approved and merged, you can create a new release on [GitHub](https://github.com/Quantco/glum/releases/new). Use the version from the change log as tag and copy the change log entry into the release description.

6.3 Install for development

The first step is to set up a conda environment and install glum in editable mode. We strongly suggest to use `mamba` instead of `conda` as this provides the same functionality at much greater speed.

```
# First, make sure you have conda-forge as your primary conda channel:
conda config --add channels conda-forge

# Clone the repository
git clone git@github.com:Quantco/glum.git
cd glum

# Set up a conda environment with name "glum"
mamba env create

# If you want to install the dependencies necessary for benchmarking against other GLM packages:
mamba env update -n glum --file environment-benchmark.yml

# If you want to work on the tutorial notebooks:
mamba env update -n glum --file environment-tutorial.yml

# Activate the previously created conda environment
conda activate glum

# Set up our pre-commit hooks for black, mypy, isort and flake8.
pre-commit install

# Install this package in editable mode.
pip install --no-use-pep517 --disable-pip-version-check -e .
```

6.4 Testing and continuous integration

The test suite is in `tests/`.

6.4.1 Golden master tests

We use golden master testing to preserve correctness. The results of many different GLM models have been saved. After an update, the tests will compare the new output to the saved models. Any significant deviation will result in a test failure. This doesn't strictly mean that the update was wrong. In case of a bug fix, it's possible that the new output will be more accurate than the old output. In that situation, the golden master results can be overwritten as explained below.

There are two sets of golden master tests, one with artificial data and one directly using the benchmarking problems from `glum_benchmarks`. For both sets of tests, creating the golden master and the tests definition are located in the same file. Calling the file with pytest will run the tests while calling the file as a python script will generate the golden master result. When creating the golden master results, both scripts accept the `--overwrite` command line flag. If set, the existing golden master results will be overwritten. Otherwise, only the new problems will be run.

6.4.2 Skipping the slow tests

If you want to skip the slow tests, add the `-m "not slow"` flag to any pytest command. The "wide" problems (all marked as slow tests) are especially poorly conditioned. This means that even for estimation with 10k observations, it might still be very slow. Furthermore, we also have golden master tests for the "narrow" and "intermediate" problems, so adding the "wide" problems do not add much coverage.

6.4.3 Storing and modifying

To store the golden master results:

```
python tests/glm/test_golden_master.py
python tests/glm/test_benchmark_golden_master.py
```

Add the `--overwrite` flag if you want to overwrite already existing golden master results

6.5 Building a conda package

To use the package in another project, we distribute it as a conda package. For building the package locally, you can use the following command:

```
conda build conda.recipe
```

This will build the recipe using the standard compiler flags set by the conda-forge activation scripts.

6.6 Developing the documentation

The documentation is built with a mix of Sphinx, autodoc, and nbsphinx. To develop the documentation:

```
cd docs
make html
python -m http.server --directory _build/html
```

Then, navigate to http://localhost:8000 to view the documentation.

Alternatively, if you install entr, then you can auto-rebuild the documentation any time a file changes with:

```
cd docs
./dev
```

Note: The tutorial notebooks are not executed as part of the documentation build. If you want to modify them, make sure to execute them manually and save the output before committing. Also don't forget to install the extra dependencies for the tutorial notebooks as described above.

If you are a newbie to Sphinx, the links below may help get you up to speed on some of the trickier aspects:

- An idiot's guide to Sphinx
- Links between documents
- Cross-referencing python objects using things like `:mod:` and `:meth:` and `:class:`.
- autodoc is used for automatically converting docstrings to docs
- We follow the numpy docstring style guide
- To create links between ipynb files when using nbsphinx

6.7 Where to start looking in the source?

The primary user interface of `glum` consists of the *`GeneralizedLinearRegressor`* and *`GeneralizedLinearRegressorCV`* classes via their constructors and the *`fit()`* and *`predict()`* functions. Those are the places to start looking if you plan to change the system in some way.

What follows is a high-level summary of the source code structure. For more details, please look in the documentation and docstrings of the relevant classes, functions and methods.

- `_glm.py` - This is the main entrypoint and implements the core logic of the GLM. Most of the code in this file handles input arguments and prepares the data for the GLM fitting algorithm.
- `_glm_cv.py` - This is the entrypoint for the cross validated GLM implementation. It depends on a lot of the code in `_glm.py` and only modifies the sections necessary for running training many models with different regularization parameters.
- `_solvers.py` - This contains the bulk of the IRLS and L-BFGS algorithms for training GLMs.
- `_cd_fast.pyx` - This is a Cython implementation of the coordinate descent algorithm used for fitting L1 penalty GLMs. Note the `.pyx` extension indicating that it is a Cython source file.
- `_distribution.py` - definitions of the distributions that can be used. Includes Normal, Poisson, Gamma, InverseGaussian, Tweedie, Binomial and GeneralizedHyperbolicSecant distributions.

- `_link.py` - definitions of the link functions that can be used. Includes identity, log, logit and Tweedie link functions.
- `_functions.pyx` - This is a Cython implementation of the log likelihoods, gradients and Hessians for several popular distributions.
- `_util.py` - This contains a few general purpose linear algebra routines that serve several other modules and don't fit well elsewhere.

6.8 The GLM benchmark suite

Before deciding to build a library custom built for our purposes, we did an thorough investigation of the various open source GLM implementations available. This resulted in an extensive suite of benchmarks for comparing the correctness, runtime and availability of features for these libraries.

The benchmark suite has two command line entrypoints:

- `glm_benchmarks_run`
- `glm_benchmarks_analyze`

Both of these CLI tools take a range of arguments that specify the details of the benchmark problems and which libraries to benchmark.

For more details on the benchmark suite, see the README in the source at `src/glum_benchmarks/README.md`.

CHAPTER

SEVEN

AN INTRODUCTION TO THE ALGORITHMS USED IN `GLUM`

Before continuing, please take a look at the sklearn documentation for a high-level intro to generalized linear models.

In addition, please take a look at *the tutorials.*

For mathematical and algorithmic details, see below:

7.1 What is a GLM?

This package is intended to fit L1 and L2-norm penalized generalized linear models (GLMs). What is a GLM?

A GLM is a linear model ($\eta = x^\top \beta$) wrapped in a transformation (link function) and equipped with a response distribution from an exponential family. The choice of link function and response distribution is very flexible. In a GLM, a predictive distribution for the response variable Y is associated with a vector of observed predictors x. The distribution has the form:

$$\begin{align} p(y \,|\, x) &= m(y, \phi) \exp\left(\frac{\theta\, T(y) - A(\theta)}{\phi}\right) && \text{random component / distribution family} \tag{7.1} \\ \theta &:= g(\eta) && \text{systematic component / link function} \tag{7.2} \\ \eta &:= x^\top \beta && \text{parametric link component / linear predictor} \tag{7.3} \end{align}$$

Here β are the parameters; ϕ is a parameter representing dispersion ("variance"); and m, T, A are characterized by the user-specified model family; and g is the **link function.** For background on the exponential family, and how common distributions can be represented in the form of the "random component" above, see Wikipedia | Exponential family.

The expected mean of Y depends on x by composition of **linear response** η and (inverse) link function, i.e.:

$$E[y|\eta] := \mu = g^{-1}(\eta).$$

Proving this involves use of moment-generating functions and cumulants.

Defining an offset of γ, we can write the linear predictor as $\eta = X^T \beta + \gamma$.

7.2 Examples of GLMs

7.2.1 Normal

Many of the "regression" models we commonly work with are GLMs. For example, the normal distribution with a linear link function is a simple GLM: $y|x \sim N(x^T \beta, \sigma^2)$. If we fit β with maximum likelihood, this is least-squares regression. If we replace the linear link function with an exponential link function, we get a simple multiplicative model that is also a GLM: $y|x \sim N(e^{x^T \beta}, \sigma^2)$.

7.2.2 Tweedie

In the insurance context, we usually work with a Tweedie distribution, which is a generalization of Poisson ($p = 1$) and Gamma ($p = 2$):

$$\begin{aligned} \theta &= \left\{ \begin{array}{ll} \frac{\mu^{1-p}}{1-p} & \text{if } p \neq 1 \\ \log \mu & \text{if } p = 1 \end{array} \right. , \\ T(y) &= y, \\ A(\theta) &= \left\{ \begin{array}{ll} \frac{\mu^{2-p}}{2-p} & \text{if } p \neq 2 \\ \log \mu & \text{if } p = 2. \end{array} \right. , \end{aligned}$$

With $1 < p < 2$, the Tweedie distribution is a compound Poisson-gamma distribution. y is distributed as if a number N is drawn from a Poisson distribution, and then N draws are taken IID from a gamma distribution and added. This distribution might model the total amount of a claim in an insurance context: There are N incidents, and each incident i has an amount X_i, and the total amount is the sum of X_i.

7.3 Objective function

To fit a GLM, we minimize the negative log-likelihood (or typically the unit deviance) subject to an elastic net constraint involving a mix of L1 and L2 penalty terms:

$$\min_{\beta} \mathcal{L} + \alpha\rho||\beta||_1 + \frac{\alpha(1-\rho)}{2}||\beta||_2^2$$

7.4 Optimizers/solvers

There are three solvers implemented in `glum`:

- `lbfgs` - This solver uses the scipy `fmin_l_bfgs_b` optimizer to minimize L2-penalized GLMs. The L-BFGS solver does not work with L1-penalties. Because L-BFGS does not store the full Hessian, it can be particularly effective for very high dimensional problems with several thousand or more columns. For more details, see the scipy documentation.
- `irls-cd` and `irls-ls` - These solvers are both based on Iteratively Reweighted Least Squares (IRLS). IRLS proceeds by iteratively approximating the objective function with a quadratic, then solving that quadratic for the optimal update. For purely L2-penalized settings, the `irls-ls` uses a least squares inner solver for each quadratic subproblem. For problems that have any L1-penalty component, the `irls-cd` uses a coordinate descent inner solver for each quadratic subproblem. The IRLS-LS and IRLS-CD implementations largely follow the algorithm described in `newglmnet` (see references below).

7.4.1 Convexity

Our objective function will generally be convex, because the log-likelihoods of members of the exponential family are convex in their "natural parameterizations." The natural parameterization may not be the most obvious one. Example: The log-likelihood of the normal distribution is convex in one over the variance, which is its natural parameterization. It is not convex in the variance. We can generally assume we are solving convex problems.

An exception is multicollinearity (rank deficiency) in the design matrix, without an L2 component to the penalty. In that case, the problem will be only weakly convex and will have no unique miminum. This is not an arcane consideration, since we frequently generate rank deficiency by constructing multiple sets of one-hot encoded categorical variables. This can make evaluating different optimizers tricky, since they could converge to different equally good optima. The original glmnet paper suggests using at least a small L2 regularization component to remove “degeneracies and wild behavior.”

7.4.2 IRLS

We minimize the log likelihood using Iteratively Reweighted Least Squares (IRLS). IRLS can be justified by seeing it as taking a Newton step, but replacing the Hessian with the expected Hessian.

In the irls-cd and irls-ls solvers, the outer loop is an IRLS iteration that forms a quadratic approximation to the negative loglikelihood. That is, we find w and z so that the problem can be expressed as:

```
min sum_i w_i (z_i - x_i beta)^2 + penalty
```

We exit when either the gradient is small (gradient_tol) or the step size is small (step_size_tol). Both of these tolerances are user configurable.

Once we have formed this quadratic approximation, an “inner solver” finds the minimum of the quadratic. In the irls-ls solver, the inner solver is simply a direct least squares solve.

See the glmintro reference for an excellent discussion of IRLS in the context of GLMs.

7.4.3 Coordinate Descent

With an L1 penalty, we use the irls-cd solver where the inner solver finds the minimum of the quadratic using coordinate descent. We exit the inner loop when the quadratic problem’s gradient is small. The classic reference here is the glmnet paper.

However, coordinate descent is older than the glmnet paper, and is a simple idea. In a problem with data y and x and parameters “params”, Coordinate Descent involves repeatedly optimizing one or several parameters while holding the rest fixed. Interestingly, Wikipedia claims that coordinate descent is often overlooked among researchers because it is simple to implement, and they would rather work on something more interesting.

[1]:

```
"""
Cyclical coordinate descent with Newton-like steps and a
twice-differentiable objective function.
"""
def coordinate_descent_inner(y, x, params, grad_func, hess_func):
    for i, elt in enumerate(params):
        step = -grad_func(y, x, params) / hess_func(y, x, params)
        params[i] += step
    return params

def coordinate_descent(y, x, params, grad_func, hess_func,
                       convergence_func):
    while not convergence_func():
        params = coordinate_descent_inner(y, x, params, grad_func,
                                          hess_func)
    return params
```

In practice, the implementation is a bit more complicated when the objective function is not differentiable. See also the coordinate_descent reference below.

7.4.4 Active set tracking

When penalizing with an L1-norm, it is common for many coefficients to be exactly zero. And, it is possible to predict during a given iteration which of those coefficients will stay zero. As a result, we track the “active set” consisting of all the coefficients that are either currently non-zero or likely to remain non-zero. We follow the outer loop active set tracking algorithm in the `newglmnet` reference. That paper refers to the same concept as “shrinkage”, whereas the `glmnet` reference calls this the “active set”. Currently, we have not yet implemented the inner loop active set tracking from the `newglmnet` reference.

7.5 Matrix Types

Along with the GLM solvers, this package supports dense, sparse, categorical matrix types and mixtures of these types. Using the most efficient matrix representations massively improves performances.

For more details, see the README for tabmat

- We support dense matrices via standard numpy arrays.
- We support sparse CSR and CSC matrices via standard `scipy.sparse` objects. However, we have extended these operations with custom matrix-vector and sandwich product routines that are optimized and parallelized. A user does not need to modify their code to take advantage of this optimization. If a `scipy.sparse.csc_matrix` object is passed in, it will be automatically converted to a `SparseMatrix` object. This operation is almost free because no data needs to be copied.
- We implement a `CategoricalMatrix` object that efficiently represents these matrices without nearly as much overhead as a normal CSC or CSR sparse matrix.
- Finally, `SplitMatrix` allows mixing different matrix types for different columns to minimize overhead.

7.6 Standardization

Internal to our solvers, all matrix types are wrapped in a `tabmat.StandardizedMatrix` which offsets columns to have mean zero and standard deviation one without modifying the matrix data itself. This avoids situations where modifying a matrix to have mean zero would result in losing the sparsity structure. It also avoids ever needing to copy or modify the input data matrix. As a result, excess memory usage is very low in `glum`.

7.7 References

`glmnet` - Regularization Paths for Generalized Linear Models via Coordinate Descent

`newglmnet` - An Improved GLMNET for L1-regularized LogisticRegression

`glmintro` - Bryan Lewis on GLMs

`coordinate_descent` - Coordinate Descent Algorithms

`glmbook` - Generalized Linear Models, McCullagh and Nelder

[ ]:

CHAPTER

EIGHT

GLUM PACKAGE

The two main classes in *glum* are *GeneralizedLinearRegressor* and *GeneralizedLinearRegressorCV*. Most users will use *fit()* and *predict()*

class glum.**BinomialDistribution**

Bases: *ExponentialDispersionModel*

A class for the Binomial distribution.

The Binomial distribution models outcomes `y` in `[0, 1]`.

See the documentation of the superclass, *ExponentialDispersionModel*, for details.

deviance(*y*, *mu*, *sample_weight=1*)

Compute the deviance.

The deviance is a weighted sum of the unit deviances. In terms of the unit log likelihood ℓ, it equals $2 \sum_i [\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which the variance is inversely proportional.

Return type

float

deviance_derivative(*y*, *mu*, *sample_weight=1*)

Compute the derivative of the deviance with respect to `mu`.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,) (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

dispersion(*y*, *mu*, *sample_weight=None*, *ddof=1*, *method='pearson'*)

Estimate the dispersion parameter ϕ.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inversely proportional.
- **ddof** (*int, optional (default=1)*) – Degrees of freedom consumed by the model for `mu`.
- **{'pearson'** (*method =*) – Whether to base the estimate on the Pearson residuals or the deviance.
- **'deviance'}** – Whether to base the estimate on the Pearson residuals or the deviance.
- **(default='pearson')** (*optional*) – Whether to base the estimate on the Pearson residuals or the deviance.

Return type
float

`eta_mu_deviance`(*link*, *factor*, *cur_eta*, *X_dot_d*, *y*, *sample_weight*)

Compute `eta`, `mu` and the deviance.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The linear predictor, `eta`, as `cur_eta + factor * X_dot_d`.
- *numpy.ndarray, shape (X.shape[0],)* – The link-function-transformed prediction, `mu`.
- *float* – The deviance.

Parameters

- **link** (`Link`)
- **factor** (*float*)

Return type
tuple[*ndarray*, *ndarray*, float]

`in_y_range`(*x*)

Return `True` if `x` is in the valid range of the EDM.

Return type
ndarray

`log_likelihood`(*y*, *mu*, *sample_weight=None*, *dispersion=1*)

Compute the log likelihood.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Sample weights.
- **dispersion** (*float, optional (default=1)*) – Ignored.

Return type
float

rowwise_gradient_hessian(*link*, *coef*, *dispersion*, *X*, *y*, *sample_weight*, *eta*, *mu*, *offset=None*)

Compute the gradient and negative Hessian of the log likelihood row-wise.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The gradient of the log likelihood, row-wise.
- *numpy.ndarray, shape (X.shape[0],)* – The negative Hessian of the log likelihood, row-wise.

Parameters

- **link** (Link)
- **X** (*MatrixBase | StandardizedMatrix*)

to_tweedie(*safe=True*)

Return the Tweedie representation of a distribution if it exists.

unit_deviance(*y*, *mu*)

Compute the unit deviance.

In terms of the unit log likelihood ℓ, the unit deviance is $2[\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

unit_deviance_derivative(*y*, *mu*)

Compute the derivative of the unit deviance with respect to mu.

The derivative of the unit deviance is given by $2 \times (\mu - y)/v(\mu)$, where $v(\mu)$ is the unit variance.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

array-like, shape (n_samples,)

unit_variance(*mu*)

Compute the unit variance.

The unit variance, $v(\mu) \equiv b''((b')^{-1}(\mu))$, determines the variance as a function of the mean μ by $\mathrm{var}(y_i) = v(\mu_i) \times \phi / w_i$. It can also be derived from the unit deviance $d(y, \mu)$ as

$$v(\mu) = 2 \div \frac{\partial^2 d(y, \mu)}{\partial \mu^2}\big|_{y=\mu}.$$

See also variance().

Parameters

mu (*array-like, shape (n_samples,)*) – Predicted mean.

unit_variance_derivative(*mu*)

Compute the derivative of the unit variance with respect to mu.

Parameters

mu (*array-like, shape (n_samples,)*) – Predicted mean.

`variance`(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the variance function.

The variance of $Y_i \sim \mathrm{EDM}(\mu_i, \phi/w_i)$ takes the form $v(\mu_i) \times \phi/w_i$, where $v(\mu)$ is the unit variance and w_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

`variance_derivative`(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the derivative of the variance with respect to `mu`.

The derivative of the variance is equal to $v(\mu_i) \times \phi/w_i$, where $v(\mu)$ is the unit variance and ws_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

`class glum.CloglogLink`

Bases: *Link*

The complementary log-log link function `log(-log(-p))`.

`derivative`(*mu*)

Compute the derivative of the link function.

Parameters

mu (*array-like, shape (n_samples,)*) – Usually the (predicted) mean.

`inverse`(*lin_pred*)

Compute the inverse link function.

The inverse link function `h` gives the inverse relationship between the linear predictor, `X * w`, and the mean, `mu  E(Y)`, so that `h(X * w) = mu`.

Parameters

lin_pred (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

`inverse_derivative`(*lin_pred*)

Compute the derivative of the inverse link function.

Parameters

lin_pred (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

inverse_derivative2(*lin_pred*)

Compute second derivative of the inverse link function.

Parameters

lin_pred (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

link(*mu*)

Compute the link function.

The link function `g` links the mean, `mu  E(Y)`, to the linear predictor, `X * w`, so that `g(mu)` is equal to the linear predictor.

Parameters

mu (*array-like, shape (n_samples,)*) – Usually the (predicted) mean.

to_tweedie(*safe=True*)

Return the Tweedie representation of a link function if it exists.

class glum.**ExponentialDispersionModel**

Bases: `object`

Base class for reproductive Exponential Dispersion Models (EDM).

The PDF of $Y \sim \mathrm{EDM}(\theta, \phi)$ is given by

$$p(y \mid \theta, \phi) = \exp\left(\frac{y\theta - b(\theta)}{\phi/w} + c(y; w/\phi)\right),$$

where θ is the scale parameter, ϕ is the dispersion parameter, w is a given weight, b is the cumulant function and c is a normalization term.

It can be shown that $\mathrm{E}(Y) = b'(\theta)$ and $\mathrm{var}(Y) = b''(\theta) \times \phi/w$.

References

< https://en.wikipedia.org/wiki/Exponential_dispersion_model >.

deviance(*y*, *mu*, *sample_weight=1*)

Compute the deviance.

The deviance is a weighted sum of the unit deviances. In terms of the unit log likelihood ℓ, it equals $2\sum_i[\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which the variance is inversely proportional.

Return type

float

deviance_derivative(*y*, *mu*, *sample_weight=1*)

Compute the derivative of the deviance with respect to `mu`.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,) (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

`dispersion`(*y*, *mu*, *sample_weight=None*, *ddof=1*, *method='pearson'*)

Estimate the dispersion parameter ϕ.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inversely proportional.
- **ddof** (*int, optional (default=1)*) – Degrees of freedom consumed by the model for `mu`.
- **{'pearson'** (*method =*) – Whether to base the estimate on the Pearson residuals or the deviance.
- **'deviance'}** – Whether to base the estimate on the Pearson residuals or the deviance.
- **(default='pearson')** (*optional*) – Whether to base the estimate on the Pearson residuals or the deviance.

Return type
float

`eta_mu_deviance`(*link*, *factor*, *cur_eta*, *X_dot_d*, *y*, *sample_weight*)

Compute `eta`, `mu` and the deviance.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The linear predictor, `eta`, as `cur_eta + factor * X_dot_d`.
- *numpy.ndarray, shape (X.shape[0],)* – The link-function-transformed prediction, `mu`.
- *float* – The deviance.

Parameters

- **link** (`Link`)
- **factor** (*float*)

Return type
tuple[*ndarray*, *ndarray*, float]

`in_y_range`(*x*)

Return `True` if `x` is in the valid range of the EDM.

Return type
ndarray

`abstract property include_lower_bound: bool`

Return whether `lower_bound` is allowed as a value of `y`.

abstract property include_upper_bound: bool

Return whether `upper_bound` is allowed as a value of `y`.

abstract property lower_bound: float

Get the lower bound of values for the EDM.

rowwise_gradient_hessian(*link*, *coef*, *dispersion*, *X*, *y*, *sample_weight*, *eta*, *mu*, *offset=None*)

Compute the gradient and negative Hessian of the log likelihood row-wise.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The gradient of the log likelihood, row-wise.
- *numpy.ndarray, shape (X.shape[0],)* – The negative Hessian of the log likelihood, row-wise.

Parameters

- **link** (`Link`)
- **X** (*MatrixBase | StandardizedMatrix*)

to_tweedie(*safe=True*)

Return the Tweedie representation of a distribution if it exists.

abstract unit_deviance(*y*, *mu*)

Compute the unit deviance.

In terms of the unit log likelihood ℓ, the unit deviance is $2[\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

unit_deviance_derivative(*y*, *mu*)

Compute the derivative of the unit deviance with respect to `mu`.

The derivative of the unit deviance is given by $2 \times (\mu - y)/v(\mu)$, where $v(\mu)$ is the unit variance.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

array-like, shape (n_samples,)

abstract unit_variance(*mu*)

Compute the unit variance.

The unit variance, $v(\mu) \equiv b''((b')^{-1}(\mu))$, determines the variance as a function of the mean μ by $\text{var}(y_i) = v(\mu_i) \times \phi / w_i$. It can also be derived from the unit deviance $d(y, \mu)$ as

$$v(\mu) = 2 \div \frac{\partial^2 d(y, \mu)}{\partial \mu^2}\big|_{y=\mu}.$$

See also `variance()`.

Parameters

mu (*array-like, shape (n_samples,)*) – Predicted mean.

abstract unit_variance_derivative(*mu*)

Compute the derivative of the unit variance with respect to `mu`.

Parameters

mu (*array-like, shape (n_samples,)*) – Predicted mean.

abstract property upper_bound: float

Get the upper bound of values for the EDM.

variance(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the variance function.

The variance of $Y_i \sim \mathrm{EDM}(\mu_i, \phi/w_i)$ takes the form $v(\mu_i) \times \phi/w_i$, where $v(\mu)$ is the unit variance and w_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

variance_derivative(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the derivative of the variance with respect to `mu`.

The derivative of the variance is equal to $v(\mu_i) \times \phi/w_i$, where $v(\mu)$ is the unit variance and ws_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

class glum.**GammaDistribution**

Bases: *ExponentialDispersionModel*

Class for the gamma distribution.

The gamma distribution models outcomes `y` in `(0, +∞)`.

See the documentation of the superclass, *ExponentialDispersionModel*, for details.

deviance(*y*, *mu*, *sample_weight=None*)

Compute the deviance.

The deviance is a weighted sum of the unit deviances. In terms of the unit log likelihood ℓ, it equals $2 \sum_i [\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which the variance is inversely proportional.

Return type
float

deviance_derivative(*y*, *mu*, *sample_weight=1*)

Compute the derivative of the deviance with respect to `mu`.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,) (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

dispersion(*y*, *mu*, *sample_weight=None*, *ddof=1*, *method='pearson'*)

Estimate the dispersion parameter ϕ.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inversely proportional.
- **ddof** (*int, optional (default=1)*) – Degrees of freedom consumed by the model for `mu`.
- **{'pearson'** (*method =*) – Whether to base the estimate on the Pearson residuals or the deviance.
- **'deviance'}** – Whether to base the estimate on the Pearson residuals or the deviance.
- **(default='pearson')** (*optional*) – Whether to base the estimate on the Pearson residuals or the deviance.

Return type
float

eta_mu_deviance(*link*, *factor*, *cur_eta*, *X_dot_d*, *y*, *sample_weight*)

Compute `eta`, `mu` and the deviance.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The linear predictor, `eta`, as `cur_eta + factor * X_dot_d`.
- *numpy.ndarray, shape (X.shape[0],)* – The link-function-transformed prediction, `mu`.
- *float* – The deviance.

Parameters

- **link** (`Link`)
- **factor** (*float*)

Return type
tuple[*ndarray*, *ndarray*, float]

`in_y_range`(*x*)

Return `True` if `x` is in the valid range of the EDM.

Return type
ndarray

`log_likelihood`(*y*, *mu*, *sample_weight=None*, *dispersion=None*)

Compute the log likelihood.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Sample weights.
- **dispersion** (*float, optional (default=None)*) – Dispersion parameter ϕ. Estimated if `None`.

Return type
float

`rowwise_gradient_hessian`(*link*, *coef*, *dispersion*, *X*, *y*, *sample_weight*, *eta*, *mu*, *offset=None*)

Compute the gradient and negative Hessian of the log likelihood row-wise.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The gradient of the log likelihood, row-wise.
- *numpy.ndarray, shape (X.shape[0],)* – The negative Hessian of the log likelihood, row-wise.

Parameters

- **link** (`Link`)
- **X** (*MatrixBase | StandardizedMatrix*)

`to_tweedie`(*safe=True*)

Return the Tweedie representation of a distribution if it exists.

`unit_deviance`(*y*, *mu*)

Compute the unit deviance.

In terms of the unit log likelihood ℓ, the unit deviance is $2[\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

`unit_deviance_derivative`(*y*, *mu*)

Compute the derivative of the unit deviance with respect to `mu`.

The derivative of the unit deviance is given by $2 \times (\mu - y)/v(\mu)$, where $v(\mu)$ is the unit variance.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

Return type
array-like, shape (n_samples,)

unit_variance(*mu*)

Compute the unit variance.

The unit variance, $v(\mu) \equiv b''((b')^{-1}(\mu))$, determines the variance as a function of the mean μ by $\mathrm{var}(y_i) = v(\mu_i) \times \phi / w_i$. It can also be derived from the unit deviance $d(y, \mu)$ as

$$v(\mu) = 2 \div \frac{\partial^2 d(y, \mu)}{\partial \mu^2}\big|_{y=\mu}.$$

See also `variance()`.

Parameters
mu (*array-like, shape (n_samples,)*) – Predicted mean.

Return type
ndarray

unit_variance_derivative(*mu*)

Compute the derivative of the unit variance with respect to `mu`.

Parameters
mu (*array-like, shape (n_samples,)*) – Predicted mean.

Return type
ndarray

variance(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the variance function.

The variance of $Y_i \sim \mathrm{EDM}(\mu_i, \phi / w_i)$ takes the form $v(\mu_i) \times \phi / w_i$, where $v(\mu)$ is the unit variance and w_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

variance_derivative(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the derivative of the variance with respect to `mu`.

The derivative of the variance is equal to $v(\mu_i) \times \phi / w_i$, where $v(\mu)$ is the unit variance and ws_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

class glum.**GeneralizedHyperbolicSecant**

Bases: *ExponentialDispersionModel*

A class for the Generalized Hyperbolic Secant (GHS) distribution.

The GHS distribution models outcomes `y` in `(-∞, +∞)`.

See the documentation of the superclass, *ExponentialDispersionModel*, for details.

deviance(*y*, *mu*, *sample_weight=1*)

Compute the deviance.

The deviance is a weighted sum of the unit deviances. In terms of the unit log likelihood ℓ, it equals $2 \sum_i [\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which the variance is inversely proportional.

Return type

float

deviance_derivative(*y*, *mu*, *sample_weight=1*)

Compute the derivative of the deviance with respect to `mu`.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,) (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

dispersion(*y*, *mu*, *sample_weight=None*, *ddof=1*, *method='pearson'*)

Estimate the dispersion parameter ϕ.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inversely proportional.
- **ddof** (*int, optional (default=1)*) – Degrees of freedom consumed by the model for `mu`.
- **{'pearson'** (*method =*) – Whether to base the estimate on the Pearson residuals or the deviance.
- **'deviance'}** – Whether to base the estimate on the Pearson residuals or the deviance.
- **(default='pearson')** (*optional*) – Whether to base the estimate on the Pearson residuals or the deviance.

Return type
float

`eta_mu_deviance`(*link*, *factor*, *cur_eta*, *X_dot_d*, *y*, *sample_weight*)

Compute `eta`, `mu` and the deviance.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The linear predictor, `eta`, as `cur_eta + factor * X_dot_d`.
- *numpy.ndarray, shape (X.shape[0],)* – The link-function-transformed prediction, `mu`.
- *float* – The deviance.

Parameters

- **link** (`Link`)
- **factor** (*float*)

Return type
tuple[*ndarray*, *ndarray*, float]

`in_y_range`(*x*)

Return `True` if `x` is in the valid range of the EDM.

Return type
ndarray

`rowwise_gradient_hessian`(*link*, *coef*, *dispersion*, *X*, *y*, *sample_weight*, *eta*, *mu*, *offset=None*)

Compute the gradient and negative Hessian of the log likelihood row-wise.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The gradient of the log likelihood, row-wise.
- *numpy.ndarray, shape (X.shape[0],)* – The negative Hessian of the log likelihood, row-wise.

Parameters

- **link** (`Link`)
- **X** (*MatrixBase | StandardizedMatrix*)

`to_tweedie`(*safe=True*)

Return the Tweedie representation of a distribution if it exists.

`unit_deviance`(*y*, *mu*)

Compute the unit deviance.

In terms of the unit log likelihood ℓ, the unit deviance is $2[\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

Return type
ndarray

`unit_deviance_derivative`(*y*, *mu*)

Compute the derivative of the unit deviance with respect to `mu`.

The derivative of the unit deviance is given by $2 \times (\mu - y)/v(\mu)$, where $v(\mu)$ is the unit variance.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

array-like, shape (n_samples,)

`unit_variance`(*mu*)

Compute the unit variance.

The unit variance, $v(\mu) \equiv b''((b')^{-1}(\mu))$, determines the variance as a function of the mean μ by $\mathrm{var}(y_i) = v(\mu_i) \times \phi/w_i$. It can also be derived from the unit deviance $d(y, \mu)$ as

$$v(\mu) = 2 \div \frac{\partial^2 d(y, \mu)}{\partial \mu^2}\big|_{y=\mu}.$$

See also `variance()`.

Parameters

mu (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

ndarray

`unit_variance_derivative`(*mu*)

Compute the derivative of the unit variance with respect to `mu`.

Parameters

mu (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

ndarray

`variance`(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the variance function.

The variance of $Y_i \sim \mathrm{EDM}(\mu_i, \phi/w_i)$ takes the form $v(\mu_i) \times \phi/w_i$, where $v(\mu)$ is the unit variance and w_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

`variance_derivative`(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the derivative of the variance with respect to `mu`.

The derivative of the variance is equal to $v(\mu_i) \times \phi/w_i$, where $v(\mu)$ is the unit variance and ws_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

class glum.**GeneralizedLinearRegressor**(**, alpha=None, l1_ratio=0, P1='identity', P2='identity', fit_intercept=True, family='normal', link='auto', solver='auto', max_iter=100, gradient_tol=None, step_size_tol=None, hessian_approx=0.0, warm_start=False, alpha_search=False, alphas=None, n_alphas=100, min_alpha_ratio=None, min_alpha=None, start_params=None, selection='cyclic', random_state=None, copy_X=None, check_input=True, verbose=0, scale_predictors=False, lower_bounds=None, upper_bounds=None, A_ineq=None, b_ineq=None, force_all_finite=True, drop_first=False, robust=True, expected_information=False, formula=None, interaction_separator=':', categorical_format='{name}[{category}]', cat_missing_method='fail', cat_missing_name='(MISSING)'*)

Bases: `GeneralizedLinearRegressorBase`

Regression via a Generalized Linear Model (GLM) with penalties.

GLMs based on a reproductive Exponential Dispersion Model (EDM) aimed at fitting and predicting the mean of the target `y` as `mu=h(X*w)`. Therefore, the fit minimizes the following objective function with combined L1 and L2 priors as regularizer:

```
1/(2*sum(s)) * deviance(y, h(X*w); s)
+ alpha * l1_ratio * ||P1*w||_1
+ 1/2 * alpha * (1 - l1_ratio) * w*P2*w
```

with inverse link function `h` and `s=sample_weight`. Note that, for `alpha=0` the unregularized GLM is recovered. This is not the default behavior (see `alpha` parameter description for details). Additionally, for `sample_weight=None`, one has `s_i=1` and `sum(s)=n_samples`. For `P1=P2='identity'`, the penalty is the elastic net:

```
alpha * l1_ratio * ||w||_1 + 1/2 * alpha * (1 - l1_ratio) * ||w||_2^2.
```

If you are interested in controlling the L1 and L2 penalties separately, keep in mind that this is equivalent to:

```
a * L1 + b * L2,
```

where:

```
alpha = a + b and l1_ratio = a / (a + b).
```

The parameter `l1_ratio` corresponds to alpha in the R package glmnet, while `alpha` corresponds to the lambda parameter in glmnet. Specifically, `l1_ratio = 1` is the lasso penalty.

Read more in *background*.

Parameters

- **alpha** (*{float, array-like}, optional (default=None)*) – Constant that multiplies the penalty terms and thus determines the regularization strength. If alpha_search is False (the default), then alpha must be a scalar or None (equivalent to alpha=0). If alpha_search is True, then alpha must be an iterable or None. See alpha_search to find how the regularization path is set if alpha is None. See the notes for the exact mathematical meaning of this parameter. alpha=0 is equivalent to unpenalized GLMs. In this case, the design matrix X must have full column rank (no collinearities).
- **l1_ratio** (*float, optional (default=0)*) – The elastic net mixing parameter, with 0 <= l1_ratio <= 1. For l1_ratio = 0, the penalty is an L2 penalty. For l1_ratio = 1, it is an L1 penalty. For 0 < l1_ratio < 1, the penalty is a combination of L1 and L2.
- **P1** (*{'identity', array-like, None}, shape (n_features,), optional (default='identity')*) – This array controls the strength of the regularization for each coefficient independently. A high value will lead to higher regularization while a value of zero will remove the regularization on this parameter. Note that n_features = X.shape[1]. If X is a pandas DataFrame with a categorical dtype and P1 has the same size as the number of columns, the penalty of the categorical column will be applied to all the levels of the categorical.
- **P2** (*{'identity', array-like, sparse matrix, None}, shape (n_features,) or (n_features, n_features), optional (default='identity')*) – With this option, you can set the P2 matrix in the L2 penalty w*P2*w. This gives a fine control over this penalty (Tikhonov regularization). A 2d array is directly used as the square matrix P2. A 1d array is interpreted as diagonal (square) matrix. The default 'identity' and None set the identity matrix, which gives the usual squared L2-norm. If you just want to exclude certain coefficients, pass a 1d array filled with 1 and 0 for the coefficients to be excluded. Note that P2 must be positive semi-definite. If X is a pandas DataFrame with a categorical dtype and P2 has the same size as the number of columns, the penalty of the categorical column will be applied to all the levels of the categorical. Note that if P2 is two-dimensional, its size needs to be of the same length as the expanded X matrix.
- **fit_intercept** (*bool, optional (default=True)*) – Specifies if a constant (a.k.a. bias or intercept) should be added to the linear predictor (X * coef + intercept).
- **family** (*str or ExponentialDispersionModel, optional (default='normal')*) – The distributional assumption of the GLM, i.e. the loss function to minimize. If a string, one of: 'binomial', 'gamma', 'gaussian', 'inverse.gaussian', 'normal', 'poisson', 'tweedie' or 'negative.binomial'. Note that 'tweedie' sets the power of the Tweedie distribution to 1.5; to use another value, specify it in parentheses (e.g., 'tweedie (1.5)'). The same applies for 'negative.binomial' and theta parameter.
- **link** (*{'auto', 'identity', 'log', 'logit', 'cloglog'} oe Link, optional (default='auto')*) – The link function of the GLM, i.e. mapping from linear predictor (X * coef) to expectation (mu). Option 'auto' sets the link depending on the chosen family as follows:
  - 'identity' for family 'normal'
  - 'log' for families 'poisson', 'gamma', 'inverse.gaussian' and 'negative.binomial'.
  - 'logit' for family 'binomial'
- **solver** (*{'auto', 'irls-cd', 'irls-ls', 'lbfgs', 'trust-constr'}, optional (default='auto')*) – Algorithm to use in the optimization problem:
  - 'auto': 'irls-ls' if l1_ratio is zero and 'irls-cd' otherwise.

  - 'irls-cd': Iteratively reweighted least squares with a coordinate descent inner solver. This can deal with L1 as well as L2 penalties. Note that in order to avoid unnecessary memory duplication of X in the fit method, X should be directly passed as a Fortran-contiguous Numpy array or sparse CSC matrix.
  - 'irls-ls': Iteratively reweighted least squares with a least squares inner solver. This algorithm cannot deal with L1 penalties.
  - 'lbfgs': Scipy's L-BFGS-B optimizer. It cannot deal with L1 penalties.
  - 'trust-constr': Calls scipy.optimize.minimize(method='trust-constr'). It cannot deal with L1 penalties. This solver can optimize problems with inequality constraints, passed via A_ineq and b_ineq. It will be selected automatically when inequality constraints are set and solver='auto'. Note that using this method can lead to significantly increased runtimes by a factor of ten or higher.

- **max_iter** (*int, optional (default=100)*) – The maximal number of iterations for solver algorithms.
- **gradient_tol** (*float, optional (default=None)*) – Stopping criterion. If None, solver-specific defaults will be used. The default value for most solvers is 1e-4, except for 'trust-constr', which requires more conservative convergence settings and has a default value of 1e-8.

  For the IRLS-LS, L-BFGS and trust-constr solvers, the iteration will stop when max{|g_i|, i = 1, ..., n} <= tol, where g_i is the i-th component of the gradient (derivative) of the objective function. For the CD solver, convergence is reached when sum_i(|minimum norm of g_i|), where g_i is the subgradient of the objective and the minimum norm of g_i is the element of the subgradient with the smallest L2 norm.

  If you wish to only use a step-size tolerance, set gradient_tol to a very small number.
- **step_size_tol** (*float, optional (default=None)*) – Alternative stopping criterion. For the IRLS-LS and IRLS-CD solvers, the iteration will stop when the L2 norm of the step size is less than step_size_tol. This stopping criterion is disabled when step_size_tol is None.
- **hessian_approx** (*float, optional (default=0.0)*) – The threshold below which data matrix rows will be ignored for updating the Hessian. See the algorithm documentation for the IRLS algorithm for further details.
- **warm_start** (*bool, optional (default=False)*) – Whether to reuse the solution of the previous call to fit as initialization for coef_ and intercept_ (supersedes start_params). If False or if the attribute coef_ does not exist (first call to fit), start_params sets the start values for coef_ and intercept_.
- **alpha_search** (*bool, optional (default=False)*) – Whether to search along the regularization path for the best alpha. When set to True, alpha should either be None or an iterable. To determine the regularization path, the following sequence is used:

  1. If alpha is an iterable, use it directly. All other parameters governing the regularization path are ignored.
  2. If min_alpha is set, create a path from min_alpha to the lowest alpha such that all coefficients are zero.
  3. If min_alpha_ratio is set, create a path where the ratio of min_alpha / max_alpha = min_alpha_ratio.
  4. If none of the above parameters are set, use a min_alpha_ratio of 1e-6.
- **alphas** (DEPRECATED. Use alpha instead.)

- **n_alphas** (*int, optional (default=100)*) – Number of alphas along the regularization path
- **min_alpha_ratio** (*float, optional (default=None)*) – Length of the path. `min_alpha_ratio=1e-6` means that `min_alpha / max_alpha = 1e-6`. If `None`, `1e-6` is used.
- **min_alpha** (*float, optional (default=None)*) – Minimum alpha to estimate the model with. The grid will then be created over `[max_alpha, min_alpha]`.
- **start_params** (*array-like, shape (n_features*,), optional (default=None)*) – Relevant only if `warm_start` is `False` or if `fit` is called for the first time (so that `self.coef_` does not exist yet). If `None`, all coefficients are set to zero and the start value for the intercept is the weighted average of `y` (If `fit_intercept` is `True`). If an array, used directly as start values; if `fit_intercept` is `True`, its first element is assumed to be the start value for the `intercept_`. Note that `n_features* = X.shape[1] + fit_intercept`, i.e. it includes the intercept.
- **selection** (*str, optional (default='cyclic')*) – For the CD solver 'cd', the coordinates (features) can be updated in either cyclic or random order. If set to `'random'`, a random coefficient is updated every iteration rather than looping over features sequentially in the same order, which often leads to significantly faster convergence, especially when `gradient_tol` is higher than `1e-4`.
- **random_state** (*int or RandomState, optional (default=None)*) – The seed of the pseudo random number generator that selects a random feature to be updated for the CD solver. If an integer, `random_state` is the seed used by the random number generator; if a `RandomState` instance, `random_state` is the random number generator; if `None`, the random number generator is the `RandomState` instance used by `np.random`. Used when `selection` is `'random'`.
- **copy_X** (*bool, optional (default=None)*) – Whether to copy `X`. Since `X` is never modified by *GeneralizedLinearRegressor*, this is unlikely to be needed; this option exists mainly for compatibility with other scikit-learn estimators. If `False`, `X` will not be copied and there will be an error if you pass an `X` in the wrong format, such as providing integer `X` and float `y`. If `None`, `X` will not be copied unless it is in the wrong format.
- **check_input** (*bool, optional (default=True)*) – Whether to bypass several checks on input: `y` values in range of `family`, `sample_weight` non-negative, `P2` positive semi-definite. Don't use this parameter unless you know what you are doing.
- **verbose** (*int, optional (default=0)*) – For the IRLS solver, any positive number will result in a pretty progress bar showing convergence. This features requires having the tqdm package installed. For the L-BFGS and `'trust-constr'` solvers, set `verbose` to any positive number for verbosity.
- **scale_predictors** (*bool, optional (default=False)*) – If `True`, scale all predictors to have standard deviation one. Should be set to `True` if `alpha > 0` and if you want coefficients to be penalized equally.

  Reported coefficient estimates are always at the original scale.

  Advanced developer note: Internally, predictors are always rescaled for computational reasons, but this only affects results if `scale_predictors` is `True`.
- **lower_bounds** (*array-like, shape (n_features,), optional (default=None)*) – Set a lower bound for the coefficients. Setting bounds forces the use of the coordinate descent solver (`'irls-cd'`).

- **upper_bounds** (*array-like, shape=(n_features,), optional (default=None)*) – See lower_bounds.
- **A_ineq** (*array-like, shape=(n_constraints, n_features), optional (default=None)*) – Constraint matrix for linear inequality constraints of the form A_ineq w <= b_ineq. Setting inequality constraints forces the use of the local gradient-based solver 'trust-constr', which may increase runtime significantly. Note that the constraints only apply to coefficients related to features in X. If you want to constrain the intercept, add it to the feature matrix X manually and set fit_intercept==False.
- **b_ineq** (*array-like, shape=(n_constraints,), optional (default=None)*) – Constraint vector for linear inequality constraints of the form A_ineq w <= b_ineq. Refer to the documentation of A_ineq for details.
- **drop_first** (*bool, optional (default = False)*) – If True, drop the first column when encoding categorical variables. Set this to True when alpha=0 and solver='auto' to prevent an error due to a singular feature matrix. In the case of using a formula with interactions, setting this argument to True ensures structural full-rankness (it is equivalent to ensure_full_rank in formulaic and tabmat).
- **robust** (*bool, optional (default = False)*) – If true, then robust standard errors are computed by default.
- **expected_information** (*bool, optional (default = False)*) – If true, then the expected information matrix is computed by default. Only relevant when computing robust standard errors.
- **formula** (*formulaic.FormulaSpec*) – A formula accepted by formulaic. It can either be a one-sided formula, in which case y must be specified in fit, or a two-sided formula, in which case y must be None.
- **interaction_separator** (*str, default=":"*) – The separator between the names of interacted variables.
- **categorical_format** (*str, optional, default='{name}[{category}]'*) – Format string for categorical features. The format string should contain the placeholder {name} for the feature name and {category} for the category name. Only used if X is a pandas DataFrame.
- **cat_missing_method** (*str {'fail'|'zero'|'convert'}, default='fail'*) – How to handle missing values in categorical columns. Only used if X is a pandas data frame. - if 'fail', raise an error if there are missing values - if 'zero', missing values will represent all-zero indicator columns. - if 'convert', missing values will be converted to the cat_missing_name

  category.
- **cat_missing_name** (*str, default='(MISSING)'*) – Name of the category to which missing values will be converted if cat_missing_method='convert'. Only used if X is a pandas data frame.
- **force_all_finite** (*bool*)

coef_

Estimated coefficients for the linear predictor (X*coef_+intercept_) in the GLM.

Type

numpy.array, shape (n_features,)

intercept_

Intercept (a.k.a. bias) added to linear predictor.

Type
float

`n_iter_`

Actual number of iterations used in solver.

Type
int

Notes

The fit itself does not need outcomes to be from an EDM, but only assumes the first two moments to be $\mu_i \equiv \mathrm{E}(y_i) = h(x_i' w)$ and $\mathrm{var}(y_i) = (\phi / s_i) v(\mu_i)$. The unit variance function $v(\mu_i)$ is a property of and given by the specific EDM; see *background*.

The parameters w (`coef_` and `intercept_`) are estimated by minimizing the deviance plus penalty term, which is equivalent to (penalized) maximum likelihood estimation.

If the target `y` is a ratio, appropriate sample weights `s` should be provided. As an example, consider Poisson distributed counts `z` (integers) and weights `s = exposure` (time, money, persons years, ...). Then you fit `y z/s`, i.e. `GeneralizedLinearModel(family='poisson').fit(X, y, sample_weight=s)`. The weights are necessary for the right (finite sample) mean. Consider $\bar{y} = \sum_i s_i y_i / \sum_i s_i$: in this case, one might say that y follows a 'scaled' Poisson distribution. The same holds for other distributions.

References

For the coordinate descent implementation:

- Guo-Xun Yuan, Chia-Hua Ho, Chih-Jen Lin An Improved GLMNET for L1-regularized Logistic Regression, Journal of Machine Learning Research 13 (2012) 1999-2030 https://www.csie.ntu.edu.tw/~cjlin/papers/l1_glmnet/long-glmnet.pdf

`aic`(*X*, *y*, *sample_weight=None*, ***, *context=None*)

Akaike's information criteria. Computed as: $-2 \log \hat{\mathcal{L}} + 2\hat{k}$ where $\hat{\mathcal{L}}$ is the maximum likelihood estimate of the model, and $\hat{k}$ is the effective number of parameters. See *_compute_information_criteria* for more information on the computation of $\hat{k}$.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – Same data as used in 'fit'
- **y** (*array-like, shape (n_samples,)*) – Same data as used in 'fit'
- **sample_weight** (*array-like, shape (n_samples,), optional (default=None)*) – Same data as used in 'fit'
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

`aicc`(*X*, *y*, *sample_weight=None*, ***, *context=None*)

Second-order Akaike's information criteria (or small sample AIC). Computed as: $-2 \log \hat{\mathcal{L}} + 2\hat{k} + \frac{2k(k+1)}{n-k-1}$ where $\hat{\mathcal{L}}$ is the maximum likelihood estimate of the model, n is the number of training instances, and $\hat{k}$

is the effective number of parameters. See *_compute_information_criteria* for more information on the computation of $\hat{k}$.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – Same data as used in 'fit'
- **y** (*array-like, shape (n_samples,)*) – Same data as used in 'fit'
- **sample_weight** (*array-like, shape (n_samples,), optional (default=None)*) – Same data as used in 'fit'
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

bic(*X*, *y*, *sample_weight=None*, ***, *context=None*)

Bayesian information criterion. Computed as: $-2\log\hat{\mathcal{L}} + k\log(n)$ where $\hat{\mathcal{L}}$ is the maximum likelihood estimate of the model, n is the number of training instances, and $\hat{k}$ is the effective number of parameters. See *_compute_information_criteria* for more information on the computation of $\hat{k}$.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – Same data as used in 'fit'
- **y** (*array-like, shape (n_samples,)*) – Same data as used in 'fit'
- **sample_weight** (*array-like, shape (n_samples,), optional (default=None)*) – Same data as used in 'fit'
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

coef_table(*X=None*, *y=None*, *sample_weight=None*, *offset=None*, ***, *confidence_level=0.95*, *mu=None*, *dispersion=None*, *robust=None*, *clusters=None*, *expected_information=None*, *context=None*)

Get a table of of the regression coefficients.

Includes coefficient estimates, standard errors, t-values, p-values and confidence intervals.

Parameters

- **confidence_level** (*float, optional, default=0.95*) – The confidence level for the confidence intervals.
- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features), optional*) – Training data. Can be omitted if a covariance matrix has already been computed or if standard errors, etc. are not desired.
- **y** (*array-like, shape (n_samples,), optional*) – Target values. Can be omitted if a covariance matrix has already been computed.
- **mu** (*array-like, optional, default=None*) – Array with predictions. Estimated if absent.
- **offset** (*array-like, optional, default=None*) – Array with additive offsets.

- **sample_weight** (*array-like, shape (n_samples,), optional, default=None*) – Individual weights for each sample.
- **dispersion** (*float, optional, default=None*) – The dispersion parameter. Estimated if absent.
- **robust** (*boolean, optional, default=None*) – Whether to compute robust standard errors instead of normal ones. If not specified, the model's `robust` attribute is used.
- **clusters** (*array-like, optional, default=None*) – Array with cluster membership. Clustered standard errors are computed if clusters is not None.
- **expected_information** (*boolean, optional, default=None*) – Whether to use the expected or observed information matrix. Only relevant when computing robust standard errors. If not specified, the model's `expected_information` attribute is used.
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

Returns

A table of the regression results.

Return type

pandas.DataFrame

covariance_matrix(*X=None*, *y=None*, *sample_weight=None*, *offset=None*, ***, *mu=None*, *dispersion=None*, *robust=None*, *clusters=None*, *expected_information=None*, *store_covariance_matrix=False*, *skip_checks=False*, *context=None*)

Calculate the covariance matrix for generalized linear models.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features), optional*) – Training data. Can be omitted if a covariance matrix has already been computed.
- **y** (*array-like, shape (n_samples,), optional*) – Target values. Can be omitted if a covariance matrix has already been computed.
- **mu** (*array-like, optional, default=None*) – Array with predictions. Estimated if absent.
- **offset** (*array-like, optional, default=None*) – Array with additive offsets.
- **sample_weight** (*array-like, shape (n_samples,), optional, default=None*) – Individual weights for each sample.
- **dispersion** (*float, optional, default=None*) – The dispersion parameter. Estimated if absent.
- **robust** (*boolean, optional, default=None*) – Whether to compute robust standard errors instead of normal ones. If not specified, the model's `robust` attribute is used.
- **clusters** (*array-like, optional, default=None*) – Array with cluster membership. Clustered standard errors are computed if clusters is not None.
- **expected_information** (*boolean, optional, default=None*) – Whether to use the expected or observed information matrix. Only relevant when computing robust standard errors. If not specified, the model's `expected_information` attribute is used.

- **store_covariance_matrix** (*boolean, optional, default=False*) – Whether to store the covariance matrix in the model instance. If a covariance matrix has already been stored, it will be overwritten.
- **skip_checks** (*boolean, optional, default=False*) – Whether to skip input validation. For internal use only.
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

Notes

We support three types of covariance matrices:

- non-robust
- robust (HC-1)
- clustered

For maximum-likelihood estimator, the covariance matrix takes the form $\mathcal{H}^{-1}(\theta_0)\mathcal{I}(\theta_0)\mathcal{H}^{-1}(\theta_0)$ where $\mathcal{H}^{-1}$ is the inverse Hessian and $\mathcal{I}$ is the Information matrix. The different types of covariance matrices use different approximation of these quantities.

The non-robust covariance matrix is computed as the inverse of the Fisher information matrix. This assumes that the information matrix equality holds.

The robust (HC-1) covariance matrix takes the form $\mathbf{H}^1(\hat{\theta})\mathbf{G}^T(\hat{\theta})\mathbf{G}(\hat{\theta})\mathbf{H}^1(\hat{\theta})$ where $\mathbf{H}$ is the empirical Hessian and $\mathbf{G}$ is the gradient. We apply a finite-sample correction of $\frac{N}{N-p}$.

The clustered covariance matrix uses a similar approach to the robust (HC-1) covariance matrix. However, instead of using $\mathbf{G}^T(\hat{\theta}\mathbf{G}(\hat{\theta})$ directly, we first sum over all the groups first. The finite-sample correction is affected as well, becoming $\frac{M}{M-1}\frac{N}{N-p}$ where M is the number of groups.

References

property family_instance: *ExponentialDispersionModel*

Return an *ExponentialDispersionModel*.

fit(*X*, *y=None*, *sample_weight=None*, *offset=None*, ***, *store_covariance_matrix=False*, *clusters=None*, *weights_sum=None*, *context=None*)

Fit a Generalized Linear Model.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – Training data. Note that a `float32` matrix is acceptable and will result in the entire algorithm being run in 32-bit precision. However, for problems that are poorly conditioned, this might result in poor convergence or flawed parameter estimates. If a Pandas data frame is provided, it may contain categorical columns. In that case, a separate coefficient will be estimated for each category. No category is omitted. This means that some regularization is required to fit models with an intercept or models with several categorical columns.
- **y** (*array-like, shape (n_samples,)*) – Target values.

- **sample_weight** (*array-like, shape (n_samples,), optional (default=None)*) – Individual weights w_i for each sample. Note that, for an Exponential Dispersion Model (EDM), one has $\mathrm{var}(y_i) = \phi \times v(mu)/w_i$. If $y_i \sim EDM(\mu, \phi/w_i)$, then $\sum w_i y_i / \sum w_i \sim EDM(\mu, \phi/\sum w_i)$, i.e. the mean of y is a weighted average with weights equal to `sample_weight`.
- **offset** (*array-like, shape (n_samples,), optional (default=None)*) – Added to linear predictor. An offset of 3 will increase expected `y` by 3 if the link is linear and will multiply expected `y` by 3 if the link is logarithmic.
- **store_covariance_matrix** (*bool, optional (default=False)*) – Whether to estimate and store the covariance matrix of the parameter estimates. If `True`, the covariance matrix will be available in the `covariance_matrix_` attribute after fitting.
- **clusters** (*array-like, optional, default=None*) – Array with cluster membership. Clustered standard errors are computed if clusters is not None.
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.
- **weights_sum** (*float, optional (default=None)*)

Return type

self

`get_formatted_diagnostics`(**, full_report=False, custom_columns=None*)

Get formatted diagnostics which can be printed with report_diagnostics.

Parameters

- **full_report** (*bool, optional (default=False)*) – Print all available information. When `False` and `custom_columns` is `None`, a restricted set of columns is printed out.
- **custom_columns** (*iterable, optional (default=None)*) – Print only the specified columns.

Return type

str | *DataFrame*

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing – A `MetadataRequest` encapsulating routing information.

Return type

MetadataRequest

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params – Parameter names mapped to their values.

Return type
dict

`linear_predictor`(*X*, *offset=None*, ***, *alpha_index=None*, *alpha=None*, *context=None*)

Compute the linear predictor, `X * coef_ + intercept_`.

If `alpha_search` is `True`, but `alpha_index` and `alpha` are both `None`, we use the last alpha value `self._alphas[-1]`.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Observations. `X` may be a pandas data frame with categorical types. If `X` was also a data frame with categorical types during fitting and a category wasn't observed at that point, the corresponding prediction will be `numpy.nan`.
- **offset** (*array-like, shape (n_samples,), optional (default=None)*)
- **alpha_index** (*int or list[int], optional (default=None)*) – Sets the index of the alpha(s) to use in case `alpha_search` is `True`. Incompatible with `alpha` (see below).
- **alpha** (*float or list[float], optional (default=None)*) – Sets the alpha(s) to use in case `alpha_search` is `True`. Incompatible with `alpha_index` (see above).
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

Returns
The linear predictor.

Return type
array, shape (n_samples, n_alphas)

property link_instance: *Link*

Return a *Link*.

`predict`(*X*, *sample_weight=None*, *offset=None*, ***, *alpha_index=None*, *alpha=None*, *context=None*)

Predict using GLM with feature matrix `X`.

If `alpha_search` is `True`, but `alpha_index` and `alpha` are both `None`, we use the last alpha value `self._alphas[-1]`.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Observations. `X` may be a pandas data frame with categorical types. If `X` was also a data frame with categorical types during fitting and a category wasn't observed at that point, the corresponding prediction will be `numpy.nan`.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=None)*) – Sample weights to multiply predictions by.
- **offset** (*array-like, shape (n_samples,), optional (default=None)*)
- **alpha_index** (*int or list[int], optional (default=None)*) – Sets the index of the alpha(s) to use in case `alpha_search` is `True`. Incompatible with `alpha` (see below).
- **alpha** (*float or list[float], optional (default=None)*) – Sets the alpha(s) to use in case `alpha_search` is `True`. Incompatible with `alpha_index` (see above).

- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

Returns
Predicted values times `sample_weight`.

Return type
array, shape (n_samples, n_alphas)

`report_diagnostics`(*, *full_report=False*, *custom_columns=None*)

Print diagnostics to `stdout`.

Parameters

- **full_report** (*bool, optional (default=False)*) – Print all available information. When `False` and `custom_columns` is `None`, a restricted set of columns is printed out.
- **custom_columns** (*iterable, optional (default=None)*) – Print only the specified columns.

Return type
None

`score`(*X*, *y*, *sample_weight=None*, *offset=None*, *, *context=None*)

Compute D^2, the percentage of deviance explained.

D^2 is a generalization of the coefficient of determination R^2. The R^2 uses the squared error and the D^2, the deviance. Note that those two are equal for `family='normal'`.

D^2 is defined as $D^2 = 1 - \frac{D(y_{\mathrm{true}}, y_{\mathrm{pred}})}{D_{\mathrm{null}}}$, D_{null} is the null deviance, i.e. the deviance of a model with intercept alone. The best possible score is one and it can be negative.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like, shape (n_samples,)*) – True values of target.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=None)*) – Sample weights.
- **offset** (*array-like, shape (n_samples,), optional (default=None)*)
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

Returns
D^2 of self.predict(X) w.r.t. y.

Return type
float

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
`**params` (*`dict`*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

`std_errors`(*X=None*, *y=None*, *sample_weight=None*, *offset=None*, ***, *mu=None*, *dispersion=None*, *robust=None*, *clusters=None*, *expected_information=None*, *store_covariance_matrix=False*, *context=None*)

Calculate standard errors for generalized linear models.

See *covariance_matrix* for an in-depth explanation of how the standard errors are computed.

Parameters

- **`X`** (*`{array-like, sparse matrix}, shape (n_samples, n_features), optional`*) – Training data. Can be omitted if a covariance matrix has already been computed.
- **`y`** (*`array-like, shape (n_samples,), optional`*) – Target values. Can be omitted if a covariance matrix has already been computed.
- **`sample_weight`** (*`array-like, shape (n_samples,), optional, default=None`*) – Individual weights for each sample.
- **`offset`** (*`array-like, optional, default=None`*) – Array with additive offsets.
- **`mu`** (*`array-like, optional, default=None`*) – Array with predictions. Estimated if absent.
- **`dispersion`** (*`float, optional, default=None`*) – The dispersion parameter. Estimated if absent.
- **`robust`** (*`boolean, optional, default=None`*) – Whether to compute robust standard errors instead of normal ones. If not specified, the model's `robust` attribute is used.
- **`clusters`** (*`array-like, optional, default=None`*) – Array with cluster membership. Clustered standard errors are computed if clusters is not None.
- **`expected_information`** (*`boolean, optional, default=None`*) – Whether to use the expected or observed information matrix. Only relevant when computing robust standard errors. If not specified, the model's `expected_information` attribute is used.
- **`store_covariance_matrix`** (*`boolean, optional, default=False`*) – Whether to store the covariance matrix in the model instance. If a covariance matrix has already been stored, it will be overwritten.
- **`context`** (*`Optional[Union[int, Mapping[str, Any]]], default=None`*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

`wald_test`(*X=None*, *y=None*, *sample_weight=None*, *offset=None*, ***, *R=None*, *features=None*, *terms=None*, *formula=None*, *r=None*, *mu=None*, *dispersion=None*, *robust=None*, *clusters=None*, *expected_information=None*, *context=None*)

Compute the Wald test statistic and p-value for a linear hypothesis.

The left hand side of the hypothesis may be specified in the following ways:

- `R`: The restriction matrix representing the linear combination of coefficients to test.
- `features`: The name of a feature or a list of features to test.
- `terms`: The name of a term or a list of terms to test.
- `formula`: A formula string specifying the hypothesis to test.

The right hand side of the tested hypothesis is specified by `r`. In the case of a `terms`-based test, the null hypothesis is that each coefficient relating to a term equals the corresponding value in `r`.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features), optional*) – Training data. Can be omitted if a covariance matrix has already been computed.
- **y** (*array-like, shape (n_samples,), optional*) – Target values. Can be omitted if a covariance matrix has already been computed.
- **sample_weight** (*array-like, shape (n_samples,), optional, default=None*) – Individual weights for each sample.
- **offset** (*array-like, optional, default=None*) – Array with additive offsets.
- **R** (*np.ndarray, optional, default=None*) – The restriction matrix representing the linear combination of coefficients to test.
- **features** (*Union[str, list[str]], optional, default=None*) – The name of a feature or a list of features to test.
- **terms** (*Union[str, list[str]], optional, default=None*) – The name of a term or a list of terms to test. It can cover one or more coefficients. In the case of a model based on a formula, a term is one of the expressions separated by + signs. Otherwise, a term is one column in the input data. As categorical variables need not be one-hot encoded in glum, in their case, the hypothesis to be tested is that the coefficients of all categories are equal to `r`.
- **r** (*np.ndarray, optional, default=None*) – The vector representing the values of the linear combination. If None, the test is for whether the linear combinations of the coefficients are zero.
- **mu** (*array-like, optional, default=None*) – Array with predictions. Estimated if absent.
- **dispersion** (*float, optional, default=None*) – The dispersion parameter. Estimated if absent.
- **robust** (*boolean, optional, default=None*) – Whether to compute robust standard errors instead of normal ones. If not specified, the model's `robust` attribute is used.
- **clusters** (*array-like, optional, default=None*) – Array with cluster membership. Clustered standard errors are computed if clusters is not None.
- **expected_information** (*boolean, optional, default=None*) – Whether to use the expected or observed information matrix. Only relevant when computing robust standard errors. If not specified, the model's `expected_information` attribute is used.

- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.
- **formula** (*str | None*)

Returns
NamedTuple with test statistic, p-value, and degrees of freedom.

Return type
WaldTestResult

`class glum.GeneralizedLinearRegressorCV`(**, l1_ratio=0, P1='identity', P2='identity', fit_intercept=True, family='normal', link='auto', solver='auto', max_iter=100, gradient_tol=None, step_size_tol=None, hessian_approx=0.0, warm_start=False, n_alphas=100, alphas=None, min_alpha_ratio=None, min_alpha=None, start_params=None, selection='cyclic', random_state=None, copy_X=True, check_input=True, verbose=0, scale_predictors=False, lower_bounds=None, upper_bounds=None, A_ineq=None, b_ineq=None, force_all_finite=True, cv=None, n_jobs=None, drop_first=False, robust=True, expected_information=False, formula=None, interaction_separator=':', categorical_format='{name}[{category}]', cat_missing_method='fail', cat_missing_name='(MISSING)'*)

Bases: `GeneralizedLinearRegressorBase`

Generalized linear model with iterative fitting along a regularization path.

The best model is selected by cross-validation.

Cross-validated regression via a Generalized Linear Model (GLM) with penalties. For more on GLMs and on these parameters, see the documentation for *GeneralizedLinearRegressor*. CV conventions follow `sklearn.linear_model.LassoCV`.

Parameters

- **l1_ratio** (*float or array of floats, optional (default=0)*) – If you pass `l1_ratio` as an array, the `fit` method will choose the best value of `l1_ratio` and store it as `self.l1_ratio`.
- **P1** (*{'identity', array-like, None}, shape (n_features,), optional (default='identity')*) – This array controls the strength of the regularization for each coefficient independently. A high value will lead to higher regularization while a value of zero will remove the regularization on this parameter. Note that `n_features = X.shape[1]`. If `X` is a pandas DataFrame with a categorical dtype and P1 has the same size as the number of columns, the penalty of the categorical column will be applied to all the levels of the categorical.
- **P2** (*{'identity', array-like, sparse matrix, None}, shape (n_features,) or (n_features, n_features), optional (default='identity')*) – With this option, you can set the P2 matrix in the L2 penalty `w*P2*w`. This gives a fine control over this penalty (Tikhonov regularization). A 2d array is directly used as the square matrix P2. A 1d array is interpreted as diagonal (square) matrix. The default `'identity'` and `None` set the identity matrix, which gives the usual squared L2-norm. If you just want to exclude certain coefficients, pass a 1d array filled with 1 and 0 for the coefficients to be excluded. Note that P2 must be positive semi-definite. If `X` is a pandas DataFrame with a

categorical dtype and P2 has the same size as the number of columns, the penalty of the categorical column will be applied to all the levels of the categorical. Note that if P2 is two-dimensional, its size needs to be of the same length as the expanded `X` matrix.

- **`fit_intercept`** (*`bool, optional (default=True)`*) – Specifies if a constant (a.k.a. bias or intercept) should be added to the linear predictor (`X * coef + intercept`).
- **`family`** (*`str or` `ExponentialDispersionModel`, `optional (default='normal')`*) – The distributional assumption of the GLM, i.e. the loss function to minimize. If a string, one of: `'binomial'`, `'gamma'`, `'gaussian'`, `'inverse.gaussian'`, `'normal'`, `'poisson'`, `'tweedie'` or `'negative.binomial'`. Note that `'tweedie'` sets the power of the Tweedie distribution to 1.5; to use another value, specify it in parentheses (e.g., `'tweedie (1.5)'`). The same applies for `'negative.binomial'` and theta parameter.
- **`link`** (*`{'auto', 'identity', 'log', 'logit', 'cloglog'},` `Link` `or None, optional (default='auto')`*) – The link function of the GLM, i.e. mapping from linear predictor (`X * coef`) to expectation (`mu`). Option `'auto'` sets the link depending on the chosen family as follows:
  - `'identity'` for family `'normal'`
  - `'log'` for families `'poisson'`, `'gamma'`, `'inverse.gaussian'` and `'negative.binomial'`.
  - `'logit'` for family `'binomial'`
- **`solver`** (*`{'auto', 'irls-cd', 'irls-ls', 'lbfgs', 'trust-constr'}, optional (default='auto')`*) – Algorithm to use in the optimization problem:
  - `'auto'`: `'irls-ls'` if `l1_ratio` is zero and `'irls-cd'` otherwise.
  - `'irls-cd'`: Iteratively reweighted least squares with a coordinate descent inner solver. This can deal with L1 as well as L2 penalties. Note that in order to avoid unnecessary memory duplication of X in the `fit` method, `X` should be directly passed as a Fortran-contiguous Numpy array or sparse CSC matrix.
  - `'irls-ls'`: Iteratively reweighted least squares with a least squares inner solver. This algorithm cannot deal with L1 penalties.
  - `'lbfgs'`: Scipy's L-BFGS-B optimizer. It cannot deal with L1 penalties.
- **`max_iter`** (*`int, optional (default=100)`*) – The maximal number of iterations for solver algorithms.
- **`gradient_tol`** (*`float, optional (default=None)`*) – Stopping criterion. If `None`, solver-specific defaults will be used. The default value for most solvers is `1e-4`, except for `'trust-constr'`, which requires more conservative convergence settings and has a default value of `1e-8`.

  For the IRLS-LS, L-BFGS and trust-constr solvers, the iteration will stop when `max{|g_i|, i = 1, ..., n} <= tol`, where `g_i` is the `i`-th component of the gradient (derivative) of the objective function. For the CD solver, convergence is reached when `sum_i(|minimum norm of g_i|)`, where `g_i` is the subgradient of the objective and the minimum norm of `g_i` is the element of the subgradient with the smallest L2 norm.

  If you wish to only use a step-size tolerance, set `gradient_tol` to a very small number.
- **`step_size_tol`** (*`float, optional (default=None)`*) – Alternative stopping criterion. For the IRLS-LS and IRLS-CD solvers, the iteration will stop when the L2 norm of the step size is less than `step_size_tol`. This stopping criterion is disabled when `step_size_tol` is `None`.

- **hessian_approx** (*float, optional (default=0.0)*) – The threshold below which data matrix rows will be ignored for updating the Hessian. See the algorithm documentation for the IRLS algorithm for further details.
- **warm_start** (*bool, optional (default=False)*) – Whether to reuse the solution of the previous call to fit as initialization for coef_ and intercept_ (supersedes start_params). If False or if the attribute coef_ does not exist (first call to fit), start_params sets the start values for coef_ and intercept_.
- **n_alphas** (*int, optional (default=100)*) – Number of alphas along the regularization path
- **alphas** (*array-like, optional (default=None)*) – List of alphas for which to compute the models. If None, the alphas are set automatically. Setting None is preferred.
- **min_alpha_ratio** (*float, optional (default=None)*) – Length of the path. min_alpha_ratio=1e-6 means that min_alpha / max_alpha = 1e-6. If None, 1e-6 is used.
- **min_alpha** (*float, optional (default=None)*) – Minimum alpha to estimate the model with. The grid will then be created over [max_alpha, min_alpha].
- **start_params** (*array-like, shape (n_features*,), optional (default=None)*) – Relevant only if warm_start is False or if fit is called for the first time (so that self.coef_ does not exist yet). If None, all coefficients are set to zero and the start value for the intercept is the weighted average of y (If fit_intercept is True). If an array, used directly as start values; if fit_intercept is True, its first element is assumed to be the start value for the intercept_. Note that n_features* = X.shape[1] + fit_intercept, i.e. it includes the intercept.
- **selection** (*str, optional (default='cyclic')*) – For the CD solver 'cd', the coordinates (features) can be updated in either cyclic or random order. If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially in the same order, which often leads to significantly faster convergence, especially when gradient_tol is higher than 1e-4.
- **random_state** (*int or RandomState, optional (default=None)*) – The seed of the pseudo random number generator that selects a random feature to be updated for the CD solver. If an integer, random_state is the seed used by the random number generator; if a RandomState instance, random_state is the random number generator; if None, the random number generator is the RandomState instance used by np.random. Used when selection is 'random'.
- **copy_X** (*bool, optional (default=None)*) – Whether to copy X. Since X is never modified by *GeneralizedLinearRegressor*, this is unlikely to be needed; this option exists mainly for compatibility with other scikit-learn estimators. If False, X will not be copied and there will be an error if you pass an X in the wrong format, such as providing integer X and float y. If None, X will not be copied unless it is in the wrong format.
- **check_input** (*bool, optional (default=True)*) – Whether to bypass several checks on input: y values in range of family, sample_weight non-negative, P2 positive semi-definite. Don't use this parameter unless you know what you are doing.
- **verbose** (*int, optional (default=0)*) – For the IRLS solver, any positive number will result in a pretty progress bar showing convergence. This features requires having the tqdm package installed. For the L-BFGS solver, set verbose to any positive number for verbosity.
- **scale_predictors** (*bool, optional (default=False)*) – If True, estimate a scaled model where all predictors have a standard deviation of 1. This can result in better estimates if predictors are on very different scales (for example, centimeters and kilometers).

Advanced developer note: Internally, predictors are always rescaled for computational reasons, but this only affects results if `scale_predictors` is `True`.

- **lower_bounds** (*array-like, shape (n_features), optional (default=None)*) – Set a lower bound for the coefficients. Setting bounds forces the use of the coordinate descent solver (`'irls-cd'`).
- **upper_bounds** (*array-like, shape=(n_features), optional (default=None)*) – See `lower_bounds`.
- **A_ineq** (*array-like, shape=(n_constraints, n_features), optional (default=None)*) – Constraint matrix for linear inequality constraints of the form `A_ineq w <= b_ineq`.
- **b_ineq** (*array-like, shape=(n_constraints,), optional (default=None)*) – Constraint vector for linear inequality constraints of the form `A_ineq w <= b_ineq`.
- **cv** (*int, cross-validation generator or Iterable, optional (default=None)*) – Determines the cross-validation splitting strategy. One of:
  - `None`, to use the default 5-fold cross-validation,
  - `int`, to specify the number of folds.
  - `Iterable` yielding (train, test) splits as arrays of indices.

  For integer/`None` inputs, `KFold` is used
- **n_jobs** (*int, optional (default=None)*) – The maximum number of concurrently running jobs. The number of jobs that are needed is `len(l1_ratio)` x `n_folds`. `-1` is the same as the number of CPU on your machine. `None` means `1` unless in a `joblib.parallel_backend` context.
- **drop_first** (*bool, optional (default = False)*) – If `True`, drop the first column when encoding categorical variables. Set this to True when alpha=0 and solver='auto' to prevent an error due to a singular feature matrix. In the case of using a formula with interactions, setting this argument to `True` ensures structural full-rankness (it is equivalent to `ensure_full_rank` in formulaic and tabmat).
- **formula** (*FormulaSpec*) – A formula accepted by formulaic. It can either be a one-sided formula, in which case `y` must be specified in `fit`, or a two-sided formula, in which case `y` must be `None`.
- **interaction_separator** (*str, default=":"*) – The separator between the names of interacted variables.
- **categorical_format** (*str, default="{name}[T.{category}]"*) – The format string used to generate the names of categorical variables. Has to include the placeholders `{name}` and `{category}`. Only used if `formula` is not `None`.
- **force_all_finite** (*bool*)
- **robust** (*bool*)
- **expected_information** (*bool*)
- **cat_missing_method** (*str*)
- **cat_missing_name** (*str*)

`alpha_`

The amount of regularization chosen by cross validation.

Type
float

`alphas_`

Alphas used by the model.

Type
array, shape (n_l1_ratios, n_alphas)

`l1_ratio_`

The compromise between L1 and L2 regularization chosen by cross validation.

Type
float

`coef_`

Estimated coefficients for the linear predictor in the GLM at the optimal (`l1_ratio_`, `alpha_`).

Type
array, shape (n_features,)

`intercept_`

Intercept (a.k.a. bias) added to linear predictor.

Type
float

`n_iter_`

The number of iterations run by the CD solver to reach the specified tolerance for the optimal alpha.

Type
int

`coef_path_`

Estimated coefficients for the linear predictor in the GLM at every point along the regularization path.

Type
array, shape (n_folds, n_l1_ratios, n_alphas, n_features)

`deviance_path_`

Deviance for the test set on each fold, varying alpha.

Type
array, shape(n_folds, n_alphas)

`robust`

If true, then robust standard errors are computed by default.

Type
bool, optional (default = False)

`expected_information`

If true, then the expected information matrix is computed by default. Only relevant when computing robust standard errors.

Type
bool, optional (default = False)

`categorical_format`

Format string for categorical features. The format string should contain the placeholder `{name}` for the feature name and `{category}` for the category name. Only used if `X` is a pandas DataFrame.

Type
str, optional (default = "{name}[{category}]")

`cat_missing_method`

How to handle missing values in categorical columns. Only used if `X` is a pandas data frame. - if 'fail', raise an error if there are missing values - if 'zero', missing values will represent all-zero indicator columns. - if 'convert', missing values will be converted to the `cat_missing_name`

category.

Type
str {'fail'|'zero'|'convert'}, default='fail'

`cat_missing_name`

Name of the category to which missing values will be converted if `cat_missing_method='convert'`. Only used if `X` is a pandas data frame.

Type
str, default='(MISSING)'

`coef_table`(*X=None*, *y=None*, *sample_weight=None*, *offset=None*, ***, *confidence_level=0.95*, *mu=None*, *dispersion=None*, *robust=None*, *clusters=None*, *expected_information=None*, *context=None*)

Get a table of of the regression coefficients.

Includes coefficient estimates, standard errors, t-values, p-values and confidence intervals.

Parameters

- **confidence_level** (*float, optional, default=0.95*) – The confidence level for the confidence intervals.
- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features), optional*) – Training data. Can be omitted if a covariance matrix has already been computed or if standard errors, etc. are not desired.
- **y** (*array-like, shape (n_samples,), optional*) – Target values. Can be omitted if a covariance matrix has already been computed.
- **mu** (*array-like, optional, default=None*) – Array with predictions. Estimated if absent.
- **offset** (*array-like, optional, default=None*) – Array with additive offsets.
- **sample_weight** (*array-like, shape (n_samples,), optional, default=None*) – Individual weights for each sample.
- **dispersion** (*float, optional, default=None*) – The dispersion parameter. Estimated if absent.
- **robust** (*boolean, optional, default=None*) – Whether to compute robust standard errors instead of normal ones. If not specified, the model's `robust` attribute is used.
- **clusters** (*array-like, optional, default=None*) – Array with cluster membership. Clustered standard errors are computed if clusters is not None.
- **expected_information** (*boolean, optional, default=None*) – Whether to use the expected or observed information matrix. Only relevant when computing robust standard errors. If not specified, the model's `expected_information` attribute is used.
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If

an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

Returns
A table of the regression results.

Return type
pandas.DataFrame

`covariance_matrix`(*X=None*, *y=None*, *sample_weight=None*, *offset=None*, ***, *mu=None*, *dispersion=None*, *robust=None*, *clusters=None*, *expected_information=None*, *store_covariance_matrix=False*, *skip_checks=False*, *context=None*)

Calculate the covariance matrix for generalized linear models.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features), optional*) – Training data. Can be omitted if a covariance matrix has already been computed.
- **y** (*array-like, shape (n_samples,), optional*) – Target values. Can be omitted if a covariance matrix has already been computed.
- **mu** (*array-like, optional, default=None*) – Array with predictions. Estimated if absent.
- **offset** (*array-like, optional, default=None*) – Array with additive offsets.
- **sample_weight** (*array-like, shape (n_samples,), optional, default=None*) – Individual weights for each sample.
- **dispersion** (*float, optional, default=None*) – The dispersion parameter. Estimated if absent.
- **robust** (*boolean, optional, default=None*) – Whether to compute robust standard errors instead of normal ones. If not specified, the model's `robust` attribute is used.
- **clusters** (*array-like, optional, default=None*) – Array with cluster membership. Clustered standard errors are computed if clusters is not None.
- **expected_information** (*boolean, optional, default=None*) – Whether to use the expected or observed information matrix. Only relevant when computing robust standard errors. If not specified, the model's `expected_information` attribute is used.
- **store_covariance_matrix** (*boolean, optional, default=False*) – Whether to store the covariance matrix in the model instance. If a covariance matrix has already been stored, it will be overwritten.
- **skip_checks** (*boolean, optional, default=False*) – Whether to skip input validation. For internal use only.
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

Notes

We support three types of covariance matrices:

- non-robust
- robust (HC-1)
- clustered

For maximum-likelihood estimator, the covariance matrix takes the form $\mathcal{H}^{-1}(\theta_0)\mathcal{I}(\theta_0)\mathcal{H}^{-1}(\theta_0)$ where $\mathcal{H}^{-1}$ is the inverse Hessian and $\mathcal{I}$ is the Information matrix. The different types of covariance matrices use different approximation of these quantities.

The non-robust covariance matrix is computed as the inverse of the Fisher information matrix. This assumes that the information matrix equality holds.

The robust (HC-1) covariance matrix takes the form $\mathbf{H}^1(\hat{\theta})\mathbf{G}^T(\hat{\theta})\mathbf{G}(\hat{\theta})\mathbf{H}^1(\hat{\theta})$ where $\mathbf{H}$ is the empirical Hessian and $\mathbf{G}$ is the gradient. We apply a finite-sample correction of $\frac{N}{N-p}$.

The clustered covariance matrix uses a similar approach to the robust (HC-1) covariance matrix. However, instead of using $\mathbf{G}^T(\hat{\theta}\mathbf{G}(\hat{\theta})$ directly, we first sum over all the groups first. The finite-sample correction is affected as well, becoming $\frac{M}{M-1}\frac{N}{N-p}$ where M is the number of groups.

References

property family_instance: *ExponentialDispersionModel*

Return an *ExponentialDispersionModel*.

fit(*X*, *y*, *sample_weight=None*, *offset=None*, ***, *store_covariance_matrix=False*, *clusters=None*, *context=None*)

Choose the best model along a 'regularization path' by cross-validation.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – Training data. Note that a `float32` matrix is acceptable and will result in the entire algorithm being run in 32-bit precision. However, for problems that are poorly conditioned, this might result in poor convergence or flawed parameter estimates. If a Pandas data frame is provided, it may contain categorical columns. In that case, a separate coefficient will be estimated for each category. No category is omitted. This means that some regularization is required to fit models with an intercept or models with several categorical columns.
- **y** (*array-like, shape (n_samples,)*) – Target values.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=None)*) – Individual weights w_i for each sample. Note that, for an Exponential Dispersion Model (EDM), one has $\mathrm{var}(y_i) = \phi \times v(mu)/w_i$. If $y_i \sim EDM(\mu, \phi/w_i)$, then $\sum w_i y_i / \sum w_i \sim EDM(\mu, \phi/\sum w_i)$, i.e. the mean of y is a weighted average with weights equal to `sample_weight`.
- **offset** (*array-like, shape (n_samples,), optional (default=None)*) – Added to linear predictor. An offset of 3 will increase expected `y` by 3 if the link is linear and will multiply expected `y` by 3 if the link is logarithmic.
- **store_covariance_matrix** (*bool, optional (default=False)*) – Whether to store the covariance matrix of the parameter estimates corresponding to the best best model.
- **clusters** (*array-like, optional, default=None*) – Array with cluster membership. Clustered standard errors are computed if clusters is not None.

- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

get_formatted_diagnostics(**, full_report=False, custom_columns=None*)

Get formatted diagnostics which can be printed with report_diagnostics.

Parameters

- **full_report** (*bool, optional (default=False)*) – Print all available information. When `False` and `custom_columns` is `None`, a restricted set of columns is printed out.
- **custom_columns** (*iterable, optional (default=None)*) – Print only the specified columns.

Return type

str | *DataFrame*

get_metadata_routing()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing – A `MetadataRequest` encapsulating routing information.

Return type

MetadataRequest

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params – Parameter names mapped to their values.

Return type

dict

linear_predictor(*X, offset=None, *, alpha_index=None, alpha=None, context=None*)

Compute the linear predictor, `X * coef_ + intercept_`.

If `alpha_search` is `True`, but `alpha_index` and `alpha` are both `None`, we use the last alpha value `self._alphas[-1]`.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Observations. `X` may be a pandas data frame with categorical types. If `X` was also a data frame with categorical types during fitting and a category wasn't observed at that point, the corresponding prediction will be `numpy.nan`.
- **offset** (*array-like, shape (n_samples,), optional (default=None)*)
- **alpha_index** (*int or list[int], optional (default=None)*) – Sets the index of the alpha(s) to use in case `alpha_search` is `True`. Incompatible with `alpha` (see below).

- **alpha** (*float or list[float], optional (default=None)*) – Sets the alpha(s) to use in case alpha_search is True. Incompatible with alpha_index (see above).
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set context=0 to make the calling scope available.

Returns

The linear predictor.

Return type

array, shape (n_samples, n_alphas)

property link_instance: *Link*

Return a *Link*.

predict(*X*, *sample_weight=None*, *offset=None*, ***, *alpha_index=None*, *alpha=None*, *context=None*)

Predict using GLM with feature matrix X.

If alpha_search is True, but alpha_index and alpha are both None, we use the last alpha value self._alphas[-1].

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Observations. X may be a pandas data frame with categorical types. If X was also a data frame with categorical types during fitting and a category wasn't observed at that point, the corresponding prediction will be numpy.nan.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=None)*) – Sample weights to multiply predictions by.
- **offset** (*array-like, shape (n_samples,), optional (default=None)*)
- **alpha_index** (*int or list[int], optional (default=None)*) – Sets the index of the alpha(s) to use in case alpha_search is True. Incompatible with alpha (see below).
- **alpha** (*float or list[float], optional (default=None)*) – Sets the alpha(s) to use in case alpha_search is True. Incompatible with alpha_index (see above).
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set context=0 to make the calling scope available.

Returns

Predicted values times sample_weight.

Return type

array, shape (n_samples, n_alphas)

report_diagnostics(***, *full_report=False*, *custom_columns=None*)

Print diagnostics to stdout.

Parameters

- **full_report** (*bool, optional (default=False)*) – Print all available information. When False and custom_columns is None, a restricted set of columns is printed out.

- **custom_columns** (*iterable, optional (default=None)*) – Print only the specified columns.

Return type
None

score(*X*, *y*, *sample_weight=None*, *offset=None*, ***, *context=None*)

Compute D^2, the percentage of deviance explained.

D^2 is a generalization of the coefficient of determination R^2. The R^2 uses the squared error and the D^2, the deviance. Note that those two are equal for `family='normal'`.

D^2 is defined as $D^2 = 1 - \frac{D(y_{\mathrm{true}}, y_{\mathrm{pred}})}{D_{\mathrm{null}}}$, D_{null} is the null deviance, i.e. the deviance of a model with intercept alone. The best possible score is one and it can be negative.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like, shape (n_samples,)*) – True values of target.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=None)*) – Sample weights.
- **offset** (*array-like, shape (n_samples,), optional (default=None)*)
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

Returns
D^2 of self.predict(X) w.r.t. y.

Return type
float

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

std_errors(*X=None*, *y=None*, *sample_weight=None*, *offset=None*, ***, *mu=None*, *dispersion=None*, *robust=None*, *clusters=None*, *expected_information=None*, *store_covariance_matrix=False*, *context=None*)

Calculate standard errors for generalized linear models.

See *covariance_matrix* for an in-depth explanation of how the standard errors are computed.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features), optional*) – Training data. Can be omitted if a covariance matrix has already been computed.
- **y** (*array-like, shape (n_samples,), optional*) – Target values. Can be omitted if a covariance matrix has already been computed.
- **sample_weight** (*array-like, shape (n_samples,), optional, default=None*) – Individual weights for each sample.
- **offset** (*array-like, optional, default=None*) – Array with additive offsets.
- **mu** (*array-like, optional, default=None*) – Array with predictions. Estimated if absent.
- **dispersion** (*float, optional, default=None*) – The dispersion parameter. Estimated if absent.
- **robust** (*boolean, optional, default=None*) – Whether to compute robust standard errors instead of normal ones. If not specified, the model's `robust` attribute is used.
- **clusters** (*array-like, optional, default=None*) – Array with cluster membership. Clustered standard errors are computed if clusters is not None.
- **expected_information** (*boolean, optional, default=None*) – Whether to use the expected or observed information matrix. Only relevant when computing robust standard errors. If not specified, the model's `expected_information` attribute is used.
- **store_covariance_matrix** (*boolean, optional, default=False*) – Whether to store the covariance matrix in the model instance. If a covariance matrix has already been stored, it will be overwritten.
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.

wald_test(*X=None*, *y=None*, *sample_weight=None*, *offset=None*, ***, *R=None*, *features=None*, *terms=None*, *formula=None*, *r=None*, *mu=None*, *dispersion=None*, *robust=None*, *clusters=None*, *expected_information=None*, *context=None*)

Compute the Wald test statistic and p-value for a linear hypothesis.

The left hand side of the hypothesis may be specified in the following ways:

- `R`: The restriction matrix representing the linear combination of coefficients to test.
- `features`: The name of a feature or a list of features to test.
- `terms`: The name of a term or a list of terms to test.
- `formula`: A formula string specifying the hypothesis to test.

The right hand side of the tested hypothesis is specified by `r`. In the case of a `terms`-based test, the null hypothesis is that each coefficient relating to a term equals the corresponding value in `r`.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features), optional*) – Training data. Can be omitted if a covariance matrix has already been computed.
- **y** (*array-like, shape (n_samples,), optional*) – Target values. Can be omitted if a covariance matrix has already been computed.

- **sample_weight** (*array-like, shape (n_samples,), optional, default=None*) – Individual weights for each sample.
- **offset** (*array-like, optional, default=None*) – Array with additive offsets.
- **R** (*np.ndarray, optional, default=None*) – The restriction matrix representing the linear combination of coefficients to test.
- **features** (*Union[str, list[str]], optional, default=None*) – The name of a feature or a list of features to test.
- **terms** (*Union[str, list[str]], optional, default=None*) – The name of a term or a list of terms to test. It can cover one or more coefficients. In the case of a model based on a formula, a term is one of the expressions separated by + signs. Otherwise, a term is one column in the input data. As categorical variables need not be one-hot encoded in glum, in their case, the hypothesis to be tested is that the coefficients of all categories are equal to `r`.
- **r** (*np.ndarray, optional, default=None*) – The vector representing the values of the linear combination. If None, the test is for whether the linear combinations of the coefficients are zero.
- **mu** (*array-like, optional, default=None*) – Array with predictions. Estimated if absent.
- **dispersion** (*float, optional, default=None*) – The dispersion parameter. Estimated if absent.
- **robust** (*boolean, optional, default=None*) – Whether to compute robust standard errors instead of normal ones. If not specified, the model's `robust` attribute is used.
- **clusters** (*array-like, optional, default=None*) – Array with cluster membership. Clustered standard errors are computed if clusters is not None.
- **expected_information** (*boolean, optional, default=None*) – Whether to use the expected or observed information matrix. Only relevant when computing robust standard errors. If not specified, the model's `expected_information` attribute is used.
- **context** (*Optional[Union[int, Mapping[str, Any]]], default=None*) – The context to add to the evaluation context of the formula with, e.g., custom transforms. If an integer, the context is taken from the stack frame of the caller at the given depth. Otherwise, a mapping from variable names to values is expected. By default, no context is added. Set `context=0` to make the calling scope available.
- **formula** (*str | None*)

Returns

NamedTuple with test statistic, p-value, and degrees of freedom.

Return type

WaldTestResult

`class glum.`**`IdentityLink`**

Bases: *Link*

The identity link function.

`derivative``(`*mu*`)`

Compute the derivative of the link function.

Parameters

mu (*array-like, shape (n_samples,)*) – Usually the (predicted) mean.

`inverse`(*lin_pred*)

Compute the inverse link function.

The inverse link function `h` gives the inverse relationship between the linear predictor, `X * w`, and the mean, `mu  E(Y)`, so that `h(X * w) = mu`.

Parameters

`lin_pred` (*`array-like, shape (n_samples,)`*) – Usually the (fitted) linear predictor.

`inverse_derivative`(*lin_pred*)

Compute the derivative of the inverse link function.

Parameters

`lin_pred` (*`array-like, shape (n_samples,)`*) – Usually the (fitted) linear predictor.

`inverse_derivative2`(*lin_pred*)

Compute second derivative of the inverse link function.

Parameters

`lin_pred` (*`array-like, shape (n_samples,)`*) – Usually the (fitted) linear predictor.

`link`(*mu*)

Compute the link function.

The link function `g` links the mean, `mu  E(Y)`, to the linear predictor, `X * w`, so that `g(mu)` is equal to the linear predictor.

Parameters

`mu` (*`array-like, shape (n_samples,)`*) – Usually the (predicted) mean.

`to_tweedie`(*safe=True*)

Return the Tweedie representation of a link function if it exists.

`class glum.InverseGaussianDistribution`

Bases: *`ExponentialDispersionModel`*

Class for the inverse Gaussian distribution.

The inverse Gaussian distribution models outcomes `y` in `(0, +∞)`.

See the documentation of the superclass, *`ExponentialDispersionModel`*, for details.

`deviance`(*y*, *mu*, *sample_weight=None*)

Compute the deviance.

The deviance is a weighted sum of the unit deviances. In terms of the unit log likelihood ℓ, it equals $2\sum_i[\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **`y`** (*`array-like, shape (n_samples,)`*) – Target values.
- **`mu`** (*`array-like, shape (n_samples,)`*) – Predicted mean.
- **`sample_weight`** (*`array-like, shape (n_samples,), optional (default=1)`*) – Weights or exposure to which the variance is inversely proportional.

Return type

float

deviance_derivative(*y*, *mu*, *sample_weight=1*)

Compute the derivative of the deviance with respect to `mu`.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,) (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

dispersion(*y*, *mu*, *sample_weight=None*, *ddof=1*, *method='pearson'*)

Estimate the dispersion parameter ϕ.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inversely proportional.
- **ddof** (*int, optional (default=1)*) – Degrees of freedom consumed by the model for `mu`.
- **{'pearson'** (*method =*) – Whether to base the estimate on the Pearson residuals or the deviance.
- **'deviance'}** – Whether to base the estimate on the Pearson residuals or the deviance.
- **(default='pearson')** (*optional*) – Whether to base the estimate on the Pearson residuals or the deviance.

Return type

float

eta_mu_deviance(*link*, *factor*, *cur_eta*, *X_dot_d*, *y*, *sample_weight*)

Compute `eta`, `mu` and the deviance.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The linear predictor, `eta`, as `cur_eta + factor * X_dot_d`.
- *numpy.ndarray, shape (X.shape[0],)* – The link-function-transformed prediction, `mu`.
- *float* – The deviance.

Parameters

- **link** (`Link`)
- **factor** (*float*)

Return type

tuple[*ndarray*, *ndarray*, float]

in_y_range(*x*)

Return `True` if `x` is in the valid range of the EDM.

Return type
ndarray

`log_likelihood`(*y*, *mu*, *sample_weight=None*, *dispersion=None*)

Compute the log likelihood.

Parameters

- `y` (*array-like, shape (n_samples,)*) – Target values.
- `mu` (*array-like, shape (n_samples,)*) – Predicted mean.
- `sample_weight` (*array-like, shape (n_samples,), optional (default=1)*) – Sample weights.
- `dispersion` (*float, optional (default=None)*) – Dispersion parameter ϕ. Estimated if `None`.

Return type
float

`rowwise_gradient_hessian`(*link*, *coef*, *dispersion*, *X*, *y*, *sample_weight*, *eta*, *mu*, *offset=None*)

Compute the gradient and negative Hessian of the log likelihood row-wise.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The gradient of the log likelihood, row-wise.
- *numpy.ndarray, shape (X.shape[0],)* – The negative Hessian of the log likelihood, row-wise.

Parameters

- `link` (`Link`)
- `X` (*MatrixBase | StandardizedMatrix*)

`to_tweedie`(*safe=True*)

Return the Tweedie representation of a distribution if it exists.

`unit_deviance`(*y*, *mu*)

Compute the unit deviance.

In terms of the unit log likelihood ℓ, the unit deviance is $2[\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- `y` (*array-like, shape (n_samples,)*) – Target values.
- `mu` (*array-like, shape (n_samples,)*) – Predicted mean.

`unit_deviance_derivative`(*y*, *mu*)

Compute the derivative of the unit deviance with respect to `mu`.

The derivative of the unit deviance is given by $2 \times (\mu - y)/v(\mu)$, where $v(\mu)$ is the unit variance.

Parameters

- `y` (*array-like, shape (n_samples,)*) – Target values.
- `mu` (*array-like, shape (n_samples,)*) – Predicted mean.

Return type
array-like, shape (n_samples,)

unit_variance(*mu*)

Compute the unit variance.

The unit variance, $v(\mu) \equiv b''((b')^{-1}(\mu))$, determines the variance as a function of the mean μ by $\mathrm{var}(y_i) = v(\mu_i) \times \phi / w_i$. It can also be derived from the unit deviance $d(y, \mu)$ as

$$v(\mu) = 2 \div \frac{\partial^2 d(y, \mu)}{\partial \mu^2}\big|_{y=\mu}.$$

See also `variance()`.

Parameters

mu (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

ndarray

unit_variance_derivative(*mu*)

Compute the derivative of the unit variance with respect to `mu`.

Parameters

mu (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

ndarray

variance(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the variance function.

The variance of $Y_i \sim \mathrm{EDM}(\mu_i, \phi / w_i)$ takes the form $v(\mu_i) \times \phi / w_i$, where $v(\mu)$ is the unit variance and w_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

variance_derivative(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the derivative of the variance with respect to `mu`.

The derivative of the variance is equal to $v(\mu_i) \times \phi / w_i$, where $v(\mu)$ is the unit variance and ws_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

`class glum.`**`Link`**

Bases: `object`

Abstract base class for link functions.

`abstract` **`derivative`**`(`*mu*`)`

Compute the derivative of the link function.

Parameters
`mu` (*`array-like, shape (n_samples,)`*) – Usually the (predicted) mean.

`abstract` **`inverse`**`(`*lin_pred*`)`

Compute the inverse link function.

The inverse link function `h` gives the inverse relationship between the linear predictor, `X * w`, and the mean, `mu  E(Y)`, so that `h(X * w) = mu`.

Parameters
`lin_pred` (*`array-like, shape (n_samples,)`*) – Usually the (fitted) linear predictor.

`abstract` **`inverse_derivative`**`(`*lin_pred*`)`

Compute the derivative of the inverse link function.

Parameters
`lin_pred` (*`array-like, shape (n_samples,)`*) – Usually the (fitted) linear predictor.

`abstract` **`inverse_derivative2`**`(`*lin_pred*`)`

Compute second derivative of the inverse link function.

Parameters
`lin_pred` (*`array-like, shape (n_samples,)`*) – Usually the (fitted) linear predictor.

`abstract` **`link`**`(`*mu*`)`

Compute the link function.

The link function `g` links the mean, `mu  E(Y)`, to the linear predictor, `X * w`, so that `g(mu)` is equal to the linear predictor.

Parameters
`mu` (*`array-like, shape (n_samples,)`*) – Usually the (predicted) mean.

`to_tweedie``(`*safe=True*`)`

Return the Tweedie representation of a link function if it exists.

`class glum.`**`LogLink`**

Bases: *`Link`*

The log link function `log(x)`.

`derivative``(`*mu*`)`

Compute the derivative of the link function.

Parameters
`mu` (*`array-like, shape (n_samples,)`*) – Usually the (predicted) mean.

`inverse``(`*lin_pred*`)`

Compute the inverse link function.

The inverse link function `h` gives the inverse relationship between the linear predictor, `X * w`, and the mean, `mu  E(Y)`, so that `h(X * w) = mu`.

Parameters
`lin_pred` (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

`inverse_derivative`(*lin_pred*)

Compute the derivative of the inverse link function.

Parameters
`lin_pred` (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

`inverse_derivative2`(*lin_pred*)

Compute second derivative of the inverse link function.

Parameters
`lin_pred` (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

`link`(*mu*)

Compute the link function.

The link function `g` links the mean, `mu  E(Y)`, to the linear predictor, `X * w`, so that `g(mu)` is equal to the linear predictor.

Parameters
`mu` (*array-like, shape (n_samples,)*) – Usually the (predicted) mean.

`to_tweedie`(*safe=True*)

Return the Tweedie representation of a link function if it exists.

`class glum.LogitLink`

Bases: *Link*

The logit link function `logit(x)`.

`derivative`(*mu*)

Compute the derivative of the link function.

Parameters
`mu` (*array-like, shape (n_samples,)*) – Usually the (predicted) mean.

`inverse`(*lin_pred*)

Compute the inverse link function.

The inverse link function `h` gives the inverse relationship between the linear predictor, `X * w`, and the mean, `mu  E(Y)`, so that `h(X * w) = mu`.

Parameters
`lin_pred` (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

`inverse_derivative`(*lin_pred*)

Compute the derivative of the inverse link function.

Parameters
`lin_pred` (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

`inverse_derivative2`(*lin_pred*)

Compute second derivative of the inverse link function.

Parameters
`lin_pred` (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

link(*mu*)

Compute the link function.

The link function `g` links the mean, `mu` `E(Y)`, to the linear predictor, `X * w`, so that `g(mu)` is equal to the linear predictor.

Parameters

mu (*array-like, shape (n_samples,)*) – Usually the (predicted) mean.

to_tweedie(*safe=True*)

Return the Tweedie representation of a link function if it exists.

class glum.**NegativeBinomialDistribution**(*theta=1.0*)

Bases: *ExponentialDispersionModel*

A class for the Negative Binomial distribution.

A negative binomial distribution with mean $\mu = \mathrm{E}(Y)$ is uniquely defined by its mean-variance relationship $\mathrm{var}(Y) \propto \mu + \theta * \mu^2$.

Parameters

theta (*float, optional (default=1.0)*) – The dispersion parameter from the `unit_variance` $v(\mu) = \mu + \theta * \mu^2$. For $\theta <= 0$, no distribution exists.

References

For the log-likelihood and deviance:

- M. L. Zwilling Negative Binomial Regression, The Mathematica Journal 2013. https://www.mathematica-journal.com/2013/06/27/negative-binomial-regression/

deviance(*y, mu, sample_weight=None*)

Compute the deviance.

The deviance is a weighted sum of the unit deviances. In terms of the unit log likelihood ℓ, it equals $2 \sum_i [\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which the variance is inversely proportional.

Return type

float

deviance_derivative(*y, mu, sample_weight=1*)

Compute the derivative of the deviance with respect to `mu`.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,) (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

`dispersion`(*y*, *mu*, *sample_weight=None*, *ddof=1*, *method='pearson'*)

Estimate the dispersion parameter ϕ.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inversely proportional.
- **ddof** (*int, optional (default=1)*) – Degrees of freedom consumed by the model for `mu`.
- **{'pearson'** (*method =*) – Whether to base the estimate on the Pearson residuals or the deviance.
- **'deviance'}** – Whether to base the estimate on the Pearson residuals or the deviance.
- **(default='pearson')** (*optional*) – Whether to base the estimate on the Pearson residuals or the deviance.

Return type
float

`eta_mu_deviance`(*link*, *factor*, *cur_eta*, *X_dot_d*, *y*, *sample_weight*)

Compute `eta`, `mu` and the deviance.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The linear predictor, `eta`, as `cur_eta + factor * X_dot_d`.
- *numpy.ndarray, shape (X.shape[0],)* – The link-function-transformed prediction, `mu`.
- *float* – The deviance.

Parameters

- **link** (`Link`)
- **factor** (*float*)

Return type
tuple[*ndarray*, *ndarray*, float]

`in_y_range`(*x*)

Return `True` if `x` is in the valid range of the EDM.

Return type
ndarray

`log_likelihood`(*y*, *mu*, *sample_weight=None*, *dispersion=1*)

Compute the log likelihood.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Sample weights.
- **dispersion** (*float, optional (default=1.0)*) – Ignored.

Return type

float

rowwise_gradient_hessian(*link*, *coef*, *dispersion*, *X*, *y*, *sample_weight*, *eta*, *mu*, *offset=None*)

Compute the gradient and negative Hessian of the log likelihood row-wise.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The gradient of the log likelihood, row-wise.
- *numpy.ndarray, shape (X.shape[0],)* – The negative Hessian of the log likelihood, row-wise.

Parameters

- **link** (Link)
- **X** (*MatrixBase | StandardizedMatrix*)

property theta: float

Return the negative binomial theta parameter.

to_tweedie(*safe=True*)

Return the Tweedie representation of a distribution if it exists.

unit_deviance(*y*, *mu*)

Compute the unit deviance.

In terms of the unit log likelihood ℓ, the unit deviance is $2[\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

unit_deviance_derivative(*y*, *mu*)

Compute the derivative of the unit deviance with respect to `mu`.

The derivative of the unit deviance is given by $2 \times (\mu - y) / v(\mu)$, where $v(\mu)$ is the unit variance.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

array-like, shape (n_samples,)

unit_variance(*mu*)

Compute the unit variance.

The unit variance, $v(\mu) \equiv b''((b')^{-1}(\mu))$, determines the variance as a function of the mean μ by $\mathrm{var}(y_i) = v(\mu_i) \times \phi / w_i$. It can also be derived from the unit deviance $d(y, \mu)$ as

$$v(\mu) = 2 \div \frac{\partial^2 d(y, \mu)}{\partial \mu^2} \big|_{y=\mu}.$$

See also *variance()*.

Parameters
`mu` (*array-like, shape (n_samples,)*) – Predicted mean.

`unit_variance_derivative`(*mu*)

Compute the derivative of the unit variance with respect to `mu`.

Parameters
`mu` (*array-like, shape (n_samples,)*) – Predicted mean.

`variance`(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the variance function.

The variance of $Y_i \sim \mathrm{EDM}(\mu_i, \phi/w_i)$ takes the form $v(\mu_i) \times \phi/w_i$, where $v(\mu)$ is the unit variance and w_i are weights.

Parameters

- `mu` (*array-like, shape (n_samples,)*) – Predicted mean.
- `dispersion` (*float, optional (default=1)*) – Dispersion parameter ϕ.
- `sample_weight` (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

`variance_derivative`(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the derivative of the variance with respect to `mu`.

The derivative of the variance is equal to $v(\mu_i) \times \phi/w_i$, where $v(\mu)$ is the unit variance and ws_i are weights.

Parameters

- `mu` (*array-like, shape (n_samples,)*) – Predicted mean.
- `dispersion` (*float, optional (default=1)*) – Dispersion parameter ϕ.
- `sample_weight` (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

`class glum.NormalDistribution`

Bases: `ExponentialDispersionModel`

Class for the normal (a.k.a. Gaussian) distribution.

The normal distribution models outcomes `y` in `(-∞, +∞)`.

See the documentation of the superclass, `ExponentialDispersionModel`, for details.

`deviance`(*y*, *mu*, *sample_weight=None*)

Compute the deviance.

The deviance is a weighted sum of the unit deviances. In terms of the unit log likelihood ℓ, it equals $2 \sum_i [\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- `y` (*array-like, shape (n_samples,)*) – Target values.
- `mu` (*array-like, shape (n_samples,)*) – Predicted mean.

- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which the variance is inversely proportional.

Return type
float

deviance_derivative(*y*, *mu*, *sample_weight=1*)

Compute the derivative of the deviance with respect to `mu`.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,) (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

dispersion(*y*, *mu*, *sample_weight=None*, *ddof=1*, *method='pearson'*)

Estimate the dispersion parameter ϕ.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inversely proportional.
- **ddof** (*int, optional (default=1)*) – Degrees of freedom consumed by the model for `mu`.
- **{'pearson'** (*method =*) – Whether to base the estimate on the Pearson residuals or the deviance.
- **'deviance'}** – Whether to base the estimate on the Pearson residuals or the deviance.
- **(default='pearson')** (*optional*) – Whether to base the estimate on the Pearson residuals or the deviance.

Return type
float

eta_mu_deviance(*link*, *factor*, *cur_eta*, *X_dot_d*, *y*, *sample_weight*)

Compute `eta`, `mu` and the deviance.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The linear predictor, `eta`, as `cur_eta + factor * X_dot_d`.
- *numpy.ndarray, shape (X.shape[0],)* – The link-function-transformed prediction, `mu`.
- *float* – The deviance.

Parameters

- **link** (`Link`)
- **factor** (*float*)

Return type
tuple[*ndarray*, *ndarray*, float]

`in_y_range`(*x*)

Return `True` if `x` is in the valid range of the EDM.

Return type
ndarray

`log_likelihood`(*y*, *mu*, *sample_weight=None*, *dispersion=None*)

Compute the log likelihood.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Sample weights.
- **dispersion** (*float, optional (default=None)*) – Dispersion parameter ϕ. Estimated if `None`.

Return type
float

`rowwise_gradient_hessian`(*link*, *coef*, *dispersion*, *X*, *y*, *sample_weight*, *eta*, *mu*, *offset=None*)

Compute the gradient and negative Hessian of the log likelihood row-wise.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The gradient of the log likelihood, row-wise.
- *numpy.ndarray, shape (X.shape[0],)* – The negative Hessian of the log likelihood, row-wise.

Parameters

- **link** (`Link`)
- **X** (*MatrixBase | StandardizedMatrix*)

`to_tweedie`(*safe=True*)

Return the Tweedie representation of a distribution if it exists.

`unit_deviance`(*y*, *mu*)

Compute the unit deviance.

In terms of the unit log likelihood ℓ, the unit deviance is $2[\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

`unit_deviance_derivative`(*y*, *mu*)

Compute the derivative of the unit deviance with respect to `mu`.

The derivative of the unit deviance is given by $2 \times (\mu - y) / v(\mu)$, where $v(\mu)$ is the unit variance.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

Return type
array-like, shape (n_samples,)

unit_variance(*mu*)

Compute the unit variance.

The unit variance, $v(\mu) \equiv b''((b')^{-1}(\mu))$, determines the variance as a function of the mean μ by $\mathrm{var}(y_i) = v(\mu_i) \times \phi / w_i$. It can also be derived from the unit deviance $d(y, \mu)$ as

$$v(\mu) = 2 \div \frac{\partial^2 d(y, \mu)}{\partial \mu^2}\big|_{y=\mu}.$$

See also `variance()`.

Parameters
mu (*array-like, shape (n_samples,)*) – Predicted mean.

Return type
ndarray

unit_variance_derivative(*mu*)

Compute the derivative of the unit variance with respect to `mu`.

Parameters
mu (*array-like, shape (n_samples,)*) – Predicted mean.

Return type
ndarray

variance(*mu, dispersion=1, sample_weight=1*)

Compute the variance function.

The variance of $Y_i \sim \mathrm{EDM}(\mu_i, \phi / w_i)$ takes the form $v(\mu_i) \times \phi / w_i$, where $v(\mu)$ is the unit variance and w_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

variance_derivative(*mu, dispersion=1, sample_weight=1*)

Compute the derivative of the variance with respect to `mu`.

The derivative of the variance is equal to $v(\mu_i) \times \phi / w_i$, where $v(\mu)$ is the unit variance and ws_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

class glum.**PoissonDistribution**

Bases: *ExponentialDispersionModel*

Class for the scaled Poisson distribution.

The Poisson distribution models discrete outcomes y in `[0, +∞)`.

See the documentation of the superclass, *ExponentialDispersionModel*, for details.

deviance(*y*, *mu*, *sample_weight=None*)

Compute the deviance.

The deviance is a weighted sum of the unit deviances. In terms of the unit log likelihood ℓ, it equals $2 \sum_i [\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which the variance is inversely proportional.

Return type

float

deviance_derivative(*y*, *mu*, *sample_weight=1*)

Compute the derivative of the deviance with respect to `mu`.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,) (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

dispersion(*y*, *mu*, *sample_weight=None*, *ddof=1*, *method='pearson'*)

Estimate the dispersion parameter ϕ.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inversely proportional.
- **ddof** (*int, optional (default=1)*) – Degrees of freedom consumed by the model for `mu`.
- **{'pearson'** (*method =*) – Whether to base the estimate on the Pearson residuals or the deviance.
- **'deviance'}** – Whether to base the estimate on the Pearson residuals or the deviance.
- **(default='pearson')** (*optional*) – Whether to base the estimate on the Pearson residuals or the deviance.

Return type
float

eta_mu_deviance(*link*, *factor*, *cur_eta*, *X_dot_d*, *y*, *sample_weight*)

Compute `eta`, `mu` and the deviance.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The linear predictor, `eta`, as `cur_eta + factor * X_dot_d`.
- *numpy.ndarray, shape (X.shape[0],)* – The link-function-transformed prediction, `mu`.
- *float* – The deviance.

Parameters

- **link** (Link)
- **factor** (*float*)

Return type
tuple[*ndarray*, *ndarray*, float]

in_y_range(*x*)

Return `True` if `x` is in the valid range of the EDM.

Return type
ndarray

log_likelihood(*y*, *mu*, *sample_weight=None*, *dispersion=None*)

Compute the log likelihood.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Sample weights.
- **dispersion** (*float, optional (default=None)*) – Dispersion parameter ϕ. Estimated if `None`.

Return type
float

rowwise_gradient_hessian(*link*, *coef*, *dispersion*, *X*, *y*, *sample_weight*, *eta*, *mu*, *offset=None*)

Compute the gradient and negative Hessian of the log likelihood row-wise.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The gradient of the log likelihood, row-wise.
- *numpy.ndarray, shape (X.shape[0],)* – The negative Hessian of the log likelihood, row-wise.

Parameters

- **link** (Link)
- **X** (*MatrixBase | StandardizedMatrix*)

`to_tweedie`(*safe=True*)

Return the Tweedie representation of a distribution if it exists.

`unit_deviance`(*y*, *mu*)

Compute the unit deviance.

`unit_deviance_derivative`(*y*, *mu*)

Compute the derivative of the unit deviance with respect to `mu`.

The derivative of the unit deviance is given by $2 \times (\mu - y)/v(\mu)$, where $v(\mu)$ is the unit variance.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

array-like, shape (n_samples,)

`unit_variance`(*mu*)

Compute the unit variance.

The unit variance, $v(\mu) \equiv b''((b')^{-1}(\mu))$, determines the variance as a function of the mean μ by $\mathrm{var}(y_i) = v(\mu_i) \times \phi / w_i$. It can also be derived from the unit deviance $d(y, \mu)$ as

$$v(\mu) = 2 \div \frac{\partial^2 d(y,\mu)}{\partial \mu^2}\big|_{y=\mu}.$$

See also `variance()`.

Parameters

mu (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

ndarray

`unit_variance_derivative`(*mu*)

Compute the derivative of the unit variance with respect to `mu`.

Parameters

mu (*array-like, shape (n_samples,)*) – Predicted mean.

Return type

ndarray

`variance`(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the variance function.

The variance of $Y_i \sim \mathrm{EDM}(\mu_i, \phi/w_i)$ takes the form $v(\mu_i) \times \phi/w_i$, where $v(\mu)$ is the unit variance and w_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

variance_derivative(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the derivative of the variance with respect to `mu`.

The derivative of the variance is equal to $v(\mu_i) \times \phi / w_i$, where $v(\mu)$ is the unit variance and ws_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type

array-like, shape (n_samples,)

class glum.**TweedieDistribution**(*power=0*)

Bases: *ExponentialDispersionModel*

A class for the Tweedie distribution.

A Tweedie distribution with mean $\mu = \mathrm{E}(Y)$ is uniquely defined by its mean-variance relationship $\mathrm{var}(Y) \propto \mu^{\mathrm{p}}$.

Special cases are:

Power	Distribution	Support
0	Normal	(-∞, +∞)
1	Poisson	[0, +∞)
(1, 2)	Compound Poisson	[0, +∞)
2	Gamma	(0, +∞)
3	Inverse Gaussian	(0, +∞)

See the documentation of the superclass, *ExponentialDispersionModel*, for details.

Parameters

power (*float, optional (default=0)*) – The variance power of the `unit_variance` $v(\mu) = \mu^{\mathrm{power}}$. For $0 < \mathrm{power} < 1$, no distribution exists.

deviance(*y*, *mu*, *sample_weight=None*)

Compute the deviance.

The deviance is a weighted sum of the unit deviances. In terms of the unit log likelihood ℓ, it equals $2 \sum_i [\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which the variance is inversely proportional.

Return type

float

deviance_derivative(*y*, *mu*, *sample_weight=1*)

Compute the derivative of the deviance with respect to `mu`.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,) (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

`dispersion`(*y*, *mu*, *sample_weight=None*, *ddof=1*, *method='pearson'*)

Estimate the dispersion parameter ϕ.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inversely proportional.
- **ddof** (*int, optional (default=1)*) – Degrees of freedom consumed by the model for `mu`.
- **{'pearson'** (*method =*) – Whether to base the estimate on the Pearson residuals or the deviance.
- **'deviance'}** – Whether to base the estimate on the Pearson residuals or the deviance.
- **(default='pearson')** (*optional*) – Whether to base the estimate on the Pearson residuals or the deviance.

Return type
float

`eta_mu_deviance`(*link*, *factor*, *cur_eta*, *X_dot_d*, *y*, *sample_weight*)

Compute `eta`, `mu` and the deviance.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The linear predictor, `eta`, as `cur_eta + factor * X_dot_d`.
- *numpy.ndarray, shape (X.shape[0],)* – The link-function-transformed prediction, `mu`.
- *float* – The deviance.

Parameters

- **link** (`Link`)
- **factor** (*float*)

Return type
tuple[*ndarray*, *ndarray*, float]

`in_y_range`(*x*)

Return `True` if `x` is in the valid range of the EDM.

Return type
ndarray

`property include_lower_bound: bool`

Return whether `lower_bound` is allowed as a value of `y`.

`log_likelihood`(*y*, *mu*, *sample_weight=None*, *dispersion=None*)

Compute the log likelihood.

For `1 < p < 2`, we use the series approximation by Dunn and Smyth (2005) to compute the normalization term.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Sample weights.
- **dispersion** (*float, optional (default=None)*) – Dispersion parameter ϕ. Estimated if `None`.

Return type

float

property lower_bound: float

Get the lower bound of values for the EDM.

property power: float

Return the Tweedie power parameter.

`rowwise_gradient_hessian`(*link*, *coef*, *dispersion*, *X*, *y*, *sample_weight*, *eta*, *mu*, *offset=None*)

Compute the gradient and negative Hessian of the log likelihood row-wise.

Returns

- *numpy.ndarray, shape (X.shape[0],)* – The gradient of the log likelihood, row-wise.
- *numpy.ndarray, shape (X.shape[0],)* – The negative Hessian of the log likelihood, row-wise.

Parameters

- **link** (`Link`)
- **X** (*MatrixBase | StandardizedMatrix*)

`to_tweedie`(*safe=True*)

Return the Tweedie representation of a distribution if it exists.

`unit_deviance`(*y*, *mu*)

Compute the unit deviance.

In terms of the unit log likelihood ℓ, the unit deviance is $2[\ell(y_i, y_i, \phi) - \ell(y_i, \mu, \phi)]$, i.e. twice the difference between the log likelihood of a saturated model (with one parameter per observation) and the model at hand.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

`unit_deviance_derivative`(*y*, *mu*)

Compute the derivative of the unit deviance with respect to `mu`.

The derivative of the unit deviance is given by $2 \times (\mu - y)/v(\mu)$, where $v(\mu)$ is the unit variance.

Parameters

- **y** (*array-like, shape (n_samples,)*) – Target values.
- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.

Return type
array-like, shape (n_samples,)

unit_variance(*mu*)

Compute the unit variance.

The unit variance, $v(\mu) \equiv b''((b')^{-1}(\mu))$, determines the variance as a function of the mean μ by $\mathrm{var}(y_i) = v(\mu_i) \times \phi / w_i$. It can also be derived from the unit deviance $d(y, \mu)$ as

$$v(\mu) = 2 \div \frac{\partial^2 d(y, \mu)}{\partial \mu^2} \big|_{y=\mu}.$$

See also `variance()`.

Parameters
mu (*array-like, shape (n_samples,)*) – Predicted mean.

unit_variance_derivative(*mu*)

Compute the derivative of the unit variance with respect to `mu`.

Parameters
mu (*array-like, shape (n_samples,)*) – Predicted mean.

variance(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the variance function.

The variance of $Y_i \sim \mathrm{EDM}(\mu_i, \phi / w_i)$ takes the form $v(\mu_i) \times \phi / w_i$, where $v(\mu)$ is the unit variance and w_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

variance_derivative(*mu*, *dispersion=1*, *sample_weight=1*)

Compute the derivative of the variance with respect to `mu`.

The derivative of the variance is equal to $v(\mu_i) \times \phi / w_i$, where $v(\mu)$ is the unit variance and ws_i are weights.

Parameters

- **mu** (*array-like, shape (n_samples,)*) – Predicted mean.
- **dispersion** (*float, optional (default=1)*) – Dispersion parameter ϕ.
- **sample_weight** (*array-like, shape (n_samples,), optional (default=1)*) – Weights or exposure to which variance is inverse proportional.

Return type
array-like, shape (n_samples,)

class glum.**TweedieLink**(*power*)

Bases: *Link*

The Tweedie link function `x^(1-p)` if `p1` and `log(x)` if `p=1`.

See the documentation of the superclass, *Link*, for details.

derivative(*mu*)

Compute the derivative of the link function.

Parameters
mu (*array-like, shape (n_samples,)*) – Usually the (predicted) mean.

inverse(***kwargs*)

Compute the inverse link function.

The inverse link function `h` gives the inverse relationship between the linear predictor, `X * w`, and the mean, `mu  E(Y)`, so that `h(X * w) = mu`.

Parameters
lin_pred (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

inverse_derivative(***kwargs*)

Compute the derivative of the inverse link function.

Parameters
lin_pred (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

inverse_derivative2(***kwargs*)

Compute second derivative of the inverse link function.

Parameters
lin_pred (*array-like, shape (n_samples,)*) – Usually the (fitted) linear predictor.

link(*mu*)

Compute the link function.

The link function `g` links the mean, `mu  E(Y)`, to the linear predictor, `X * w`, so that `g(mu)` is equal to the linear predictor.

Parameters
mu (*array-like, shape (n_samples,)*) – Usually the (predicted) mean.

to_tweedie(*safe=True*)

Return the Tweedie representation of a link function if it exists.

glum.**get_link**(*link*, *family*)

For the Tweedie distribution, this code follows actuarial best practices regarding link functions. Note that these links are sometimes not canonical:

- identity for normal (`p = 0`);
- no convention for `p < 0`, so let's leave it as identity;
- log otherwise.

Parameters

- **link** (*str | Link*)
- **family** (ExponentialDispersionModel)

Return type

Link

CHAPTER

NINE

CHANGELOG

9.1 3.0.0 - 2024-04-27

Breaking changes:

- All arguments to GeneralizedLinearRegressorBase, *GeneralizedLinearRegressor* and GeneralizedLinearRegressorCV are now keyword-only.
- All arguments to public methods of GeneralizedLinearRegressorBase, *GeneralizedLinearRegressor* or GeneralizedLinearRegressorCV except X, y, sample_weight and offset are now keyword-only.
- *GeneralizedLinearRegressor*'s default value for alpha is now 0, i.e. no regularization.
- *GammaDistribution*, *InverseGaussianDistribution*, *NormalDistribution* and *PoissonDistribution* no longer inherit from *TweedieDistribution*.
- The power parameter of *TweedieLink* has been renamed from p to power, in line with *TweedieDistribution*.
- *TweedieLink* no longer instantiates *IdentityLink* or *LogLink* for power=0 and power=1, respectively. On the other hand, *TweedieLink* is now compatible with power=0 and power=1.

New features:

- Added a formula interface for specifying models.
- Improved feature name handling. Feature names are now created for non-pandas input matrices too. Furthermore, the format of categorical features can be specified by the user.
- Term names are now stored in the model's attributes. This is useful for categorical features, where they refer to the whole variable, not just single levels.
- Added more options for treating missing values in categorical columns. They can either raise a ValueError ("fail"), be treated as all-zero indicators ("zero") or represented as a new category ("convert").
- *meth:GeneralizedLinearRegressor.wald_test* can now perform tests based on a formula string and term names.
- *InverseGaussianDistribution* gains a *log_likelihood()* method.

9.2 2.7.0 - 2024-02-19

Bug fix:

- Added cython compiler directive legacy_implicit_noexcept = True to fix performance regression with cython 3.

Other changes:

- Require Python>=3.9 in line with *NEP 29 <https://numpy.org/neps/nep-0029-deprecation_policy.html#support-table>*.
- Build and test with Python 3.12 in CI.
- Added line search stopping criterion for tiny loss improvements based on gradient information.
- Added warnings about breaking changes in future versions.

9.3 2.6.0 - 2023-09-05

New features:

- Added the complementary log-log (`cloglog`) link function.
- Added the option to store the covariance matrix after estimating it. In this case, the covariance matrix does not have to be recomputed when calling inference methods.
- Added methods for performing Wald tests based on a restriction matrix, feature names or term names.
- Added a method for creating a coefficient table with confidence intervals and p-values.

Bug fix:

- Fixed `covariance_matrix()` mutating feature names when called with a data frame. See here.

Other changes:

- When computing the covariance matrix, check whether the design matrix is ill-conditioned for all types of input. Furthermore, do it in a more efficient way.
- Pin `tabmat<4.0.0` (the new release will bring breaking changes).

9.4 2.5.2 - 2023-06-02

Bug fix

- Fix the `glm_benchmarks_analyze` command line tool. See here.
- Fixed a bug in *`GeneralizedLinearRegressor`* when fit on a data set with a constant column and `warm_start=True`. See here.

Other changes:

- Remove dev dependency on `dask_ml`.
- We now pin `llvm-openmp=11` when creating the wheel for macOS in line with what scikit-learn does.

9.5 2.5.1 - 2023-05-19

Bug fix:

- We fixed a bug in the computation of `log_likelihood()`. Previously, this method just returned `None`.

9.6 2.5.0 - 2023-04-28

New feature:

- Added Negative Binomial distribution by setting the `'family'` parameter of *`GeneralizedLinearRegressor`* and *`GeneralizedLinearRegressorCV`* to `'negative.binomial'`.

9.7 2.4.1 - 2023-03-14

Bug fixes:

- Fixed an issue with `_score_matrix()` which failed when called with a tabmat matrix input.

Other changes:

- Removes unused scikit-learn cython imports.

9.8 2.4.0 - 2023-01-31

Other changes:

- *`LogitLink`* has been made public.
- Apple Silicon wheels are now uploaded to PyPI.

9.9 2.3.0 - 2023-01-06

Bug fixes:

- A data frame with dense and sparse columns was transformed to a dense matrix instead of a split matrix by `_set_up_and_check_fit_args()`. Fixed by calling `tabmat.from_pandas` on any data frame.

New features:

- The following classes and functions have been made public: *`BinomialDistribution`*, *`ExponentialDispersionModel`*, *`GammaDistribution`*, *`GeneralizedHyperbolicSecant`*, *`InverseGaussianDistribution`*, *`NormalDistribution`*, *`PoissonDistribution`*, *`IdentityLink`*, *`Link`*, *`LogLink`*, *`TweedieLink`*, `get_family()` and `get_link()`.
- The distribution and link classes now feature a more lenient equality check instead of the default identity check, so that, e.g., `TweedieDistribution(1) == TweedieDistribution(1)` now returns `True`.

9.10 2.2.1 - 2022-11-25

Other changes:

- Fixing pypi upload issue. Version 2.2.0 will not be available through the standard distribution channels.

9.11 2.2.0 - 2022-11-25

New features:

- Add an argument to GeneralizedLinearRegressorBase to drop the first category in a Categorical column using [implementation in tabmat](https://github.com/Quantco/tabmat/pull/168)
- One may now request the Tweedie loss by setting the `'family'` parameter of *GeneralizedLinearRegressor* and *GeneralizedLinearRegressorCV* to `'tweedie'`.

Bug fixes:

- Setting bounds for constant columns was not working (bounds were internally modified to 0). A similar issue was preventing inequalities from working with constant columns. This is now fixed.

Other changes:

- No more builds for 32-bit systems with python >= 3.8. This is due to scipy not supporting it anymore.

9.12 2.1.2 - 2022-07-01

Other changes:

- Next attempt to build wheel for PyPI without `--march=native`.

9.13 2.1.1 - 2022-07-01

Other changes:

- We are now building the wheel for PyPI without `--march=native` to make it more portable across architectures.

9.14 2.1.0 - 2022-06-27

New features:

- Added `aic()`, `aicc()` and `bic()` attributes to the *GeneralizedLinearRegressor*. These attributes provide the information criteria based on the training data and the effective degrees of freedom of the maximum likelihood estimate for the model's parameters.
- *std_errors()* and *covariance_matrix()* of *GeneralizedLinearRegressor* now accept data frames with categorical data.

Bug fixes:

- The `score()` method of *GeneralizedLinearRegressor* and *GeneralizedLinearRegressorCV* now accepts offsets.

- Fixed the calculation of the information matrix for the Binomial distribution with logit link, which affected nonrobust standard errors.

Other:

- The CI now runs daily unit tests against the nightly builds of numpy, pandas and scikit-learn.
- The minimally required version of tabmat is now 3.1.0.

9.15 2.0.3 - 2021-11-05

Other:

- We are now specifying the run time dependencies in `setup.py`, so that missing dependencies are automatically installed from PyPI when installing `glum` via pip.

9.16 2.0.2 - 2021-11-03

Bug fix:

- Fixed the sign of the log likelihood of the Gaussian distribution (not used for fitting coefficients).
- Fixed the wide benchmarks which had duplicated columns (categorical and numerical).

Other:

- The CI now builds the wheels and upload to pypi with every new release.
- Renamed functions checking for qc.matrix compliance to refer to tabmat.

9.17 2.0.1 - 2021-10-11

Bug fix:

- Fixed pyproject.toml. We now support installing through pip and pep517.

9.18 2.0.0 - 2021-10-08

Breaking changes:

- Renamed the package to `glum`!! Hurray! Celebration.
- *`GeneralizedLinearRegressor`* and *`GeneralizedLinearRegressorCV`* lose the `fit_dispersion` parameter. Please use the `dispersion()` method of the appropriate family instance instead.
- All functions now use `sample_weight` as a keyword instead of `weights`, in line with scikit-learn.
- All functions now use `dispersion` as a keyword instead of `phi`.
- Several methods *`GeneralizedLinearRegressor`* and *`GeneralizedLinearRegressorCV`* that should have been private have had an underscore prefixed on their names: `tear_down_from_fit()`, `_set_up_for_fit()`, `_set_up_and_check_fit_args()`, `_get_start_coef()`, `_solve()` and `_solve_regularization_path()`.

- *glum.GeneralizedLinearRegressor.report_diagnostics()* and *glum.GeneralizedLinearRegressor.get_formatted_diagnostics()* are now public.

New features:

- P1 and P2 now accepts 1d array with the same number of elements as the unexpanded design matrix. In this case, the penalty associated with a categorical feature will be expanded to as many elements as there are levels, all with the same value.
- `ExponentialDispersionModel` gains a `dispersion()` method.
- `BinomialDistribution` and `TweedieDistribution` gain a `log_likelihood()` method.
- The `fit()` method of *GeneralizedLinearRegressor* and *GeneralizedLinearRegressorCV* now saves the column types of pandas data frames.
- *GeneralizedLinearRegressor* and *GeneralizedLinearRegressorCV* gain two properties: `family_instance` and `link_instance`.
- *std_errors()* and *covariance_matrix()* have been added and support non-robust, robust (HC-1), and clustered covariance matrices.
- *GeneralizedLinearRegressor* and *GeneralizedLinearRegressorCV* now accept `family='gaussian'` as an alternative to `family='normal'`.

Bug fix:

- The `score()` method of *GeneralizedLinearRegressor* and *GeneralizedLinearRegressorCV* now accepts data frames.
- Upgraded the code to use tabmat 3.0.0.

Other:

- A major overhaul of the documentation. Everything is better!
- The methods of the link classes will now return scalars when given scalar inputs. Under certain circumstances, they'd return zero-dimensional arrays.
- There is a new benchmark available `glm_benchmarks_run` based on the Boston housing dataset. See here.
- `glm_benchmarks_analyze` now includes `offset` in the index. See here.
- `glmnet_python` was removed from the benchmarks suite.
- The innermost coordinate descent was optimized. This speeds up coordinate descent dominated problems like LASSO by about 1.5-2x. See here.

9.19 1.5.1 - 2021-07-22

Bug fix:

- Have the `linear_predictor()` and `predict()` methods of *GeneralizedLinearRegressor* and *GeneralizedLinearRegressorCV* honor the offset when `alpha` is `None`.

9.20 1.5.0 - 2021-07-15

New features:

- The linear_predictor() and predict() methods of *GeneralizedLinearRegressor* and *GeneralizedLinearRegressorCV* gain an alpha parameter (in complement to alpha_index). Moreover, they are now able to predict for multiple penalties.

Other:

- Methods of *Link* now consistently return NumPy arrays, whereas they used to preserve pandas series in special cases.
- Don't list sparse_dot_mkl as a runtime requirement from the conda recipe.
- The minimal numpy pin should be dependent on the numpy version in host and not fixed to 1.16.

9.21 1.4.3 - 2021-06-25

Bug fix:

- copy_X = False will now raise a value error when X has dtype int32 or int64. Previously, it would only raise for dtype int64.

9.22 1.4.2 - 2021-06-15

Tutorials and documentation improvements:

- Adding tutorials to the documentation.
- Additional documentation improvements.

Bug fix:

- Verbose progress bar now working again.

Other:

- Small improvement in documentation for the alpha_index argument to *predict()*.
- Pinned pre-commit hooks versions.

9.23 1.4.1 - 2021-05-01

We now have Windows builds!

9.24 1.4.0 - 2021-04-13

Deprecations:

- Fusing the `alpha` and `alphas` arguments for *GeneralizedLinearRegressor*. `alpha` now also accepts array like inputs. `alphas` is now deprecated but can still be used for backward compatibility. The `alphas` argument will be removed with the next major version.

Bug fix:

- We removed entry points to functions in `glum_benchmarks` from the conda package.

9.25 1.3.1 - 2021-04-12

Bug fix:

- `glum._distribution.unit_variance_derivative()` is evaluating a proper numexpr expression again (regression in 1.3.0).

9.26 1.3.0 - 2021-04-12

New features:

- We added a new solver based on `scipy.optimize.minimize(method='trust-constr')`.
- We added support for linear inequality constraints of type `A_ineq.dot(coef_) <= b_ineq`.

9.27 1.2.0 - 2021-02-04

We removed `glum_benchmarks` from the conda package.

9.28 1.1.1 - 2021-01-11

Maintenance release to get a fresh build for OSX.

9.29 1.1.0 - 2020-11-23

New feature:

- Direct support for pandas categorical types in `fit` and `predict`. These will be converted into a `CategoricalMatrix`.

9.30 1.0.1 - 2020-11-12

This is a maintenance release to be compatible with tabmat>=1.0.0.

9.31 1.0.0 - 2020-11-11

Other:

- Renamed alpha_level attribute of *GeneralizedLinearRegressor* and *GeneralizedLinearRegressorCV* to alpha_index.
- Clarified behavior of scale_predictors.

9.32 0.0.15 - 2020-11-11

Other:

- Pin tabmat<1.0.0 as we are expecting a breaking change with version 1.0.0.

9.33 0.0.14 - 2020-08-06

New features:

- Add Tweedie Link.
- Allow infinite bounds.

Bug fixes:

- Unstandardize regularization path.
- No copying in predict.

Other:

- Various memory and performance improvements.
- Update pre-commit hooks.

9.34 0.0.13 - 2020-07-23

See git history.

9.35 0.0.12 - 2020-07-07

See git history.

9.36 0.0.11 - 2020-07-02

See git history.

9.37 0.0.10 - 2020-06-30

See git history.

9.38 0.0.9 - 2020-06-26

See git history.

9.39 0.0.8 - 2020-06-24

See git history.

9.40 0.0.7 - 2020-06-17

See git history.

9.41 0.0.6 - 2020-06-16

See git history.

9.42 0.0.5 - 2020-06-10

See git history.

9.43 0.0.4 - 2020-06-08

See git history.

9.44 0.0.3 - 2020-06-08

See git history.

genindex

BIBLIOGRAPHY

[Yuan2012] Yuan, G. X., Ho, C. H., & Lin, C. J. (2012). An improved glmnet for l1-regularized logistic regression. The Journal of Machine Learning Research, 13(1), 1999-2030.

PYTHON MODULE INDEX

g

glum, 77

INDEX

A

aic() (*glum.GeneralizedLinearRegressor method*), 96

aicc() (*glum.GeneralizedLinearRegressor method*), 96

alpha_ (*glum.GeneralizedLinearRegressorCV attribute*), 108

alphas_ (*glum.GeneralizedLinearRegressorCV attribute*), 109

B

bic() (*glum.GeneralizedLinearRegressor method*), 97

BinomialDistribution (*class in glum*), 77

C

cat_missing_method (*glum.GeneralizedLinearRegressorCV attribute*), 110

cat_missing_name (*glum.GeneralizedLinearRegressorCV attribute*), 110

categorical_format (*glum.GeneralizedLinearRegressorCV attribute*), 109

CloglogLink (*class in glum*), 80

coef_ (*glum.GeneralizedLinearRegressor attribute*), 95

coef_ (*glum.GeneralizedLinearRegressorCV attribute*), 109

coef_path_ (*glum.GeneralizedLinearRegressorCV attribute*), 109

coef_table() (*glum.GeneralizedLinearRegressor method*), 97

coef_table() (*glum.GeneralizedLinearRegressorCV method*), 110

covariance_matrix() (*glum.GeneralizedLinearRegressor method*), 98

covariance_matrix() (*glum.GeneralizedLinearRegressorCV method*), 111

D

derivative() (*glum.CloglogLink method*), 80

derivative() (*glum.IdentityLink method*), 117

derivative() (*glum.Link method*), 122

derivative() (*glum.LogitLink method*), 123

derivative() (*glum.LogLink method*), 122

derivative() (*glum.TweedieLink method*), 138

deviance() (*glum.BinomialDistribution method*), 77

deviance() (*glum.ExponentialDispersionModel method*), 81

deviance() (*glum.GammaDistribution method*), 84

deviance() (*glum.GeneralizedHyperbolicSecant method*), 88

deviance() (*glum.InverseGaussianDistribution method*), 118

deviance() (*glum.NegativeBinomialDistribution method*), 124

deviance() (*glum.NormalDistribution method*), 127

deviance() (*glum.PoissonDistribution method*), 131

deviance() (*glum.TweedieDistribution method*), 134

deviance_derivative() (*glum.BinomialDistribution method*), 77

deviance_derivative() (*glum.ExponentialDispersionModel method*), 81

deviance_derivative() (*glum.GammaDistribution method*), 85

deviance_derivative() (*glum.GeneralizedHyperbolicSecant method*), 88

deviance_derivative() (*glum.InverseGaussianDistribution method*), 118

deviance_derivative() (*glum.NegativeBinomialDistribution method*), 124

deviance_derivative() (*glum.NormalDistribution method*), 128

deviance_derivative() (*glum.PoissonDistribution method*), 131

deviance_derivative() (*glum.TweedieDistribution method*), 134

deviance_path_ (*glum.GeneralizedLinearRegressorCV attribute*), 109

dispersion() (*glum.BinomialDistribution method*), 77

dispersion() (*glum.ExponentialDispersionModel method*), 82

dispersion() (*glum.GammaDistribution method*), 85

dispersion() (*glum.GeneralizedHyperbolicSecant method*), 88
dispersion() (*glum.InverseGaussianDistribution method*), 119
dispersion() (*glum.NegativeBinomialDistribution method*), 125
dispersion() (*glum.NormalDistribution method*), 128
dispersion() (*glum.PoissonDistribution method*), 131
dispersion() (*glum.TweedieDistribution method*), 135

E

eta_mu_deviance() (*glum.BinomialDistribution method*), 78
eta_mu_deviance() (*glum.ExponentialDispersionModel method*), 82
eta_mu_deviance() (*glum.GammaDistribution method*), 85
eta_mu_deviance() (*glum.GeneralizedHyperbolicSecant method*), 89
eta_mu_deviance() (*glum.InverseGaussianDistribution method*), 119
eta_mu_deviance() (*glum.NegativeBinomialDistribution method*), 125
eta_mu_deviance() (*glum.NormalDistribution method*), 128
eta_mu_deviance() (*glum.PoissonDistribution method*), 132
eta_mu_deviance() (*glum.TweedieDistribution method*), 135
expected_information (*glum.GeneralizedLinearRegressorCV attribute*), 109
ExponentialDispersionModel (*class in glum*), 81

F

family_instance (*glum.GeneralizedLinearRegressor property*), 99
family_instance (*glum.GeneralizedLinearRegressorCV property*), 112
fit() (*glum.GeneralizedLinearRegressor method*), 99
fit() (*glum.GeneralizedLinearRegressorCV method*), 112

G

GammaDistribution (*class in glum*), 84
GeneralizedHyperbolicSecant (*class in glum*), 87
GeneralizedLinearRegressor (*class in glum*), 91
GeneralizedLinearRegressorCV (*class in glum*), 105
get_formatted_diagnostics() (*glum.GeneralizedLinearRegressor method*), 100
get_formatted_diagnostics() (*glum.GeneralizedLinearRegressorCV method*), 113
get_link() (*in module glum*), 138
get_metadata_routing() (*glum.GeneralizedLinearRegressor method*), 100
get_metadata_routing() (*glum.GeneralizedLinearRegressorCV method*), 113
get_params() (*glum.GeneralizedLinearRegressor method*), 100
get_params() (*glum.GeneralizedLinearRegressorCV method*), 113
glum
    module, 77

I

IdentityLink (*class in glum*), 117
in_y_range() (*glum.BinomialDistribution method*), 78
in_y_range() (*glum.ExponentialDispersionModel method*), 82
in_y_range() (*glum.GammaDistribution method*), 86
in_y_range() (*glum.GeneralizedHyperbolicSecant method*), 89
in_y_range() (*glum.InverseGaussianDistribution method*), 119
in_y_range() (*glum.NegativeBinomialDistribution method*), 125
in_y_range() (*glum.NormalDistribution method*), 129
in_y_range() (*glum.PoissonDistribution method*), 132
in_y_range() (*glum.TweedieDistribution method*), 135
include_lower_bound (*glum.ExponentialDispersionModel property*), 82
include_lower_bound (*glum.TweedieDistribution property*), 135
include_upper_bound (*glum.ExponentialDispersionModel property*), 82
intercept_ (*glum.GeneralizedLinearRegressor attribute*), 95
intercept_ (*glum.GeneralizedLinearRegressorCV attribute*), 109
inverse() (*glum.CloglogLink method*), 80
inverse() (*glum.IdentityLink method*), 117
inverse() (*glum.Link method*), 122
inverse() (*glum.LogitLink method*), 123
inverse() (*glum.LogLink method*), 122
inverse() (*glum.TweedieLink method*), 138
inverse_derivative() (*glum.CloglogLink method*), 80
inverse_derivative() (*glum.IdentityLink method*), 118
inverse_derivative() (*glum.Link method*), 122
inverse_derivative() (*glum.LogitLink method*), 123
inverse_derivative() (*glum.LogLink method*), 123

inverse_derivative() (*glum.TweedieLink method*), 138
inverse_derivative2() (*glum.CloglogLink method*), 80
inverse_derivative2() (*glum.IdentityLink method*), 118
inverse_derivative2() (*glum.Link method*), 122
inverse_derivative2() (*glum.LogitLink method*), 123
inverse_derivative2() (*glum.LogLink method*), 123
inverse_derivative2() (*glum.TweedieLink method*), 138
InverseGaussianDistribution (*class in glum*), 118

L

l1_ratio_ (*glum.GeneralizedLinearRegressorCV attribute*), 109
linear_predictor() (*glum.GeneralizedLinearRegressor method*), 101
linear_predictor() (*glum.GeneralizedLinearRegressorCV method*), 113
Link (*class in glum*), 121
link() (*glum.CloglogLink method*), 81
link() (*glum.IdentityLink method*), 118
link() (*glum.Link method*), 122
link() (*glum.LogitLink method*), 123
link() (*glum.LogLink method*), 123
link() (*glum.TweedieLink method*), 138
link_instance (*glum.GeneralizedLinearRegressor property*), 101
link_instance (*glum.GeneralizedLinearRegressorCV property*), 114
log_likelihood() (*glum.BinomialDistribution method*), 78
log_likelihood() (*glum.GammaDistribution method*), 86
log_likelihood() (*glum.InverseGaussianDistribution method*), 120
log_likelihood() (*glum.NegativeBinomialDistribution method*), 125
log_likelihood() (*glum.NormalDistribution method*), 129
log_likelihood() (*glum.PoissonDistribution method*), 132
log_likelihood() (*glum.TweedieDistribution method*), 135
LogitLink (*class in glum*), 123
LogLink (*class in glum*), 122
lower_bound (*glum.ExponentialDispersionModel property*), 83
lower_bound (*glum.TweedieDistribution property*), 136

M

module
glum, 77

N

n_iter_ (*glum.GeneralizedLinearRegressor attribute*), 96
n_iter_ (*glum.GeneralizedLinearRegressorCV attribute*), 109
NegativeBinomialDistribution (*class in glum*), 124
NormalDistribution (*class in glum*), 127

P

PoissonDistribution (*class in glum*), 130
power (*glum.TweedieDistribution property*), 136
predict() (*glum.GeneralizedLinearRegressor method*), 101
predict() (*glum.GeneralizedLinearRegressorCV method*), 114

R

report_diagnostics() (*glum.GeneralizedLinearRegressor method*), 102
report_diagnostics() (*glum.GeneralizedLinearRegressorCV method*), 114
robust (*glum.GeneralizedLinearRegressorCV attribute*), 109
rowwise_gradient_hessian() (*glum.BinomialDistribution method*), 78
rowwise_gradient_hessian() (*glum.ExponentialDispersionModel method*), 83
rowwise_gradient_hessian() (*glum.GammaDistribution method*), 86
rowwise_gradient_hessian() (*glum.GeneralizedHyperbolicSecant method*), 89
rowwise_gradient_hessian() (*glum.InverseGaussianDistribution method*), 120
rowwise_gradient_hessian() (*glum.NegativeBinomialDistribution method*), 126
rowwise_gradient_hessian() (*glum.NormalDistribution method*), 129
rowwise_gradient_hessian() (*glum.PoissonDistribution method*), 132
rowwise_gradient_hessian() (*glum.TweedieDistribution method*), 136

S

score() (*glum.GeneralizedLinearRegressor method*), 102 score() (*glum.GeneralizedLinearRegressorCV method*), 115

set_params() (*glum.GeneralizedLinearRegressor method*), 102

set_params() (*glum.GeneralizedLinearRegressorCV method*), 115

std_errors() (*glum.GeneralizedLinearRegressor method*), 103

std_errors() (*glum.GeneralizedLinearRegressorCV method*), 115

T

theta (*glum.NegativeBinomialDistribution property*), 126

to_tweedie() (*glum.BinomialDistribution method*), 79

to_tweedie() (*glum.CloglogLink method*), 81

to_tweedie() (*glum.ExponentialDispersionModel method*), 83

to_tweedie() (*glum.GammaDistribution method*), 86

to_tweedie() (*glum.GeneralizedHyperbolicSecant method*), 89

to_tweedie() (*glum.IdentityLink method*), 118

to_tweedie() (*glum.InverseGaussianDistribution method*), 120

to_tweedie() (*glum.Link method*), 122

to_tweedie() (*glum.LogitLink method*), 124

to_tweedie() (*glum.LogLink method*), 123

to_tweedie() (*glum.NegativeBinomialDistribution method*), 126

to_tweedie() (*glum.NormalDistribution method*), 129

to_tweedie() (*glum.PoissonDistribution method*), 132

to_tweedie() (*glum.TweedieDistribution method*), 136

to_tweedie() (*glum.TweedieLink method*), 138

TweedieDistribution (*class in glum*), 134

TweedieLink (*class in glum*), 137

U

unit_deviance() (*glum.BinomialDistribution method*), 79

unit_deviance() (*glum.ExponentialDispersionModel method*), 83

unit_deviance() (*glum.GammaDistribution method*), 86

unit_deviance() (*glum.GeneralizedHyperbolicSecant method*), 89

unit_deviance() (*glum.InverseGaussianDistribution method*), 120

unit_deviance() (*glum.NegativeBinomialDistribution method*), 126

unit_deviance() (*glum.NormalDistribution method*), 129

unit_deviance() (*glum.PoissonDistribution method*), 133

unit_deviance() (*glum.TweedieDistribution method*), 136

unit_deviance_derivative() (*glum.BinomialDistribution method*), 79

unit_deviance_derivative() (*glum.ExponentialDispersionModel method*), 83

unit_deviance_derivative() (*glum.GammaDistribution method*), 86

unit_deviance_derivative() (*glum.GeneralizedHyperbolicSecant method*), 89

unit_deviance_derivative() (*glum.InverseGaussianDistribution method*), 120

unit_deviance_derivative() (*glum.NegativeBinomialDistribution method*), 126

unit_deviance_derivative() (*glum.NormalDistribution method*), 129

unit_deviance_derivative() (*glum.PoissonDistribution method*), 133

unit_deviance_derivative() (*glum.TweedieDistribution method*), 136

unit_variance() (*glum.BinomialDistribution method*), 79

unit_variance() (*glum.ExponentialDispersionModel method*), 83

unit_variance() (*glum.GammaDistribution method*), 87

unit_variance() (*glum.GeneralizedHyperbolicSecant method*), 90

unit_variance() (*glum.InverseGaussianDistribution method*), 120

unit_variance() (*glum.NegativeBinomialDistribution method*), 126

unit_variance() (*glum.NormalDistribution method*), 130

unit_variance() (*glum.PoissonDistribution method*), 133

unit_variance() (*glum.TweedieDistribution method*), 137

unit_variance_derivative() (*glum.BinomialDistribution method*), 79

unit_variance_derivative() (*glum.ExponentialDispersionModel method*), 83

unit_variance_derivative() (*glum.GammaDistribution method*), 87

unit_variance_derivative() (*glum.GeneralizedHyperbolicSecant method*), 90

unit_variance_derivative() (*glum.InverseGaussianDistribution method*),

121
unit_variance_derivative() (*glum.NegativeBinomialDistribution method*), 127
unit_variance_derivative() (*glum.NormalDistribution method*), 130
unit_variance_derivative() (*glum.PoissonDistribution method*), 133
unit_variance_derivative() (*glum.TweedieDistribution method*), 137
upper_bound (*glum.ExponentialDispersionModel property*), 84

V

variance() (*glum.BinomialDistribution method*), 79
variance() (*glum.ExponentialDispersionModel method*), 84
variance() (*glum.GammaDistribution method*), 87
variance() (*glum.GeneralizedHyperbolicSecant method*), 90
variance() (*glum.InverseGaussianDistribution method*), 121
variance() (*glum.NegativeBinomialDistribution method*), 127
variance() (*glum.NormalDistribution method*), 130
variance() (*glum.PoissonDistribution method*), 133
variance() (*glum.TweedieDistribution method*), 137
variance_derivative() (*glum.BinomialDistribution method*), 80
variance_derivative() (*glum.ExponentialDispersionModel method*), 84
variance_derivative() (*glum.GammaDistribution method*), 87
variance_derivative() (*glum.GeneralizedHyperbolicSecant method*), 90
variance_derivative() (*glum.InverseGaussianDistribution method*), 121
variance_derivative() (*glum.NegativeBinomialDistribution method*), 127
variance_derivative() (*glum.NormalDistribution method*), 130
variance_derivative() (*glum.PoissonDistribution method*), 133
variance_derivative() (*glum.TweedieDistribution method*), 137

W

wald_test() (*glum.GeneralizedLinearRegressor method*), 103
wald_test() (*glum.GeneralizedLinearRegressorCV method*), 116