Insurance Pricing With GLMs

Alan Chalk

2025-01-22

Table of contents

Acknowledgements

To Serban Dragne:

You've taught me so much about R, Python, SQL and ML in general.

You are a constant reminder to me about how little I know.

Thanks.

I think.

1 Preface

Dear Reader,

Can we fit good GLMs?

Can we use the data to choose which features to select?

Can we fit those features accurately, with manual smoothing left as close to the end of the process as possible?

Can we automatically find and fit interactions?

Some techniques to start us on this journey are discussed in the CAS monograph: From GLMs to Comprehensive Insurance Pricing: Techniques and Challenges.

You might want to have a go yourself by running R code. This could be a for a few reasons:

- Your or your company prefer to build code yourself rather than buy a propriety software. (For full information on the *build vs buy* debate, see elsewhere.)
- You have software that does this already, but you want to make sure you really understand how it works. Often the best way to understand something is to build and run it yourself.
- You are a glutton for punishment.

Whatever you reason for being here, our aim is to provide you with enough information and R code to to build pretty decent GLMs and to understand what you are doing along the way.

Happy coding!

2 Getting started

Before you start any programming task there are some things you need to deal with.

2.1 Directory structure

In whichever directory your project resides, decide upfront what your directory structure could be. (The other approach - called the mayhem approach - is to allow each person in your team to add randomly named directories as and when they think they need them.)

We always have the following subdirectories (capitalized exactly as given):

- RawData
- IntermediateData
- RCode
- RData
- ROutput

If we are also using SAS or Python we will have in addition:

- PCode
- PData
- SASCode
- SASData
- etc

The content of what needs to go in each of the above directories, is hopefully obvious. If not, you will see what goes where as we proceed with our analysis here. Ofcourse the above approach is far from the only way. In Approaching (almost) any machine learning problem (Thakur 2020), Thakur uses:

- input (for raw data)
- src (for Python code)
- models
- notebooks
- README.md
- LICENSE

The details don't matter (much). The main point is to figure something out before you start and then get everyone to stick to it.

2.2 Sequential naming of program files

Imagine you are reviewing code created by a team member. You know where to go, the RCode directory. In there you find various files

- read_data.R
- adjust_data.R
- manipulate_data.R
- correct_data.R
- and so on.

You want to replicate the process by running the files in order. Which order should you run the files in? Would it not have been much easier if you were faced with:

- 01a_read_data.R
- 02c_adjust_data.R
- 02a_manipulate_data.R
- 02b_correct_data.R
- and so on.

In general it seems like a good idea to give each file a prefix which indicates where it come in the overall progamming pipeline.

We have found in pretty much every project, that we need to do at least four things, read the raw data, manipulate it, carry out some initial data exploration, do the modelling. We therefore try to stick to:

- 01a/b/c etc for reading raw data
- 02/a/b/c etc for data manipulation
- 03/a/b/c etc for data exploration
- 04/a/b/c etc for fitting (training) models.

Once again, exactly how you do this, is not important. Only that you do it.

2.3 Naming data files

You are busy reviewing a project, and you come across a file `dt_train.RData`. You want to see how it was created and so you go to the program which created it. Which is…? You have no idea. So you start searching through each .R file for the code which might have created it.

Something like `save(dt_train, file = "dt_train.RData")`. You might get lucky, or you might find that this code appears in a few places, each time overwriting the previous. Imagine instead it was labelled `02a_dt_train.RData`, when the prefix tells you which program it was created in. That's what we do. You might want to do something similar.

2.4 Naming variables

Mainly. Be consistent.

Some languages (e.g. SAS) do not differentiate between lower and upper case. Some (e.g. R and Python) do. Coders in SAS can be inconsistent and get away with it. When two similar datasets are exported fom SAS to R, a program that works with one, will not work with the other if the capitalisation is inconsistent.

Consider the variable name for policyholder age. You could use policyholderAge (camel case), PolicyholderAge (Pascal case), policyholder_age (snake case). Choose one and stick to it.

Consider a data.table (in R) of data to be used for training. You could call it `train` or `dt_train` or `train_dt`. The middle of these (`dt_train`) leads us to the idea that we might want the variable names of our objects to start with something which tells us what they are.

Sometimes we might want to *top and tail* a feature. Imagine that we have a small number of policy holders who are older than 99. We create a new version which is never less than 17 and never more than 99 - `min(99, max(17, policyholder_age))`. Should we simply replace policyholder_age with the new version, or should we give in a new name. If the later, what should our new name be? (We use `policyholder_age_tt`).

Sometimes we replace a feature with its rank. Or its percentile. You can probably now guess that we use `policyholder_age_r` or `policyholder_age_p`.

We often come across abreviations which are capitalised when used in a sentence (NCB for No Claims Bonus or VRN for Vehicle Registration Number). Should we capitalize the variable names for these?

When we deal with missing values we often guess what the missing value should be. (This is known as imputation.) Then, we create an indicator, which takes the value of 1 wherever we have imputed a value in place of missing data. If `policyholder_age` is sometimes missing and we decide to use imputation to replace missing values and to use a missing value indicator, we need a name for the indicator. We use `policyholder_age_m`.

Consider three types of claim (accidental damage (ad), fire or theft (ft) and personal injury (pi)) and two items of information about these claims, their number (num) and their incurred value (inc). Our variable names can be; num_ad, num_ft, num_pi, inc_ad, ind_ft and inc_pi. Alternatively we can choose, ad_num, ad_inc etc.

Some languages have style guides. These can be useful, but the issues we come across tend to be detail which is not covered elsewhere. Throughout our code, we don't aim to know what's right (even if there is such a thing), we try to be consistent.

2.5 Version control

Talking about version control is beyond the scope of our discussion here. But, unless you are on your home PC just to learn (and maybe even then), then - obviously - do it.

2.6 Managing packages

In both R and Python, we load packages to give us access to useful code written by someone else. Packages allow us to easily read and manipulate data, explore data and carry out supervised learning tasks (such as fitting GLMs or gradient boosting models).

In Python it can be quite difficult to work with packages without using some kind of package management system. That is because often only specific versions of two different packages are compatible with each other. For example `pandas` version 2.2 is not compatible with `numpy` 1.22.3 and prior (pandas_installation_dependecies 2025).

From practical experience, it is easier to work in R without using a package management system.

3 Data preparation

It is very tempting once a dataset is available, to jump right in to the modelling stage - because that is interesting and data preparation is boring. Or because you are being chased by management for a new and improved rating plan which they in turn have promised their boss to have ready by yesterday. In this regard they will often quote the 80-20 rule to you. Meaning that you can get 80% of the benefit of a data cleansing exercise by just 20% of the effort. (Or they might use the phrase "perfect is the enemy of good".) This magical rule means that if you need a month to do data cleansing, your boss can reasonably expect it of you in just under a week, and that whatever has been left undone will not matter to the final result.

A typical outcome of poor data preparation is getting to the end of a few days (or months) of modelling and then realizing that you included a feature which should not be included. Or you treated a feature as numeric which should been treated as a character / factor. An example of the latter is the number of doors of a vehicle (in auto insurance). Treating this as a numeric feature will often lead to the assumption that the risk of accident decreases (or increases) linearly as the number of doors increases. You probably did not want to assume this. The net result of such errors can vary from some minor changes needing to be made post-hoc, to having to redo the whole exercise.

This chapter will discuss and provide code examples for various aspects of data preparation, including:

- Naming conventions for our features
- Character and numeric features
- Feature screening
- Missing values and general sense-checking

4 Finding and fitting interactions

When we train decision trees, random forests, or gradient boosting trees, or neural networks, they automatically find interactions for us (even if it is not totally straightforward to actually find out what they are). In fact, some time back, a certain type of decision tree was known as CHAID - a CHi-squared Automatic Interaction Detector.

When we train GLMs they do not automatically find interactions for us.

Thus, a price for the transparency of the GLM is that we need to find (and code up) interactions (if there are any).

The approach we take here is to first find and propose candidate interaction pairs and then, for each candidate, to code it up as a feature for our GLM and finally to fit it. Interactions which improve CV performance and seem sensible for the domain we are working in, are likely to end up in our final model.

- Domain knowledge. policyjolder age x vehicle accelaration (0-60)
- Learning from black-box models
- MARS

5 Summary

References

pandas_installation_dependecies. 2025. "Installation — Pandas Documentation." https://pandas.pydata.org/pandas-docs/stable/getting_started/install.html.

Thakur, Abhishek. 2020. *Approaching (Almost) Any Machine Learning Problem*. Abhishek Thakur.