Fast and user-friendly GLMs with glum 3

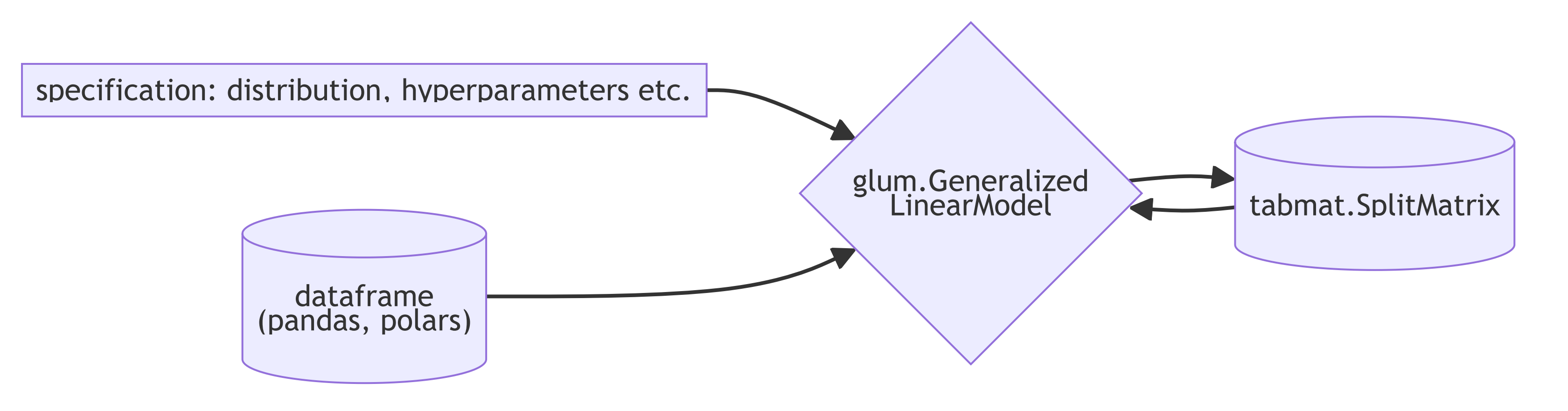
Matthias Schmidtblaicher, QuantCo

Glum

- Library for Generalized Linear Models (GLMs).
- Originally built in 2020, when there was no good option to estimate GLMs in Python. Best option was glmnet in R.
- Designed along three principles:
    - Python-first,
    - performant,
    - feature-rich: supporting, among others:
        - L1/L2 regularization,
        - custom regularization matrices (e.g., for spatial smoothing or for penalized splines),
        - coefficient bounds,
        - efficient cross-validation and regularization path computation,
        - standard errors, and
        - statistical testing.

Tabmat

- Library with standardized interface for dense, sparse, and categorical matrices.



Plan of the Talk

- Background on GLMs.
- How to compute fast sandwich products with tabmat.
- Formula interface of glum 3.
- Other new features in glum 3.

What is a GLM?

- Exponential Dispersion Model (EDM) with probability density function for observation i:

$$f(y_i \mid \mu_i, \sigma_i^2) = f_0(\sigma_i^2, y_i) \exp\left(\frac{\theta_i y_i - b(\theta_i)}{\sigma_i^2} \right).$$

- f depends on θ_i only through "exponential tilt."
- Mean is $\mu_i = b'(\theta_i)$ and the variance is $b''(\theta_i)\sigma_i^2$.
- A GLM is an EDM in which a monotone transformation of the mean is a linear function of p predictors,

$$g(\mu_i) = \sum_{j=1}^{p} x_{ij}\beta_j.$$

Matrix operations in GLMs

- β can be estimated by the following equations:

$$\mathbf{X}'\mathrm{diag}(\mathbf{v}(\mu)))(\mathbf{y} - \mu) = 0,$$

- with feature matrix $\mathbf{X}$, outcome vector $\mathbf{y}$, and prediction vector μ.

- Derivation: set the score of the log likelihood with respect to β_j to zero and stack results into matrices (see appendix).

- Newton-Raphson requires Fisher information (negative of expected value of Hessian), $\mathbf{X}'\mathrm{diag}(\mathbf{v})\mathbf{X}$.

Training a GLM quickly

- Various optimizations of solvers in glum:
    - Hyperparameters.
    - Dealing with numerical instabilities, e.g., by standardization.
    - Cythonizing coordinate descent, likelihoods, scores, etc.
- Main speedup is through matrix operations in tabmat:
    - Sandwich products $\mathbf{X}'\mathrm{diag}(\mathbf{v})\mathbf{X}$ with $n \times p$ $\mathbf{X}$, $n \gg p$.
    - Also matrix-vector products such as $\mathbf{X}\beta$ or cross-products $\mathbf{X}'\mathrm{diag}(\mathbf{v})\mathbf{M}$.
    - A lot of structure can be exploited!

Sandwich product with one-hot-encoded (OHE) categoricals

```python
import numpy as np
import pandas as pd
import tabmat as tm
from scipy import sparse

rng = np.random.default_rng()

cat = pd.Categorical(rng.integers(low=0, high=200, size=100_000))

X = pd.get_dummies(cat)
X.head()
```

	0	1	2	3	4	5	6	7	8	9	...	190	191	192	1
0	False	False	False	False	False	False	False	False	False	False	...	False	False	False	Fa
1	False	False	False	False	False	False	False	False	False	False	...	False	False	False	Fa
2	False	False	False	False	False	False	False	False	False	False	...	False	False	False	Fa
3	False	False	False	False	False	False	False	False	False	False	...	False	False	False	Fa
4	False	False	False	False	False	False	False	False	False	False	...	False	False	False	Fa

5 rows × 200 columns

Naive implementation

```
d = rng.uniform(size=len(cat))
```

```
%%timeit

X = pd.get_dummies(cat, dtype="float64").to_numpy()
X.T * d[np.newaxis, :] @ X
```

```
130 ms ± 13.3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Implementing the dense sandwich product in C++ and other speedups

```
%%timeit

X_dense = tm.DenseMatrix(pd.get_dummies(cat, dtype="float64").to_numpy())
X_dense.sandwich(d)
```

```
29.8 ms ± 395 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Specialized sandwich product for OHE categoricals

```
%%timeit

X_cat = tm.CategoricalMatrix(cat)
X_cat.sandwich(d)
```

```
92.8 µs ± 1.21 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops eac
h)
```

Speeding up sandwich products for OHE categoricals (pseudocode)

- `i`, `j` 'th element of sandwich product is `sum_k X[k, i] d[k] X[k, j]`.
- If `i != j`, `sum_k X[k, i] d[k] X[k, j] = 0` because OHE matrices have only one nonzero entry per row.
- For `i == j`, `sandwich(X, d)[i, i] = sum_k X[k, i] d[k]`, because OHE matrices consist only of zeroes and ones and `1*1=1`.
- So `sandwich(X, d) = diag(X.T @ d)`.
- No need to expand the categorical to `X`. Instead, keep categorical codes in `cat`, initialize the sandwich matrix with shape `(p, p)` as zeroes, and compute:

```
for k in range(n):
    i = cat[k]
    sandwich(X, d)[i, i] += d[k]
```

Combining categorical, dense, and sparse matrices

```
dns = rng.normal(size=(100_000, 100))
sps = sparse.random(100_000, 100, density=0.01, random_state=rng)

sm = tm.SplitMatrix(
    [tm.CategoricalMatrix(cat), tm.DenseMatrix(dns), tm.SparseMatrix(sps)]
)


sm.sandwich(d)
```

```
array([[2.62108688e+02, 0.00000000e+00, ..., 7.60231539e-01,
        1.53523965e+00],
       [0.00000000e+00, 2.70724717e+02, ..., 1.49602246e-01,
        1.42353807e+00],
       ...,
       [7.60231539e-01, 1.49602246e-01, ..., 1.66799521e+02,
        1.26643752e+00],
       [1.53523965e+00, 1.42353807e+00, ..., 1.26643752e+00,
        1.84677336e+02]])
```

Usually, a `tm.SplitMatrix` will be initialized via `tm.from_(pandas|polars|formula)`.

GLM building

- Compared to other machine learning algorithms, GLM's require substantial manual preprocessing to get `X`.
- Manual preprocessing has pros and cons.

Pros	**Cons**
Modeller learns about main effects in the data.	Time-intensive.
Interpretable, since all effects and interactions were explicitly modelled.	To achieve high predictive performance, complicated transformations and interactions are often needed, harming interpretability.

- Time-tested idea to make GLM model building less time-intensive is to specify models via Wilkinson (1973) formulas: a domain-specific language that captures the essence of a model specification.

- Glum 3 introduces Wilkinson formulas, based on formulaic.

Formulas

```python
import formulaic
import glum

from vega_datasets import data

cars = data.cars().dropna()

formula = (
    "Miles_per_Gallon ~ {np.log(2.20462 * Weight_in_lbs)} "
    "+ bs(Horsepower, 3) + C(Cylinders) * Origin"
)
model = glum.GeneralizedLinearRegressor(
    formula=formula, drop_first=True, fit_intercept=False, family="gamma", alpha=0
)
model.fit(cars[:380])
model.predict(cars[380:])
```

```
array([18.22549226, 19.73007056, 27.66390534, 17.8747613 , 32.21080938,
       27.41974637, 29.19519495, 28.93554487, 28.86996769, 27.100013  ,
       28.86903563, 28.99728628])
```

Efficient interactions

- `tm.from_formula` has various speedups for interactions:
  - Sparsity preserving: interaction of sparse and dense columns will always be sparse.
  - Interactions between categoricals are internally represented by one new categorical column.
- Take the following input:

```
df = pd.DataFrame(
    {
        "c1": pd.Categorical(rng.integers(low=0, high=200, size=1_000_000)),
        "c2": pd.Categorical(rng.integers(low=0, high=200, size=1_000_000)),
        "y": np.random.rand(1_000_000),
    }
)
```

Encoding categorical interactions outside of glum

The following crashes on my computer:

```
X = formulaic.model_matrix("c1 : c2", data=df)
big_model_ext = glum.GeneralizedLinearRegressor(alpha=1).fit(X=X, y=df["y"])
```

```
The Kernel crashed while executing code in the current cell or a previou
s cell.

Please review the code in the cell(s) to identify a possible cause of th
e failure.

Click <a href='https://aka.ms/vscodeJupyterKernelCrash'>here</a> for mor
e info.

View Jupyter <a href='command:jupyter.viewOutput'>log</a> for further de
tails.
```

Encoding categorical interactions internally

Letting glum create the model matrix:

```
big_model = glum.GeneralizedLinearRegressor(formula="y ~ c1 : c2", alpha=1).fit(df
big_model.coef_table()
```

```
intercept           4.999537e-01
c1[0]:c2[0]         1.205366e-07
c1[1]:c2[0]         5.852428e-07
c1[2]:c2[0]        -1.144714e-09
c1[3]:c2[0]        -1.398972e-07
                        ...
c1[195]:c2[199]     8.068055e-07
c1[196]:c2[199]    -1.201487e-06
c1[197]:c2[199]    -1.315570e-06
c1[198]:c2[199]     2.831186e-07
c1[199]:c2[199]    -5.820133e-07
Name: coef, Length: 40001, dtype: float64
```

Other usability improvements in glum 3

- Missings in categorical can be custom-handled, e.g., as `"y ~ C(x1, missing_method='zero') + C(x2, missing_method='convert')"`.
- Term names:
  - A term is all columns in the "expanded" matrix pertaining to a single feature prior to expansion (e.g., before OHE).
  - Wald tests can test terms such as `"model.wald_test(cars, terms=["bs(Horsepower, 3)"], r=[0])"`.
- Wald tests can also be specified with formulas, e.g., ``model.wald_test(cars, terms="`bs(Horsepower, 3)[1]` = 2 * `bs(Horsepower, 3)[2]`")``.
- Freely customizable interaction separators and categorical formats.

Want to contribute?

- Try out glum/tabmat for yourself.
- Report bugs through our issue tracker.
- Reach out if you want to contribute a pull request.

Q

Questions?

Appendix

Deriving the score equations of a GLM

One can show that the mean of an EDM is $\mu_i = b'(\theta_i)$ and that the variance is $b''(\theta_i)\sigma_i^2$. Define unit variance $v_i = b''(\theta_i)$.

Log likelihood with respect to θ_i is

$$\frac{\theta_i y_i - b(\theta_i)}{\sigma_i^2} + \text{constant}.$$

Score of the log likelihood with respect to one β_j for one observation is:

$$\begin{aligned} &\frac{1}{\sigma_i^2}(y_i - b'(\theta_i))\frac{\partial \theta_i}{\partial \mu_i}\frac{\partial \mu_i}{\partial \beta_j} \\ =& \frac{1}{\sigma_i^2}(y_i - \mu_i)\frac{1}{v_i}\frac{x_{ij}}{g'(\mu_i)}. \end{aligned}$$

Setting to zero and stacking the data into matrices,

$$\mathbf{X}'\text{diag}(\mathbf{v})(\mathbf{y} - \mu) = 0,$$

with $\frac{1}{v_i(\mu_i)g'(\mu_i)\sigma_i^2}$ in $\mathbf{v}$.

Benchmarking the sparse sandwich product

Slower than equivalent code for `tm.DenseMatrix` because `pd.get_dummies` appears to slow down when `sparse` argument is set.

```
%%timeit

X_sparse = tm.SparseMatrix(pd.get_dummies(cat, dtype="float64", sparse=True))
X_sparse.sandwich(d)
```

```
118 ms ± 1.14 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Compare only the tabmat parts:

```
%%timeit
X_sparse.sandwich(d)
```

```
275 µs ± 3.7 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
%%timeit
X_dense.sandwich(d)
```

```
19.9 ms ± 567 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```