Practitioner's Guide to Building Actuarial Reserving Workflows Using chainladder-python

John Bogaardt, FCAS, MAAA
Gene Dan, FCAS, MAAA, CSPA
Kenneth Hsu, FCAS, MAAA, CSPA

Aug 30, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Introduction

Practicing actuaries are managing more complex system and workflows every day, especially in reserving. While reserving in of itself is a complex topic, many actuaries prefer routine workflows that are intuitive and easy to manage.

This paper discusses practical considerations on the analytical and workflow structure of setting an unpaid claim estimate. Literature abounds on actuarial methods and assumptions used to tackle specific reserving issues, but there is very little research into the best practices of managing actuarial reserving workflows. Today, management teams of companies increasingly desire

1. Faster speed in end-to-end analysis,
2. Lower cost in running a reserving shop, and
3. More detailed insight into the drivers of loss experience

These pressures continue to shape the evolution of reserving against a backdrop of technology advancements that are outpacing advancements in actuarial reserving processes. They point to the need to scale up the reserving function to meet the needs of the business.

At the heart of our recommendation for scaling up a reserving function is the establishment of standards that improve the ability to scale. Standards are used throughout a variety of industries to establish common language, seamlessly integrate different technologies, and improve interoperability where possible. We propose a standard for data management of the loss triangle along with a standard for building actuarial models with an explicit declaration of model assumptions.

Our belief is that these standards can improve the efficacy of a reserving function by preserving traditional techniques and standards of practice while simultaneously enabling a foundation for more advanced data and modeling needs. These standards are borne out of the idea that a separation of concerns is a good thing for scaling up the complexity of a workflow.

In order to stay true with the intentions of this work's title, we have chosen to write this paper using open-source tools. The contents of Chapters 1, 2, and 3 was written using a markup language called reStructuredText. Chapter 4, which covers the chainladder-python tutorial, was written using a combination of Markdown and Python in a Jupyter notebook file. Together, these documents were compiled using a Python package called Sphinx which in turn uses a LaTeX engine to generate this PDF. Moreover, this project is version controlled via Git.

The paper's source code is available on a GitHub repository via the following link: https://github.com/genedan/cl-practitioners-guide. On this repo, readers can examine the development of this paper from inception to publication, and can even see the challenges faced by the authors by examining the project's issues page. By making the project's source code and development history public, we intend to demonstrate that this paper follows the guidelines that it prescribes.

CHAPTER

TWO

DATA MANAGEMENT PRINCIPLES AND IMPLEMENTATION PHILOSOPHY

2.1 Data And Logic

2.1.1 Should Data and Business Logic be Separate?

The actuarial profession is used to working with spreadsheets. The CAS technology survey consistently shows Microsoft Excel as the dominant tool used in actuarial work (Fannin 2022). Indeed, for a large swath of actuarial problems, spreadsheets are fantastic tool. According to the ASTIN 2016 Non-Life Reserving Practices Survey, the plurality of practitioners worldwide use spreadsheet software for meeting reserving obligations.

Spreadsheets differ from traditional software applications in that they generally promote interdependence between the data and business logic. Because of this, there is no difference between cells with raw data and cells with formulae. At a small enough scale, this is easy to reason about making spreadsheets great for simple models, prototypes, and when working with small data sets.

Because of the interdependence between data and logic, spreadsheet complexity tends to grow at a faster pace than analytical complexity. Most spreadsheet functions are designed to reference static cells and ranges. Changes to the shape of data often implies a manual update of references is inevitable.

The desire to decouple data and business logic is compelling. Updating a spreadsheet business logic to accommodate another year or quarter of data becomes tedious, prone to error, and is an inefficient use of actuarial time in the long run. Actuaries have developed tips and tricks to avoid this. Those familiar with Microsoft Excel can point to strategies such as including using combinations of formulas such as OFFSET/MATCH to generate dynamic cell ranges. This is a sound approach, but not without trade-offs. First, the spreadsheet becomes substantially more difficult to audit and debug because dependency tracing is eliminated. Second, spreadsheets take a hit on performance with dynamic or volatile functions. Volatile functions are recalculated every time a spreadsheet changes. Examples such as OFFSET and INDIRECT are difficult for Excel to optimize due to their dynamic nature. Lastly, complex formulas require robust documentation, and often, important detail can be lost in communication.

Visual Basic for Applications (VBA) and PowerPivot backend are excellent extensions for Microsoft Excel that naturally address some of these concerns. The inclusion of LAMBDA functions, dynamic arrays and tables with structured references are also moves towards improving the scalability of spreadsheets. These are welcome advancements in capabilities that allow for much better chances for scale. The specifics of these capabilities is beyond the scope of this paper, but the evolution of Microsoft Excel to include them is a nod to the growing analytical complexity of spreadsheet work. Despite this, data and logic remain co-mingled making it difficult to use spreadsheets as software. The separation of data and logic is a defining difference between spreadsheets and software with the latter having much more scalable virtues.

Shareability is another key requirement to a software-oriented approach to actuarial work. Every company has their own implementation of spreadsheets with much of the same basic functionality, albeit with lots of variation. As a profession that adheres to standards of practice, it's natural for commonalities in approaches to be used throughout industry. Yet spreadsheets are not shared in the public domain with as much frequency as software packages. This

is because of the co-mingled nature of data and logic as well as the approach to designing spreadsheets which often emphasizes practical and actuarial aspects rather than emphasizing quality and reusability of work product.

In the context of open-source software, reserving capabilities can be developed with much more rigor than any one company's spreadsheet implementation. Widely used open-source software logic is tested in the public domain. Usage of software by multiple people in multiple companies substantially feeds back into the quality of the work product. A 2009 meta-analysis found that over 90 percent of reviewed spreadsheets contain errors (Powell 2009). While software is not immune to errors, the expected volume of errors within a piece of software reduces as the user base of that software increases. This is particularly beneficial for actuarial teams in smaller firms without the resources to test and maintain bespoke spreadsheet applications.

Another benefit to a software-oriented approach to reserving is reusability. By design, the business logic is intended to support a variety of different use cases whether that be for start-up companies with limited history or long-tenured companies with decades of data.

Specialized reserving software exists to deal with a lot of these scalability issues already mentioned. There are several reserving softwares available and these applications are often vetted by well-funded teams of actuaries and software developers and are an excellent choice for scaling up enterprise reserving. However, these applications' source code is not open, so auditing the internal calculations can be challenging, or even impossible. Furthermore, modifications of software are generally closed to customers with niche needs. In other words, clients must work with the developers for feature enhancements, and cannot make modification to the product that they had purchased themselves.

Increases in analytical sophistication has consequences for the complexity of data, process, and workflow of a reserving work product. Contrasted against the discipline of software development, which has tools and capabilities to manage these complexities, there is a compelling argument that scaling analytical sophistication should warrant closer alignment and even adoption of software development capabilities. Software development is often subject to strict requirements to ensure a high-quality output. This often includes:

- Version control: Logic version control systems, such as git, track logic changes over time, allowing developers to manage different versions that ensure consistency, collaboration, and history tracking. In the context of analytical complexity, this allows for efficiently managing different versions of analysis and can benefit scenario testing, prototyping, and general collaboration on an analysis.
- Computational environments: Environments are distinct setups for software development that allow for variations on set-up to include different technologies or different versions of the same technology. In the context of analytical complexity, this ensures reproducibility of results.
- Testing: Software testing validates functionality, performance, and security in an automated fashion. In the context of analytical complexity, ensuring a component of a broader analysis works consistently provides assurances that evolution of an analysis does not unwittingly introduce errors.

The software development life cycle treats these capabilities as first-class citizens. The usage of these capabilities directly facilitates a higher quality work product at scale. None of these is particularly easy to use or accomplish when logic is maintained in spreadsheet software. This is in part driven by the lack of separation between data and business logic.

If the reader is comfortable with the notion that data and business logic should be separate, let's discuss how the chainladder-python package, the most popular open-source actuarial software on GitHub, approaches the topic of data and business logic standards.

2.1.2 A Standard for Data: The Multidimensional Triangle

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types.

In tidy data:

1. Each variable is a column; each column is a variable.
2. Each observation is a row; each row is an observation.
3. Each value is a cell; each cell is a single value.

Tidy data makes it easy for an analyst or a computer to extract needed variables because it provides a standard way of structuring a dataset (Wickham 2014). The basic triangle is not considered tidy data. A single variable, say losses paid, spans multiple rows. A single observation, say accident year, spans multiple columns. Data shaped as a basic triangle contains rows representing a time-dependent cohort of loss data. This is commonly the accident period, but could be report period as well as policy period. Cohorts are reviewed at regular intervals to elucidate the loss development process. In its simplest form, a triangle may look like the following:

```
In [1]: import chainladder as cl

In [2]: cl.load_sample('raa')
Out[2]:
         12       24       36       48       60       72       84       96       108    ␣
↪ 120
1981  5012.0   8269.0  10907.0  11805.0  13539.0  16181.0  18009.0  18608.0  18662.0 ␣
↪18834.0
1982   106.0   4285.0   5396.0  10666.0  13782.0  15599.0  15496.0  16169.0  16704.0    ␣
↪ NaN
1983  3410.0   8992.0  13873.0  16141.0  18735.0  22214.0  22863.0  23466.0      NaN    ␣
↪ NaN
1984  5655.0  11555.0  15766.0  21266.0  23425.0  26083.0  27067.0      NaN      NaN    ␣
↪ NaN
1985  1092.0   9565.0  15836.0  22169.0  25955.0  26180.0      NaN      NaN      NaN    ␣
↪ NaN
1986  1513.0   6445.0  11702.0  12935.0  15852.0      NaN      NaN      NaN      NaN    ␣
↪ NaN
1987   557.0   4020.0  10946.0  12314.0      NaN      NaN      NaN      NaN      NaN    ␣
↪ NaN
1988  1351.0   6947.0  13112.0      NaN      NaN      NaN      NaN      NaN      NaN    ␣
↪ NaN
1989  3133.0   5395.0      NaN      NaN      NaN      NaN      NaN      NaN      NaN    ␣
↪ NaN
1990  2063.0      NaN      NaN      NaN      NaN      NaN      NaN      NaN      NaN    ␣
↪ NaN
```

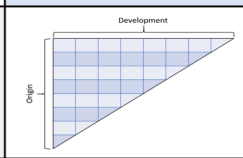
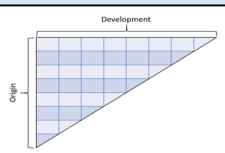
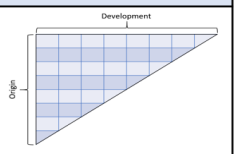
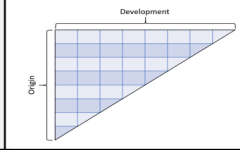
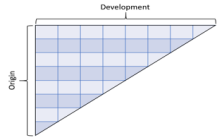
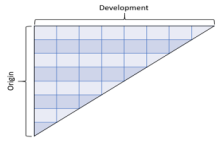
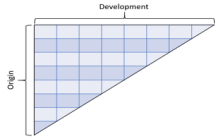
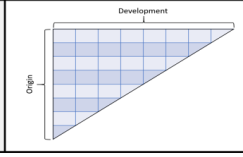
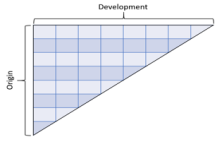
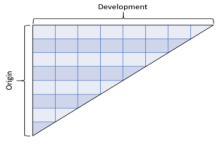
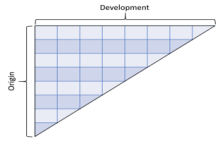
This standard view of a loss triangle is an important structure for actuarial work. On its own, it is a useful structure for performing analysis, but the lack of tidy structure makes it more challenging to derive more complex insights. For example, actuaries seldom look at a single triangle to formulate an opinion on unreported claims. Actuaries will often have a suite of triangles, many of which are arithmetic combinations of other triangles to inform their analysis. Such triangles include:

- Paid vs incurred loss data.
- Loss vs loss adjustment expense data.
- Reported, open and closed claim count data.

- Exposure-based triangles for auditable exposures.
- Reserve groupings that reflect homogenous groupings of a heterogeneous book of business.

The suite of triangles available to an actuary tend to vary along two aspects – quantitative (e.g. reported count, paid loss) and qualitative groupings (e.g. line of business, jurisdiction). These different groupings are often called measures and dimensions in data modeling.

The multidimensional triangle aims to blend the need for a suite of triangles and the benefits of tidy data. So as to differentiate between the conventional definition of a triangle and a multidimensional triangle, we will refer to the multidimensional triangle as a *Triangle*. Rather than considering each unique triangle as its own independent messy data, a single observation of a *Triangle* is a conventional triangle. A suite of conventional triangles can be laid out in tidy format in a table of triangles where each cell of the table is a conventional triangle. It can look like this:

	Column$_1$	**Column$_2$**	**Column$_{...}$**	**Column$_M$**
Index$_1$			...	
Index$_2$			...	
Index$_{...}$	...	...	...	
Index$_N$				

Here, *index* is defined analogous to the pandas library (Mckinney 2011) and includes the qualitative properties of the observation, and *column* contains the quantitative properties.

Though tidy data finds its roots in R, tidy concepts apply to all tables of data and can be queried by any dataframe library syntax. Because of the implementation of chainladder-python in the Python programming language, the syntax for working with a *Triangle* follows Python's most widely used dataframe library, pandas. Treating a suite of triangles as a tidy dataframe substantially enhances the diagnostic capabilities of the practitioner as it allows for exposition of data manipulation used by pandas while preserving access to the untidy traditional loss triangle format.

With the pandas API, we can filter our data, perform aggregations across groups, derive new quantitative measures, and apply basic arithmetic to our suite of triangles.

Triangle is not just used for selecting development patterns, it becomes a query tool for diagnostic insights into the reserve setting process. For example, the ratio of a closed count triangle to a reported count triangle yields a triangle of closure rates. A ratio of paid losses to case incurred losses yields a view into changes into paid patterns relative to incurred patterns. Arithmetic of triangles is so common in practice that it should follow the simple syntax of the arithmetic of columns in a table.

While a tidy format substantially expands on the capabilities of loss reserving data, not all use-cases can be supported by treating a basic loss triangle as an atomic unit of data. Accessing origins, development lags, and diagonals is also

a common need for actuaries. This is akin to needing to access detailed components of other complex data type such as strings and dates. Most dataframe libraries including pandas have solved for this level of access. To access these granular components of a triangle, the multidimensional triangle also borrows from the accessor capabilities of pandas. In pandas, parsing a broader text field for key pieces of information is handled by exposing the *str* object of a text column. Doing date manipulation is handled by exposing the *dt* object. As an extension of this approach, the *Triangle* exposes *origin*, *development* and *valuation* accessors to access data which allows for expanded query capabilities such as a comparative view of age-to-age factors of one development lag or run-off of claims activity over the subsequent diagonal.

Being able to manipulate a suite of triangles as a dataframe using a syntax broadly adopted by the pandas community not only allows for rapid exploration of reserving data, but also reinforces skills more broadly used across the Python data ecosystem. The trade-off of tidy vs untidy data structures is substantially diminished through the exposition of accessors.

2.1.3 A Standard for Modeling: Borrowing from Machine Learning

Estimation of an unpaid claim analysis is informed by three sources:

1. Data: This is typically a suite of triangles and was discussed in the previous section.
2. Reserving Models: Often referred to as actuarial methods. The practitioner decides which methods are appropriate for the analysis at hand. The choice of model inherently has model risk and actuaries will typically use several models to reduce this risk.
3. Assumptions: The practitioner determines a set of assumptions to parameterize each reserving model and may include how to average age-to-age factors, whether to include an exogenous tail calculation, etc.

Models and assumptions are related, but are not the same thing. In the domain of machine learning, practitioners are equipped with a diverse array of algorithms or methods. However, each algorithm comes with its own set of assumptions and requires the tuning of specific hyperparameters to effectively guide the model's convergence toward a solution. In short, assumptions are model dependent.

Taking inspiration from scikit-learn, the most popular machine learning library in Python, we can explore how general purpose modeling standards can be applied to reserving. scikit-learn includes a suite of Machine Learning estimators that range anywhere from data prep (e.g. PCA, OneHotEncoding) to classification (e.g. RandomForestClassifier, K-neighbors), to regression (e.g. LinearRegression, ElasticNet), to clustering (e.g. K-means). A consistent API across the package makes scikit-learn very usable in practice. Experimenting with different learning algorithm is as simple as substituting different estimators (Buitinck 2013)

The chainladder-python package uses the scikit-learn estimator as the foundation to model construction. Similar to scikit-learn, actuaries use a variety of techniques and algorithms to model unpaid claim estimates. These can span a variety of use cases including:

1. Selecting loss development factors (*Development*, *ClarkLDF*, *DevelopmentConstant*)
2. Extrapolating tail factors (*TailCurve*, *TailBondy*)
3. Triangle data adjustment (*ParallelogramOLF*, *BerquistSherman*)
4. Developing unpaid claim estimates (*Chainladder*, *BornhuetterFerguson*, *CapeCod*)

Model selection is a starting point for an analysis, how the model behaves can be altered through the usage of hyperparameters. For example, scikit-learn's ElasticNet estimator includes the following hyperparamters to influence how the model behaves (alpha, l1_ratio, fit_intercept, precompute, max_iter, copy_X, tol, warm_start, positive, random_state, selections). A key property of these hyperparameters is that they can be set prior to the fitting of the estimator to any data. This is similar to assumption setting where an actuary may want to influence how development factors are calculated. The Development estimator has the following hyperparameters to aid in assumption setting (n_periods, average, sigma_interpolation, drop, drop_high, drop_low, preserve, drop_valuation, drop_above, drop_below, fillna, groupby). *n_periods* would indicate the number of diagonals from a triangle to be used in selecting loss development. *average*

allows for selection between 'simple', 'volume' and 'regression'. Each of these can be varied for each development lag and are specified before fitting the estimator to a Triangle.

Analytical workflows are more complex than just fitting single estimators. Scikit-learn accommodates chaining separate algorithms together to support more complex workflows (Buitinck 2013). It's entirely reasonable to perform PCA on data before pushing it into a KNeighbors classifier. Chaining algorithms together is possible in chainladder and is facilitated through the use of composite estimators called `Pipeline`s.

As is the case with the suite of machine learning estimators, not all of use-cases are intended to develop unpaid claims estimates in isolation. An actuary may want to perform a basic chainladder projection on a Berquist-Sherman adjusted set of triangles. It is also common to see a single set of development factors being used across both a multiplicative chainladder and a Bornhuetter-Ferguson approach. Separating techniques into composable estimators allows for reuse. As a practitioner, one can declare individual estimators and use those to create a *Pipeline* that describe a reserving process.

An example reserving *Pipeline* might be declared as follows:

```
In [3]: import chainladder as cl

In [4]: cl.Pipeline(
   ...:     steps=[
   ...:       ('sample', cl.BootstrapODPSample(random_state=42)),
   ...:       ('dev', cl.Development(average='volume')),
   ...:       ('tail', cl.TailCurve(curve='exponential')),
   ...:       ('model', cl.Chainladder())
   ...:     ]
   ...: )
   ...:
Out[4]:
Pipeline(steps=[('sample', BootstrapODPSample(random_state=42)),
                ('dev', Development()), ('tail', TailCurve()),
                ('model', Chainladder())])
```

It's clear to see that this is a volume-weighted chainladder model with a tail factor set using exponential curve fitting. Further, this model will resample the *Triangle* it receives using over-dispersed poisson bootstrapping to provide a simulated set of reserve estimates.

Some advantages of this approach:

1. It is declared independent of the data it will be used on.
2. The models used are explicit: *BootstrapODPSample*, *Development*, *TailCurve* and *Chainladder*.
3. The assumptions used are also explicit: *random_state=42*, *average='volume'*, *curve='exponential'*.

These estimators also benefit from standardized models results. When performing an unpaid claim analysis, the actuary is seldom only interested in the ultimate unpaid claim amount. Projecting ultimates automatically produces IBNR and Run-Off expectations. These are standard outputs regardless of whether the practitioner uses a *CapeCod* method or a *Benktander* method. Such outputs allow for further diagnostic development such as duration and cashflow analysis and calendar period performance against prior expectations.

Leveraging the modeling framework of scikit-learn allows the practitioner and library maintainers to capitalize on lessons learned in analytical workflow management from the machine learning community. Additionally, the framework reinforces skills more broadly used across the Python data ecosystem.

The primary goals of the chainladder-python library are inherently to manage analytical complexity. It does so by exposing a code-based API to the practitioner. This enables the usage of many software development facilities that support scaling up complexity. By leveraging the syntax standards of the most popular data manipulation package

(pandas) and machine learning package (scikit-learn), chainladder-python is designed to remove as much friction from the learning process as possible.

CHAPTER

THREE

PACKAGE MANAGEMENT AND THE CHAINLADDER ECOSYSTEM

3.1 Package Management

Reserve studies apply algorithms to manipulate data. These algorithms can be written as Excel formulas, as a sequence of statements in a script, a collection of functions and classes in a package, or in a variety of other ways. The aim of this section is to promote the organization of these algorithms into packages, of which both the R and Python versions of chainladder are examples of. We will start by following what an actuary interested in applying programming on the job might do as they begin to learn their language of preference, by applying simple Python statements to an input triangle. We will then describe the problems the actuary may face when doing so and the motivations towards organizing their code into higher levels of abstraction - that is, functions, classes, and packages - designed to solve such problems.

We will then dive into how packages can be managed within and across teams and give examples of practices that can be adopted to ease the guesswork involved in coordinating such efforts.

3.1.1 Hierarchy of Abstraction

Algorithms Stored as a Sequence of Statements

Data manipulation involving code often starts out with the practitioner writing a sequence of statements that transforms imported data. The need to reuse these statements then arises, perhaps when the analysis needs to be redone the next quarter on a refreshed version of the data. One way the practitioner can make the code reusable is by storing these statements as a script and then altering the lines of code specific to the new source of data, such as editing the line where the data get imported (i.e., changing the suffix “Q3” to “Q4” of a source csv file).

While this method offers some amount of reusability, it has drawback of having the need to edit the script each time it needs to be rerun on new data. This may also lead the practitioner to copy the same lines of code multiple times if they wish to preserve a copy of the version that was used to produce the results each quarter. If a bug is discovered, or if the script itself evolves and needs to be rerun, it will be difficult to go back and edit each file or the organization of the code files will be messy if they decide to use a fresh copy of the script on the old data.

Algorithms Encapsulated as Functions

One way to avoid these issues is to parameterize the script into a function to abstract away its inner workings. This prevents the need to copy every line of code each time it’s run when only one aspect - the input filename, changes.

If a script is long, it may need to be broken down into several sub-functions, or sub-routines. A simple rule of thumb is that if a few lines of code is repeated within a project, it should be transformed into a function. Functions should then be called, and the differences in those reruns passed in as arguments or parameters.

Algorithms Organized into Classes

Data and the functions that transform it are logically related. For example, when importing a triangle of data, one may wish to produce link ratios. The input triangle and link ratios are often thought of together during the phase of data analysis.

Rather than having references to data and its transforming functions scattered throughout a file, the practitioner may wish to group them together in a single object to make it easy to mentally reason about it. This is where the concept of classes comes into play - it is an abstract representation of data and its associated functions, grouped together.

Classes Organized into Packages

Code may mature to the point where it needs to be shared with other team members to be used in other project this is when the practitioner should decide to organize the classes and their containing modules into a single package so that it may be easily transferred as a single entity.

3.1.2 Sharing Code Between Team Members

As a project increases in scope and visibility, there will be a time when the maintainer will need to share code with others or invite others to work on it.

Reasons for Sharing Code

- Promote visibility within or beyond their organization
- Need to grant access to other departments, such as IT to assist in deployment and maintenance
- The project scope has grown so that more people need to be core contributors, such as new direct reports in an expanding team
- The project has become useful and other people want to use it
- A portion of the project's "routine" can be re-used in a different project

Non-recommended Methods for Sharing Code

The following methods for sharing code are not recommended but may have been used in the past prior to the rise of repository hosting providers. Because these methods are not optimal and come with many downsides, but can still be used to share code, it may be difficult to convince IT or upper management to provide proper tools or adopt industry best practices (source control hosted on a repository provider). Since one may be told that as long as the jobs get done, we will often put up with whatever inconveniences they may carry. Our goal is to clearly articulate these downsides in the hopes that a practitioner can overcome the communicative barrier at work.

Email

In the absence of a proper version control hosting provider to share code, one may wish to distribute code to teammates via company email.

Why we shouldn't share code this way:

- Companies may forbid attachments with certain code extensions (.py, .R).

- If you shared your code and later make an update, you will need to send a new file. If your colleague also made their own update, they will need to manually inspect your new version and their version and hope that they didn't make any errors in doing so.
- Emails are reactive, if you don't check your email, you might not know a new version is available.

Shared Folders

Perhaps you've gotten your department to adopt version control and have also been provided a shared drive to place your project.

Why you shouldn't share code this way:

- Poor integration with project management software: As a project grows in size, you will want start using tools to manage the project. This includes things like issue tracking, continuous integration and continuous deployment (CI/CD), and automated testing. There are well-known commercial products such as GitHub, Azure DevOps, and Atlassian which provide such features out-of-the box, in addition to hosting code repositories on either an on-premises server or in the cloud.
- Merge conflicts: Leaving an editable set of code on a shared drives opens the possibility of having uncommitted edits if people choose to edit the shared drive version of the code rather than locally and then pushing their code to a centralized hosting provider. This type of workflow will lead to conflicting versions of code across team members, which will be extremely difficult to reconcile if the project is large.

Physical Media

At this time of writing, we would find this situation to be rare, but you never know what you might find at companies, even today. It is common for companies to lock down the ability of their employees to put data onto physical media for security reasons, such as protecting data and intellectual property. For this reason alone, attempting to share code this way is not recommended, nor practical. Another reason would be the inconvenience of physical media compared to Intra-/Internet transfer capabilities that we would hope would exist at most companies.

3.1.3 Using a Version Control System

It doesn't take long for a practitioner who is interested in code to independently arrive at the conclusion that some form of version management is needed. Even with the absence of code, practitioners who work primarily with spreadsheets will recognize the importance of preserving prior versions of their work so that they may be revisited later. For example, if one were to update a spreadsheet model, it may still be necessary to preserve a version of the model prior to the update in order to answer questions from stakeholders as to why the prior model produced the numbers it did at the time it was used.

It has been the authors' experience that actuarial departments will develop their own practices when it comes to managing prior versions of actuarial work. Such practices may involve appending spreadsheet names with some kind of suffix, e.g., "v1", "v2", etc., and inserting a sheet that includes a changelog with a verbal description of material changes between spreadsheet versions, and who was responsible for those changes.

While such practices are well-intentioned, and indeed solve many problems that actuaries encounter, they come with shortcomings and lack features that version control systems used in software have already solved and implemented.

One such shortcoming arises when the progression of complex actuarial projects is not monolithically linear. Imagine a large model embedded in a spreadsheet. The actuary decides to call this first version "v1." Later on, the actuary overwrites large portions of v1 and calls the new spreadsheet v2. At some future point the actuary may find out that they cannot make v3 without revisiting v1 while at the same time using the newer features found in v2. This creates and awkward situation where project history is no unclear without awkward workarounds. On the other hand, designing a

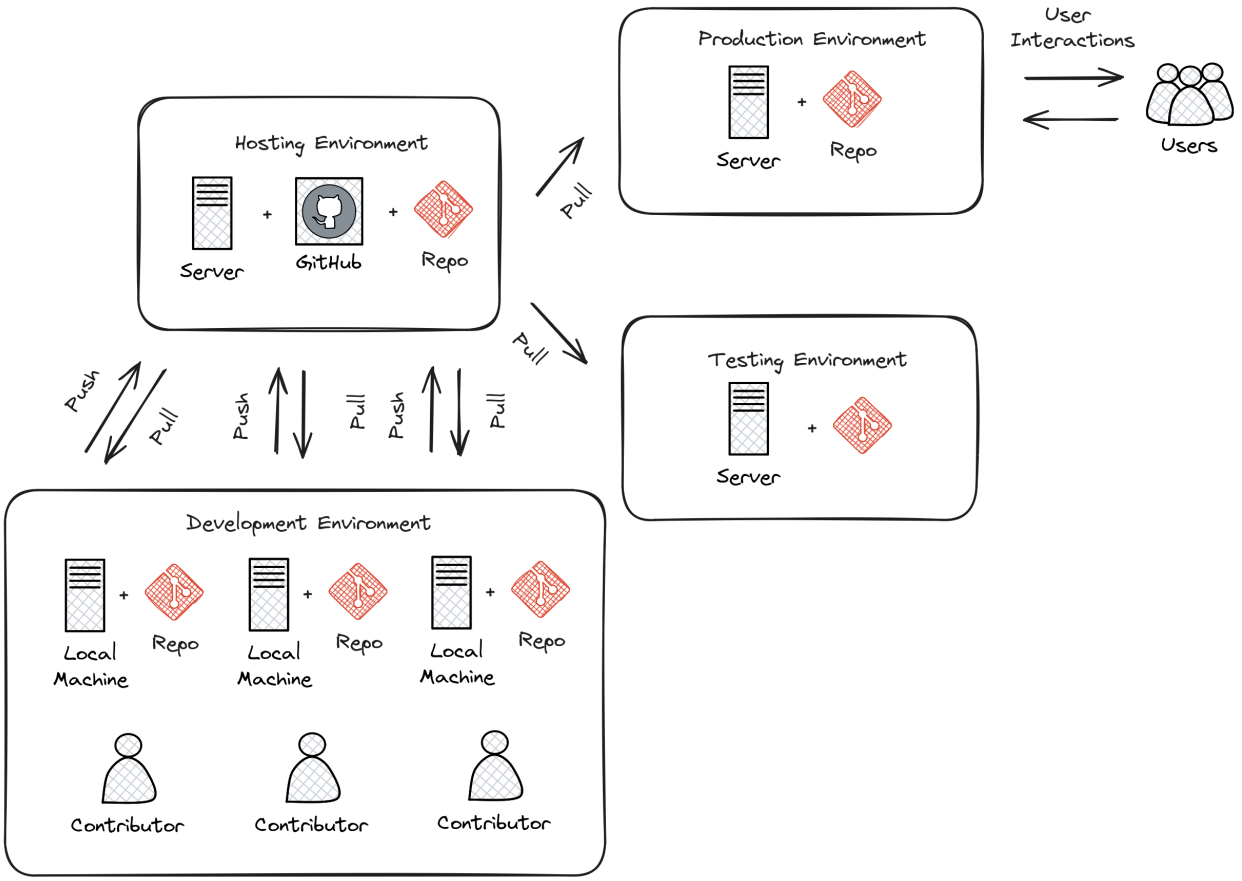
modular package with a language like R or Python, along with using the conflict resolution and checkout features of a version control system, avoids these issues.

Examples of popular version control systems are Git, Subversion, and Mercurial. Of these, we would recommend the use of Git due to the large social community that has evolved around it since its inception. For more information about Git, including some of its basic commands, please refer to the *section on Git* later in this chapter.

3.1.4 Hosting

Code repositories need to be stored in a location accessible by contributors, as well as any downstream projects and applications that depend on them. We recommend storing them with a hosting provider, which is a service that combines code storage with project management features such as issue-tracking, automated testing, and CI/CD. Repository hosting is provided by many companies, and some examples of platforms include GitHub, Azure DevOps, Bitbucket, and GitLab. Depending on the provider, the storage location may be on the cloud or on a local server if the actuarial employer wishes to store their data on premises.

3.1.5 Workflow



The above figure depicts an example workflow on how an organization may choose to maintain a reserving package or application that is version controlled as a git repository. This workflow is divided into four main environments described as follows:

- **Hosting Environment:** A server with an installation of a version control hosting provider (in this case, GitHub).

- **Development Environment:** The collection of employees or team members and their machines who are responsible for contributing to the repository.
- **Production Environment:** A server running the deployed application that users interact with.
- **Testing Environment:** A server that aims to closely replicate the production environment, used to test the application before deployment.

These environments each have their own copy (or copies) so that they can serve their purpose without interference from changes occurring in the other environments. For example, one would not want to make experimental code changes to the production server because that may lead to users experiencing bugs during important tasks. Furthermore, one would not want the contributors to query data directly from the production environment, even when they have read-only access because doing so may place unwanted load on the production server's database which may slow down or even halt the tasks being carried out by the users.

Next, we take a deeper look into each environment.

Hosting Environment

The hosting environment serves as the communicative link between the contributors, the testing environment, and the production environment. It stores the official version of the code repository, and by official we mean that this is the version that gets copied when new contributors are added to the development team, and when the production and testing environment need to fetch updated versions of the project's code.

The hosting environment is typically accessed via a web browser, although modern Integrated Development Environments (IDEs) also support integration with commonly used hosting providers. Contributors access a portal which typically offers project management features such as raising and assigning tickets to fix bugs or add new features, Kanban boards, and discussion forums. Hosting providers also have features that make it easy to look at past versions of code and to view which contributor was responsible for writing which lines.

Development Environment

The development environment is the machine or collection of machines that the contributor(s) uses to make code changes to the project. Each development machine contains its own local copy of the code. Contributors do not share code with each other directly, that is, from one development machine to another. Rather, they upload code changes to the hosting environment in a process called "pushing." Team members then receive these changes from the hosting provider to their own machines in a process by "pulling." While this style of workflow is intended to minimize conflicting copies of code, such conflicts can still happen, such as when two contributors work on the same area of the same file. In this scenario, the hosting provider's issue tracking and project management features can be used to coordinate the efforts of the team. This way, the contributors can figure out which version of the code or file to accept or reject.

Production Environment

The production environment is the server that contains the live application that users interact with. For example, in the context of a web-based reserving application, the users will access and interact with the application via their browser. Data resulting from such interactions are then stored on the production environment's database.

Testing Environment

The testing environment replicates the production environment as close as possible and allows contributors to interact with the application and access data. Data in the testing environment are populated with a periodic feed. Code changes for new features that are made in the development environment can be pushed to the test environment via the hosting environment, which allows contributors to test the behavior of those features to make sure they are working as intended.

Dependency Management

Because the maintenance of the package involves having multiple running copies used by several machines by several different people, there is a need to keep the dependencies consistent between the machines as well as isolate them from other projects or applications on those machines.

A practitioner who mostly works alone may not encounter the need to adopt dependency management practices, since many of the problems that arise concerning dependencies only manifest themselves once a project grows large enough to include multiple people, teams, and machines. Thus, practices such as virtual environments, requirements files, and containerization might be regarded as contributing to a steep learning curve and thus deprioritized for an actuary whose primary responsibility is to analyze financial data and provide strategic guidance to company leadership.

The authors recognize that when figures are due the next quarter, it may not be practical for the one person responsible for financial close to read a 250-page book on containerization or to hire someone who knows the subject to help out before the next deadline hits. These practices represent an ideal that may be subject to the practical constraint of time and resources of an organization - an ideal towards which a company strives to reach over time but continually moves due to the ever-changing business environment and even changes in the practices themselves.

As a project grows to involve more machines and people, a practitioner will eventually find the need for dependency management. One such need involves making sure that the project works on the machines of new people who are added to the project. This proves challenging as these people may have machines that are newer that of the project's original contributor. Therefore, it may contain a newer operating system as well as newer versions of prepackaged software, such as Python or R if those were used to write the project. Another need for dependency management arises when the project's dependencies may interfere with another project's dependencies when both of those projects are used on the same machine. For example, if two projects depend on different versions of Pandas (a Python library for data manipulation), this can potentially cause the project depending on the older version of Pandas to fail if it has been updated to work for the project that depends on the newer version.

For these reasons, package management practices should be adopted by the project team, and we will introduce a few of the technologies that are used. Certain practices may become mandatory in the future and can no longer be avoided. For example, newer operating systems such as Ubuntu 23.10 require the creation of virtual environments (discussed below) prior to the installation of Python packages.

Virtual Environments and Their Cousins

Python has a concept called a virtual environment, which is a folder that contains an installation of Python as well as any Python-based dependencies that a project requires. This is distinguished from the computer's base installation, which is the version Python that gets installed when the user first installs Python on their machine. Therefore, instead of using the base installation, the project uses the virtual environment instead. Different projects may have their own virtual environments separate from those of other projects, which allows different projects to run different versions of Python and Python packages without coming into conflict with each other. A project may even have multiple virtual environments so that the practitioner can test the project under different sets of dependencies (for example, when checking for compatibility of the updated dependent package).

Some Python practitioners may prefer to use an analogous environment called a Conda environment, which works similarly but is associated with the Anaconda distribution of Python, which is common amongst data analytics professionals.

R has various package management systems, notably Packrat and renv.

Containers

Dependency management extends beyond the language-specific dependency issues that a practitioner may encounter when making a project. For example, an application might not only require its own set of Python packages but also have other dependencies beyond Python, such as database drivers. These dependencies can be managed via a concept called containerization, which is similar to virtual environments discussed above, but isolates a broader set of dependencies than just language packages, such as other programs of software that the application depends on.

Currently, Docker is a popular software used for containerization.

Cross-Team Sharing

Once a package is ready to be shared with other people and teams, beyond those involved in writing the package, the practitioner needs a way to share it. This section will list some methods to help practitioners who are tempted to use the non-recommended methods, such as emails or shared folders.

PyPI/CRAN Mirroring

Organizations may prohibit uploading Python or R packages to public repositories such as PyPI or CRAN. This is because they do not want their private IP to be exposed, as the packages uploaded to these repositories are visible to the public. These public repositories are oftentimes the initial location that practitioners gravitate towards when first learning to installing packages because many books and open-source documentations use commands that set PyPI or CRAN as the default location where packages are downloaded from prior to installation. Practitioners who are not familiar with any other way to download a package may be left wondering how a team member can use the **pip** or **install.packages** command to install a package developed internally in the company.

One such way is repository mirroring. The company creates a mirror of PyPI or CRAN and then the employees can upload packages to this mirror instead of a public repository. Team members can then configure their Python and R installations to pull new packages from this mirror instead of the public repositories.

Installing from the Git Hosting Provider

Another way to share packages is to install from source by downloading the package from a hosting provider. Python and R provide ways to do this via the pip or install.packages commands, respectively. Instead of downloading from PyPI or CRAN, one can point these commands to the git hosting provider, for example, an on-prem instance of GitHub Enterprise, instead. The installation command will then download the source code, build the package, and then install it on the user's machine.

3.2 Proprietary Tools vs Open-Source Packages

The CAS conducts an annual technology survey, and the responses to the question "In which of the following tools do you plan to increase your proficiency in the next 12 months?" speak for themselves. Respondents are overwhelmingly interested in advancing skills that utilize open-source concepts; the top three choices (R, Python, and SQL) are all open-source tools. Even though the respondents might not consider if a project is open-source or not as the most important criteria, there is a certain appeal to open-source tools that make them popular and attractive.

While we recognize the attractiveness of continuing to use spreadsheet as the primary actuarial tool, there are many benefits to open-source tools.

- **Cost**: Open-source tools are cost-effective because they are typically free to use, making them accessible to individuals and organizations with limited financial budgets.
- **Permissive**: Open-source licenses often allow for extensive freedom in how the software is used, modified, and distributed, enabling diverse and innovative applications. They often have standardized licenses. We can finally leave our lawyers out of it.
- **Transparent**: Open-source tools provide transparency into their codebase, fostering trust and enabling users to verify security, functionality and privacy. Reproducibility: Because these tools are transparent, anyone will be able to reproduce and replicate the results published by someone else.
- **Flexible**: Open-source tools can be customized and adapted to suit specific needs, making them versatile for a wide range of applications and industries. In fact, most open-source tools are utilized because of their flexibility.
- **Direct access to developers**: Users of open-source tools often have direct access to the developer community, facilitating quick issue resolution and collaboration. There's no need to go through helpdesks and wait for your tickets to be rerouted by customer service agents that have little to no clue what you are asking for.
- **Speed**: Open-source tools are generally faster, especially for making complex calculations or running simulations.
- **Source of ideas and talent**: Open-source projects attract talent from around the world, fostering innovation and generating new ideas that benefit the entire community.
- **Democratic**: Open-source tools can promote an inclusive and democratic development model, where contributions and decisions are made by a global community rather than a single entity.
- **Pull requests**: Open-source projects encourage collaboration through pull requests, which are requests that can be made when an independent developer wants their code or contribution "pulled" into the main project's code repository. This allows anyone to contribute by fixing bugs or implementing new features to the project.
- **Fork**: In the event of a disagreement, for example, when a pull request is rejected, the ability to fork open-source projects allows users to create new versions of the software, promoting diversity and competition in the development ecosystem.

But of course, there are benefits to proprietary actuarial softwares, here are some of the weakness-es of open-source tools:

- **Skills**: Open-source tools can be more complex, necessitating a more significant up-front investment of time and effort to master compared to some commercial alternatives, especially if you need to customize the tools for your specific needs.
- **Familiarity**: Open-source tool projects are often foreign to many users, both in terms of its understanding the raw source code and accessing their results. This sometimes create additional friction between business units.
- **Compatibility**: Compatibility issues may arise when integrating open-source tools with existing legacy systems, potentially causing disruptions and additional development efforts.
- **Usage restrictions**: Some organizations may have cybersecurity policies or government regulatory constraints that limit the use of open-source tools, potentially hindering their adoption.
- **Limited resources**: Open-source projects may experience irregular maintenance or even abandonment due to their limited resources, this can leave users with outdated or unsupported software and potential security vulnerabilities.
- **Intellectual property & licenses**: Users sometimes need to be aware of the various open source licenses and the associated permissions and requirements and how they would impact their company intellectual property. For example, the General Public License (GPL) requires derivative works to be distributed under GPL terms, which means the derivative works must be open-sourced as well.

While there are many other elements to consider when choosing the product that suits your analytics needs, these are generally the main consideration between open-source and proprietary tools when all else being equal.

3.3 The Powerful Microsoft Excel

The authors recognize the usefulness of spread-sheet tools such as Microsoft Excel. Excel is widely used in the actuarial profession due to its versatility, accessibility, and rich feature set tailored for financial and actuarial analysis. Here is a detailed look at why actuaries might prefer Excel over other scripting tools like Python, R, or MATLAB:

1. **User-Friendly Interface**

Excel offers a graphical user interface that is highly intuitive and accessible even to those with minimal programming experience. This makes it easier for actuaries to manipulate data, perform calculations, and visualize results without the need for extensive coding knowledge.

2. **Real-Time Data Visualization**

Excel provides robust tools for creating charts and graphs that update in real time as data changes. This is particularly useful for actuaries who need to present data in a way that is easy to understand and interpret for stakeholders who may not have a technical background.

3. **Widespread Adoption and Familiarity**

Excel is a standard tool in most business environments, including insurance and financial services. This widespread adoption means that sharing files, collaborating on projects, and integrating with other business processes is streamlined, reducing the friction that might arise with less familiar or more specialized tools.

4. **Built-in Financial Functions**

Excel comes equipped with numerous built-in functions that are specifically designed for financial and actuarial calculations, such as NPV, IRR, and various amortization functions. This pre-built functionality can save time and reduce errors compared to coding similar functions from scratch in a scripting language.

5. **Pivot Tables and Data Analysis**

Actuaries often deal with large datasets. Excel's pivot tables allow for dynamic summarization and analysis of data, enabling actuaries to quickly extract insights without needing to write complex scripts.

6. **Integration with Other Microsoft Products**

Excel integrates seamlessly with other Microsoft Office products like Word and PowerPoint, making it easier to transfer data and results into reports or presentations. This compatibility is especially useful in corporate environments where Microsoft Office is the norm.

7. **Dependency by Other Teams**

Excel integrates well with many other products, and as such, many of the downstream work product demands that the actuaries feed them the result in Excel.

8. **Excel Add-Ins and Tools**

There are numerous add-ins available for Excel that enhance its capabilities, some of which are specifically designed for actuarial work. Tools like @RISK or the Excel add-in for SQL Server bring advanced statistical and stochastic modeling capabilities right into the spreadsheet.

9. **Macro and VBA Support**

For more complex or repetitive tasks, Excel supports macros and VBA (Visual Basic for Applications), allowing actuaries to automate their workflows. While VBA does require some programming skills, it is generally considered more accessible than more complex programming languages used in other statistical tools.

3.4 Git and Version Control Management

Global Information Tracker, or more commonly known as git, is a distributed version control system that can track changes within a repository, or folder. It is very commonly used by engineers or programmers to control source code versioning collaboratively. The "distributed" version control system just means that the local machines each have a local complete copy of the complete repository. That way, each team member can view, modify, and save changes to the repository without relying on a central system or impact others while they work on specific files.

There are in fact many versions of distributed source code management systems, such as Mercurial and Bazaar. These distributed version control systems help software development teams create strong workflows and hierarchies, with each developer pushing code changes to their own repository and maintainers setting a code review process to ensure only quality code is merged to the main branch, which can then be hosted on a centralized server.

3.4.1 Benefits of Distributed Version Control Systems

- **Reliable backup copies**: A distributed version control system can be seen as a collection of backups. When a team member copies a repository, they create an offline backup. If the server crashes, or is in anyway jeopardized, every local copy serves as a backup. Unlike centralized systems, these distributed systems eliminates reliance on a single backup, enhancing reliability. Although some may think multiple copies waste storage space, most development involves plain text or code files, so the storage impact is minimal.
- **Flexibility for offline work**: A distributed version control system allows development activities to be performed offline, with internet needed only for confirming final changes to the main repository on a server. Individual users have a local copy of the repository, enabling them to view history and make changes independently. This flexibility lets team members address issues promptly and efficiently. Having a local copy also speeds up common tasks, as developers don't need to wait for server to respond, thereby boosting productivity and reducing frustration.
- **Quicker feedback for experimentation**: A distributed version control system simplifies branching by keeping the entire repository history locally, enabling quick code implementation, experimentation, and review. Developers benefit from fast feedback and can experiment new features by comparing changes locally before merging. This can greatly reduce merge conflicts, and it allows for easy access to the full local history helps in identifying bugs, tracking changes, and reverting to previous versions.
- **Faster merging**: Distributed version control systems enable quick code merging without needing remote server communication. Unlike centralized systems, they support diverse branching strategies (such as for implementation of new features, or testing in different code environments). This accelerates delivery and boosts business value by allowing team members to focus on innovation instead of dealing with slow builds.

3.4.2 Git: An Example of a Distributed Version Control System

It is important to note that git is only an example of a distributed version control system, and there are many other tools of version control systems.

It is also important to know that git does not equate to GitHub, or GitLab, despite their similar names. Git is the version control tracking system, which is a tool, that allows you to track code changes, and GitHub, GitLab, or Bitbucket are the centralized server that act as hosts to your repositories. GitHub by Microsoft and GitLab by its eponymous organization. They are each spaces for developers to work on Git projects, collaborate, and share and test their work. Both repositories are constantly evolving and have attracted user bases with millions of members.

3.4.3 Commonly Used Git Commands

There are many git commands, however, a practicing actuary might be able to get by with just knowing a few. Let's explore some of the popular commands and how it might can relate to how an actuary that use a centralized folder sharing system for collaboration.

`git clone`: The `git clone` command downloads existing source code from a remote repository (e.g., GitHub), creating an identical copy of the latest project version on your local computer. This is similar to copying all the files (and folders) from your shared drive to your local machine.

`git commit -m "commit message"`: Once we reach a certain point in development, we want to "save" our changes . `git commit` is like setting a checkpoint in the development process which you can go back to later if needed. Note that a commit message is required for a commit, this message ideally should be descriptive in documenting what we have developed or changed in the source code. For example, useful messages might look like, "capping all large losses at $250k", or "setting the reinsurance quota share to 25%". This is very similar to "save as", but in the git world. In addition, each commit compares and saves all differences since the previous commit, so we can easily compare changes between commits. This is similar to how we open two files and compare them side-by-side to spot the differences.

`git branch <branch-name>`: Branches are very important when it comes to version control systems. By using branches, several developers are able to work in parallel on the same project simultaneously. We can use the `git branch` command for creating, listing and deleting branches. Think of branching as creating a road marker, which represents an independent line of development. Do you want to find out what the indication would look like with a different trend? Or how the a-priori change with a different development patterns? These are all great use cases of branches, and branches allow different analysts to work on these changes simultaneously to be combined later. Setting branches are like marking the roads where we deviate from the main road, so we know what changes are made subsequently, before these changes are merged to the main road. This is also useful for developing experimental features that are not quite ready for production yet, for example, understanding what the indications would look like with different trend assumptions or using different sets of loss development pattern.

`git checkout <name-of-your-branch>`: To work on a branch, we need to first be on that branch. We can use `git checkout` to switch from one branch to another. But we can also use `git checkout` for checking out specific files from another branch and a historical commit. Did another analyst update multiple files that we need from another branch? We can get on them by checking out the files on the branch. Or is there a new piece of code that was developed that we need to bring in? We can checkout a specific file that way.

`git status`: How do we know what the current branch looks like (since the last commit) as we work on it over time? The `git status` command gives us all the necessary information about the changes on the current branch. It will tell us if the current branch is up-to-date. Or whether there are files that had been created, modified, or deleted. Of if there are files staged (changes are tracked, and ready to be committed), unstaged (changes not tracked, not ready to be committed) or untracked (telling git to ignore these files). Or whether if there is anything to commit, push, or pull. This can help us stay organized and double-check before we commit the changes to the branch that we are on.

`git add <files>`: From `git status`, we can see a list of files that aren't tracked (or unstaged). We can add the files that we want staged for a commit by using `git add`. For traditional practitioners, this is very similar to preparing a list of new files that we are ready to deploy to a centralized shared-folder. Note that in the git world, when a brand new file is created, it is untracked by default. We will need to use `git add` to tell git that we want this new file tracked.

`git push`: After adding and committing the changes, the next thing you want to do is send these changes to the remote server. `git push` uploads all committed changes to the remote repository. Remember that unless we push the changes to the server, all changes made so far had only been for our local copy. This is similar to uploading our local files to the centralized server. Note that we don't have to push every time we make a commit.

`git pull`: `git pull` is the opposite of `git push`, where we download updates from the remote server. This is similar to downloading the remote files to our local machine. Pulling is important because git will not tell us what has changed in the remote server. We have to pull to see those changes.

`git merge`: When we have completed development in our branch and everything works fine, the final step is merging the branch with the "main" branch. This is done with the `git merge` command. `git merge` basically integrates the

experimental or developmental branch with all of its commits back to the main branch. It's important to remember that you first need to be on the specific branch that you want to merge with your feature branch. This is mainly used to consolidate different implemented changes between branches to and bring those changes together. Sometimes pushing and pulling will cause what is called a merge conflict, this is when git doesn't know how to resolve changes that are made on the local version of the file versus the remove version of the file. This can happen when code changes are made between multiple branches. While git is often smart enough to figure out what changes are newer, when it's ambiguous, it is up to the server to resolve these merge conflicts.

`git revert`: Sometimes we need to undo the changes that we've made. There are various ways to undo our changes locally or remotely, but we must carefully use these commands to avoid unwanted deletions.

There are many more git commands that might become useful for very specific use cases, but by understanding the basic commands above should allow you to have a basic understanding of how git works.

3.5 The Chainladder Ecosystem

Chainladder's open-source license, along with its transparent API, not only encourages users to enhance the quality of its existing features, but also enables them to extend the package with new features and to use it as a dependency in downstream applications. This section will highlight the nascent, but growing involvement of the actuarial community, along with new projects that have emerged since chainladder's inception.

3.5.1 Community Effort

Inspired by the R-implementation of chainladder by @mages. @jbogaardt made his initial commit on June 14, 2017. The chainladder repository has undergone significant development, amassing over 1,400 commits contributed by 19 distinct individuals. This collaborative effort has resulted in a dynamic project, with new versions released every 3 to 6 months on average. Currently, the project boasts 75 released versions, with the latest iteration, v0.8.20, launched on April 10, 2024.

The project's vitality is further evidenced by the acceptance and merging of 158 pull requests and the active engagement of the community in issue reporting. Over 250 issues have been raised collectively, with more than 230 of them successfully addressed and resolved. This responsiveness to community feedback has fostered a positive reception, as demonstrated by the 173 stars received on the @casact GitHub community, positioning chainladder as one of the most acclaimed packages in the whole actuarial community, including non-Property and Casualty practices.

The success of chainladder can be attributed in part to its presence on GitHub, a widely accessible software development platform. Leveraging the platform's visibility, actuarial practitioners and other stakeholders with an interest in open-source software have organically contributed to its growth and refinement.

The transparency afforded by GitHub, coupled with its social-media functionalities, has facilitated global discussions among developers, enabling them to collaboratively enhance the package's quality. By embedding itself within the broader open-source ecosystem, chainladder remains adaptable, continuously evolving to address the evolving needs of both business and technology.

3.5.2 Downstream Projects

Here are well known projects that depend on chainladder-python.

Tryangle by Balona and Richman

Tryangle is an automatic chainladder reserving framework. It provides scoring and optimization methods based on machine learning techniques to automatically select optimal parameters to minimize reserve prediction error. Key features include optimizing loss development factors, choosing between multiple IBNR models, or optimally blending these models.

FASLR: Free Actuarial System for Loss Reserving by Dan

FASLR is a graphical user interface that uses chainladder as its core reserving engine. It is intended to extend the functionality of chainladder by enabling the user to conduct reserve studies via interactive menus, screens, and mouse clicks. It is built on top of a relational database, adding the ability to store and retrieve the results LDF and ultimate selections.

CHAPTER

FOUR

PRACTITIONER'S GUIDE OF CHAINLADDER (PYTHON PACKAGE)

[1]:
```
# jupyter_black is a linter that improves the appearance of code. You can disregard␣
↪these lines.
import jupyter_black
jupyter_black.load()
```

```
<IPython.core.display.HTML object>
```

These tutorials are written to help a practitioner get familiar with some of the common functionalities that most reserving actuaries will perform in their day-to-day responsibilities that are provided by the `chainladder` package. We will be using datasets that come included with the package, allowing one to follow and reproduce the results as shown here.

Keep in mind that these tutorials were written to only demonstrate the functionalities of the package, and the end user should always follow all applicable laws, the Code of Professional Conduct, Actuarial Standards of Practice, and exercise their best actuarial judgement. These tutorials are not written in a way to encourage certain workflow, or recommendation for analyzing a dataset or rendering an actuarial opinion.

The tutorials assume that you have the basic understanding of commonly used actuarial terms, and can independently perform an actuarial analysis in another tool, such as Microsoft Excel or another actuarial software. Furthermore, it is assumed that you already have some familiarity with Python, and that you have the basic knowledge and experience in using some common packages that are popular in the Python community, such as `pandas` and `numpy`. In addition, we will not cover installation or setting up a development environment, as those tasks are well documented on the web and our documentation site, chainladder-python.readthedocs.io. Furthermore, as with all software, updates will impact the functionality of the package overtime, and this guide might lose its usefulness as it ages. Please refer to the documentation site for the most up-to-date guides.

If you are new to `pandas` or `numpy`, which are both extremely popular tools in the broader data science community, you may wish to visit https://pandas.pydata.org/ and https://numpy.org/ for guides and reference material.

All tutorials and exercises rely on `chainladder` v0.8.19 and `pandas` v2.2.1 or later. If you have trouble reconciling the results from your workflow to this tutorial, you should verify the versions of the packages installed in your work environment and check the release notes in case updates are issued subsequently.

Lastly, this notebook can be downloaded from GitHub, at https://github.com/genedan/cl-practitioners-guide/blob/main/chapter_4/chainladder-python%20tutorials.ipynb.

[2]:
```
import pandas as pd
import numpy as np
import chainladder as cl

print("chainladder: " + cl.__version__)
print("pandas: " + pd.__version__)
print("numpy: " + np.__version__)
```

```
chainladder: 0.8.20
pandas: 2.2.2
numpy: 1.24.3
```

4.1 Working with Triangles

4.1.1 Importing Data

Let's begin by looking at a raw triangle dataset and load it into a `pandas.DataFrame`. We'll use the data `raa`, which is available from the repository. Note that this dataset is currently in `csv` format.

[3]:
```
raa_df = pd.read_csv(
    "https://raw.githubusercontent.com/casact/chainladder-python/master/chainladder/
↪utils/data/raa.csv"
)
raa_df.head(20)
```

[3]:

	development	origin	values
0	1981	1981	5012.0
1	1982	1982	106.0
2	1983	1983	3410.0
3	1984	1984	5655.0
4	1985	1985	1092.0
5	1986	1986	1513.0
6	1987	1987	557.0
7	1988	1988	1351.0
8	1989	1989	3133.0
9	1990	1990	2063.0
10	1982	1981	8269.0
11	1983	1982	4285.0
12	1984	1983	8992.0
13	1985	1984	11555.0
14	1986	1985	9565.0
15	1987	1986	6445.0
16	1988	1987	4020.0
17	1989	1988	6947.0
18	1990	1989	5395.0
19	1983	1981	10907.0

The dataset has three columns:

- `development`: or valuation time, in this case, the valuation year.
- `origin`: or accident date, in this case, the accident year.
- `values`: the values recorded for the specific accident date at the specific valuation time (such as incurred losses, paid losses, or claim counts), in this case, these are just "values" within the triangle, and has no specific metric unit associated with them.

A table of loss experience showing total losses for a certain period (origin) at various, regular valuation dates (development), reflects the change in amounts as claims mature and emerge. Older periods in the table will have one more entry than the next youngest period, leading to the triangle shape of the data in the table or any other measure that matures over time from an origin date. Loss triangles can be used to determine loss development for a given risk.

Let's put our data into the `chainladder.Triangle` format.

[4]:
```
raa = cl.Triangle(
    data=raa_df,
    origin="origin",
    development="development",
    columns="values",
    cumulative=True,
)
raa
```

[4]:
```
         12       24       36       48       60       72       84       96       108      ␣
↪ 120
1981  5012.0   8269.0  10907.0  11805.0  13539.0  16181.0  18009.0  18608.0  18662.0 ␣
↪18834.0
1982   106.0   4285.0   5396.0  10666.0  13782.0  15599.0  15496.0  16169.0  16704.0     ␣
↪ NaN
1983  3410.0   8992.0  13873.0  16141.0  18735.0  22214.0  22863.0  23466.0      NaN     ␣
↪ NaN
1984  5655.0  11555.0  15766.0  21266.0  23425.0  26083.0  27067.0      NaN      NaN     ␣
↪ NaN
1985  1092.0   9565.0  15836.0  22169.0  25955.0  26180.0      NaN      NaN      NaN     ␣
↪ NaN
1986  1513.0   6445.0  11702.0  12935.0  15852.0      NaN      NaN      NaN      NaN     ␣
↪ NaN
1987   557.0   4020.0  10946.0  12314.0      NaN      NaN      NaN      NaN      NaN     ␣
↪ NaN
1988  1351.0   6947.0  13112.0      NaN      NaN      NaN      NaN      NaN      NaN     ␣
↪ NaN
1989  3133.0   5395.0      NaN      NaN      NaN      NaN      NaN      NaN      NaN     ␣
↪ NaN
1990  2063.0      NaN      NaN      NaN      NaN      NaN      NaN      NaN      NaN     ␣
↪ NaN
```

In the above example,

- `data` is the single `DataFrame` that contains columns representing all other arguments to the Triangle constructor. In our example, the dataset `raa_df`.
- `origin` is the representation of the accident, reporting, or more generally, the origin period of the triangle that will map to the `origin` dimension. In our example, the `origin` column.
- `development` is the representation of the development or valuation periods of the triangle that will map to the `development` dimension. In our example, the `development` column.
- `columns` is the representation of the numeric data of the triangle that will map to the `columns` dimension. If `None`, then a single 'Total' key will be generated. In our example, the `values` column.
- `columuative` is the indicator of whether the triangle is cumulative or incremental. In our example, while it is not super obvious from looking at the raw data, our triangle dataset is actually a cumulative triangle. So we'll set this to `True`.

4.1.2 Triangles Attributes

Now that we have our `Triangle` object declared within the `chainladder` package, we can get a lot of its attributes. First, let's get the latest diagonal of the Triangle with `.latest_diagonal`.

[5]:
```
raa.latest_diagonal
```

[5]:
```
         1990
1981  18834.0
1982  16704.0
1983  23466.0
1984  27067.0
1985  26180.0
1986  15852.0
1987  12314.0
1988  13112.0
1989   5395.0
1990   2063.0
```

Another attribute that is commonly used is `.link_ratio` to get the LDFs of the triangle.

[6]:
```
raa.link_ratio
```

[6]:
```
          12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
1981   1.649840  1.319023  1.082332  1.146887  1.195140  1.112972  1.033261  1.002902  1.
↪009217
1982  40.424528  1.259277  1.976649  1.292143  1.131839  0.993397  1.043431  1.033088  ␣
↪    NaN
1983   2.636950  1.542816  1.163483  1.160709  1.185695  1.029216  1.026374       NaN  ␣
↪    NaN
1984   2.043324  1.364431  1.348852  1.101524  1.113469  1.037726       NaN       NaN  ␣
↪    NaN
1985   8.759158  1.655619  1.399912  1.170779  1.008669       NaN       NaN       NaN  ␣
↪    NaN
1986   4.259749  1.815671  1.105367  1.225512       NaN       NaN       NaN       NaN  ␣
↪    NaN
1987   7.217235  2.722886  1.124977       NaN       NaN       NaN       NaN       NaN  ␣
↪    NaN
1988   5.142117  1.887433       NaN       NaN       NaN       NaN       NaN       NaN  ␣
↪    NaN
1989   1.721992       NaN       NaN       NaN       NaN       NaN       NaN       NaN  ␣
↪    NaN
```

A useful feature is the `.heatmap()` method, which highlights the highs and lows within each column.

[7]:
```
raa.link_ratio.heatmap()
```

[7]:
```
<IPython.core.display.HTML object>
```

	12	24	36	48	60	72	84	96	108	120
1988	0.4600	0.7151	0.8376	0.8987	0.9373	0.9600	0.9739	0.9811	0.9865	0.9891
1989	0.4683	0.7344	0.8406	0.9038	0.9427	0.9643	0.9765	0.9843	0.9884	
1990	0.4910	0.7394	0.8460	0.9086	0.9438	0.9650	0.9780	0.9848		
1991	0.4860	0.7370	0.8440	0.9085	0.9473	0.9672	0.9770			
1992	0.4876	0.7434	0.8518	0.9125	0.9483	0.9661				
1993	0.4958	0.7548	0.8581	0.9176	0.9512					
1994	0.5033	0.7594	0.8602	0.9158						
1995	0.5103	0.7670	0.8636							
1996	0.5220	0.7671								
1997	0.5078									

Here are some other attributes that might be useful to the practitioner:

- `is_cumulative`: boolean, returns True if the data across the development periods is cumulative, or False if it is incremental.
- `is_ultimate`: boolean, returns True if the ultimate values are projected for the triangle.
- `is_val_tri`: boolean, returns True if the development period is stated as a valuation data as opposed to an age. i.e. Schedule P styled triangle (True) or the more commonly used triangle by development age (False).
- `is_full`: boolean, returns True if the triangle has been "squared".

[8]:
```
print("Is triangle cumulative?", raa.is_cumulative)
print("Does triangle contain ultimate projections?", raa.is_ultimate)
print("Is this a valuation triangle?", raa.is_val_tri)
print('Has the triangle been "squared"?', raa.is_full)
```

```
Is triangle cumulative? True
Does triangle contain ultimate projections? False
Is this a valuation triangle? False
Has the triangle been "squared"? False
```

We can also inspect the triangle to understand its data granularity with `origin_grain` and `development_grain`. The supported `grains` are:

- Monthly: denoted with `M`.
- Quarterly: denoted with `Q`.
- Semi-annually: denoted with `S`.
- Annually: denoted with `Y`.

[9]:
```
print("Origin grain:", raa.origin_grain)
print("Development grain:", raa.development_grain)
```

```
Origin grain: Y
Development grain: Y
```

4.1.3 Manipulating Triangles

There are also useful methods to convert an cumulative triangle into an incremental one with `.cum_to_incr()`.

[10]:
```
raa.cum_to_incr()
```

[10]:

	12	24	36	48	60	72	84	96	108	120
1981	5012.0	3257.0	2638.0	898.0	1734.0	2642.0	1828.0	599.0	54.0	172.0
1982	106.0	4179.0	1111.0	5270.0	3116.0	1817.0	-103.0	673.0	535.0	NaN
1983	3410.0	5582.0	4881.0	2268.0	2594.0	3479.0	649.0	603.0	NaN	NaN
1984	5655.0	5900.0	4211.0	5500.0	2159.0	2658.0	984.0	NaN	NaN	NaN
1985	1092.0	8473.0	6271.0	6333.0	3786.0	225.0	NaN	NaN	NaN	NaN
1986	1513.0	4932.0	5257.0	1233.0	2917.0	NaN	NaN	NaN	NaN	NaN
1987	557.0	3463.0	6926.0	1368.0	NaN	NaN	NaN	NaN	NaN	NaN
1988	1351.0	5596.0	6165.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1989	3133.0	2262.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1990	2063.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

You can also convert an incremental triangle to a cumuative one with `.incr_to_cum()`.

[11]:
```
raa.cum_to_incr().incr_to_cum()
```

[11]:

	12	24	36	48	60	72	84	96	108	120
1981	5012.0	8269.0	10907.0	11805.0	13539.0	16181.0	18009.0	18608.0	18662.0	18834.0
1982	106.0	4285.0	5396.0	10666.0	13782.0	15599.0	15496.0	16169.0	16704.0	NaN
1983	3410.0	8992.0	13873.0	16141.0	18735.0	22214.0	22863.0	23466.0	NaN	NaN
1984	5655.0	11555.0	15766.0	21266.0	23425.0	26083.0	27067.0	NaN	NaN	NaN
1985	1092.0	9565.0	15836.0	22169.0	25955.0	26180.0	NaN	NaN	NaN	NaN
1986	1513.0	6445.0	11702.0	12935.0	15852.0	NaN	NaN	NaN	NaN	NaN
1987	557.0	4020.0	10946.0	12314.0	NaN	NaN	NaN	NaN	NaN	NaN
1988	1351.0	6947.0	13112.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1989	3133.0	5395.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1990	2063.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Another useful one is to convert a development triangle to a valuation triangle (Schedule P style), with `.dev_to_val()`.

[12]:
```
raa.dev_to_val()
```

[12]:
```
        1981    1982     1983     1984     1985     1986     1987     1988     1989     ␣
↪1990
1981  5012.0  8269.0  10907.0  11805.0  13539.0  16181.0  18009.0  18608.0  18662.0 ␣
↪18834.0
1982     NaN   106.0   4285.0   5396.0  10666.0  13782.0  15599.0  15496.0  16169.0 ␣
↪16704.0
1983     NaN     NaN   3410.0   8992.0  13873.0  16141.0  18735.0  22214.0  22863.0 ␣
↪23466.0
1984     NaN     NaN      NaN   5655.0  11555.0  15766.0  21266.0  23425.0  26083.0 ␣
↪27067.0
1985     NaN     NaN      NaN      NaN   1092.0   9565.0  15836.0  22169.0  25955.0 ␣
↪26180.0
1986     NaN     NaN      NaN      NaN      NaN   1513.0   6445.0  11702.0  12935.0 ␣
↪15852.0
1987     NaN     NaN      NaN      NaN      NaN      NaN    557.0   4020.0  10946.0 ␣
↪12314.0
1988     NaN     NaN      NaN      NaN      NaN      NaN      NaN   1351.0   6947.0 ␣
↪13112.0
1989     NaN     NaN      NaN      NaN      NaN      NaN      NaN      NaN   3133.0  ␣
↪5395.0
1990     NaN     NaN      NaN      NaN      NaN      NaN      NaN      NaN      NaN  ␣
↪2063.0
```

And of course, you can convert it back with `.val_to_dev()`.

[13]:
```
raa.dev_to_val().val_to_dev()
```

[13]:
```
        12       24       36       48       60       72       84       96       108     ␣
↪ 120
1981  5012.0   8269.0  10907.0  11805.0  13539.0  16181.0  18009.0  18608.0  18662.0 ␣
↪18834.0
1982   106.0   4285.0   5396.0  10666.0  13782.0  15599.0  15496.0  16169.0  16704.0     ␣
↪ NaN
1983  3410.0   8992.0  13873.0  16141.0  18735.0  22214.0  22863.0  23466.0      NaN     ␣
↪ NaN
1984  5655.0  11555.0  15766.0  21266.0  23425.0  26083.0  27067.0      NaN      NaN     ␣
↪ NaN
1985  1092.0   9565.0  15836.0  22169.0  25955.0  26180.0      NaN      NaN      NaN     ␣
↪ NaN
1986  1513.0   6445.0  11702.0  12935.0  15852.0      NaN      NaN      NaN      NaN     ␣
↪ NaN
1987   557.0   4020.0  10946.0  12314.0      NaN      NaN      NaN      NaN      NaN     ␣
↪ NaN
1988  1351.0   6947.0  13112.0      NaN      NaN      NaN      NaN      NaN      NaN     ␣
↪ NaN
1989  3133.0   5395.0      NaN      NaN      NaN      NaN      NaN      NaN      NaN     ␣
↪ NaN
1990  2063.0      NaN      NaN      NaN      NaN      NaN      NaN      NaN      NaN     ␣
↪ NaN
```

4.1.4 4-D Triangle

The triangle described so far is a two-dimensional (accident date by valuation date) structure that spans multiple cells of data. This is a useful structure for exploring individual triangles, but becomes more problematic when working with **sets** of triangles. As in a prior chapter, we discussed the usefulness of 4-D triangle objects. `Pandas` does not have a triangle `dtype`, but if it did, working with sets of triangles would be much more convenient. To facilitate working with more than one triangle at a time, `chainladder.Triangle` acts like a `pandas.DataFrame` (with an index and columns) where each cell (row x col) is an individual triangle. This structure manifests itself as a four-dimensional space.

Let's take a look at another sample dataset, `clrd`, which can be used as a 4-D triangle.

[14]:
```
clrd_df = pd.read_csv(
    "https://raw.githubusercontent.com/casact/chainladder-python/master/chainladder/
↪utils/data/clrd.csv"
)
clrd_df.head()
```

[14]:

	GRCODE	GRNAME	AccidentYear	DevelopmentYear	DevelopmentLag	IncurLoss	CumPaidLoss	BulkLoss	EarnedPremDIR	EarnedPremCeded	EarnedPremNet	Single	PostedReserve97	LOB
0	86	Allstate Ins Co Grp	1988	1988	1	367404	70571	127737	400699	5957	394742	0	281872	wkcomp
1	86	Allstate Ins Co Grp	1988	1989	2	362988	155905	60173	400699	5957	394742	0	281872	wkcomp
2	86	Allstate Ins Co Grp	1988	1990	3	347288	220744	27763	400699	5957	394742	0	281872	wkcomp
3	86	Allstate Ins Co Grp	1988	1991	4	330648	251595	15280	400699	5957	394742	0	281872	wkcomp
4	86	Allstate Ins Co Grp	1988	1992	5	354690	274156	27689	400699	5957	394742	0	281872	wkcomp

Let's load the data into the sets of triangles.

[15]:
```
clrd = cl.Triangle(
    data=clrd_df,
    origin="AccidentYear",
    development="DevelopmentYear",
    columns=[
        "IncurLoss",
        "CumPaidLoss",
        "BulkLoss",
        "EarnedPremDIR",
        "EarnedPremCeded",
        "EarnedPremNet",
    ],
    index=["GRNAME", "LOB"],
    cumulative=True,
```

(continues on next page)

(continued from previous page)

```
)
clrd
```

[15]:
```
                                          Triangle Summary
Valuation:                                        1997-12
Grain:                                               OYDY
Shape:                                   (775, 6, 10, 10)
Index:                                      [GRNAME, LOB]
Columns:    [IncurLoss, CumPaidLoss, BulkLoss, EarnedPremD...
```

In this example, `data`, `origin`, `development`, and `cumulative` are all no different from what we had done before with the `raa` dataset. But, we need to use `columns` a bit differently, and also, declare a new variable, `index`.

- `columns` is the list of "triangles" metrics that we will have. Think of this as the type of triangle metrics that we can use to describe a segment. For example, they can be Paid, Incurred, Closed Claim counts, or even exposure data. Note that while exposure data do not develop over time commonly, they can still be presented in a triangle format. In our example dataset, `IncurLoss` is financial incurred loss (not just reported, as in paid + case + IBNR), `CumPaidLoss` for paid loss, `BulkLoss` for case reserves, `EarnedPremDIR` for direct and assumed premium earned, `EarnedPremCeded` for ceded premium earned, and `EarnedPremNet` for the net premium earned.
- `index` is can be thought of as portfolio or business segments. In our example, it's a combination of `GRNAME`, the Company Name, and `LOB`, the line of business.

Since 4D structures do not fit nicely on 2D screens, we see a summary view instead that describes the structure rather than the underlying data itself.

We see 5 rows of information: * `Valuation`: the valuation date. * `Grain`: the granularity of the data, `O` stands for origin, and `D` stands for development, `OYDY` represents triangles with accident year by development year. * `Shape`: contains 4 numbers, represents the 4-D structure. * 775: the number of segments, which is the combination of `index`, that represents the data segments. In this case, it is each of the `GRNAME` and `LOB` combination. * 6: the number of triangles for each segment, which is also the columns `[IncurLoss, CumPaidLoss, BulkLoss, EarnedPremDIR, EarnedPremCeded, EarnedPremNet]`. * 10: the number of accident periods. * 10: the number of valuation periods. * This sample triangle represents a collection of 775x6 or 4,650 triangles that are themselves 10 accident years by 10 development periods. * `Index`: the segmentation level of the triangles. * `Columns`: the value types recorded in the triangles.

Now we have a 4D triangle, let's do some `pandas`-style operations.

First, let's try filtering.

[16]:
```
clrd[clrd["LOB"] == "wkcomp"]
```

[16]:
```
                                          Triangle Summary
Valuation:                                        1997-12
Grain:                                               OYDY
Shape:                                   (132, 6, 10, 10)
Index:                                      [GRNAME, LOB]
Columns:    [IncurLoss, CumPaidLoss, BulkLoss, EarnedPremD...
```

Note that only the shape changed, from `(775, 6, 10, 10)` to `(132, 6, 10, 10)`.

`pandas` has the `.loc` feature, which is available in `chainladder`, too. Let's use `.loc` to filter by index name.

[17]:
```
clrd.loc["Allstate Ins Co Grp"]
```

[17]:
```
                                          Triangle Summary
Valuation:                                        1997-12
```

(continues on next page)

(continued from previous page)

```
Grain:                                                    OYDY
Shape:                                           (2, 6, 10, 10)
Index:                                                   [LOB]
Columns:    [IncurLoss, CumPaidLoss, BulkLoss, EarnedPremD...
```

Let's see what LOBs `Allstate Ins Co Grp` writes.

[18]:
```
clrd.loc["Allstate Ins Co Grp"].index
```

[18]:
```
        LOB
0  prodliab
1    wkcomp
```

Since we have `.loc`, we must also have `.iloc`, which stands for by index location. You can even chain them together.

Let's get `Allstate Ins Co Grp`'s `prodliab` by calling `iloc[0]` and get the `CumPaidLoss` triangle.

[19]:
```
clrd.loc["Allstate Ins Co Grp"].iloc[0]["CumPaidLoss"]
```

[19]:
```
         12       24       36       48       60       72       84       96       108  ␣
↪ 120
1988  1501.0   3916.0   8834.0  17450.0  22495.0  28687.0  31311.0  32039.0  36357.0 ␣
↪36358.0
1989  1697.0   5717.0  10442.0  18125.0  23284.0  30092.0  34338.0  41094.0  41164.0  ␣
↪ NaN
1990  1373.0   4002.0  10829.0  16695.0  21788.0  25332.0  34875.0  34893.0      NaN  ␣
↪ NaN
1991  1069.0   4594.0   6920.0   9996.0  13249.0  19221.0  19256.0      NaN      NaN  ␣
↪ NaN
1992  1134.0   3068.0   5412.0   8210.0  19164.0  19187.0      NaN      NaN      NaN  ␣
↪ NaN
1993   979.0   3079.0   6407.0  16113.0  16131.0      NaN      NaN      NaN      NaN  ␣
↪ NaN
1994  1397.0   2990.0  25688.0  26030.0      NaN      NaN      NaN      NaN      NaN  ␣
↪ NaN
1995  1016.0  21935.0  22095.0      NaN      NaN      NaN      NaN      NaN      NaN  ␣
↪ NaN
1996  9852.0  10071.0      NaN      NaN      NaN      NaN      NaN      NaN      NaN  ␣
↪ NaN
1997   319.0      NaN      NaN      NaN      NaN      NaN      NaN      NaN      NaN  ␣
↪ NaN
```

`iloc[...]` actually can take in 4 parameters, which are `index`, `columns`, `origin`, and `development`. For example, if we want `Allstate Ins Co Grp`'s `prodliab` index, we can search for it first, then call the indices of `CumPaidLoss`, and origin `1990`.

[20]:
```
clrd.index[clrd.index["GRNAME"] == "Allstate Ins Co Grp"]
```

[20]:
```
                GRNAME       LOB
21  Allstate Ins Co Grp  prodliab
22  Allstate Ins Co Grp    wkcomp
```

[21]:
```
clrd
```

[21]:
```
                                            Triangle Summary
Valuation:                                           1997-12
Grain:                                                  OYDY
Shape:                                        (775, 6, 10, 10)
Index:                                           [GRNAME, LOB]
Columns:    [IncurLoss, CumPaidLoss, BulkLoss, EarnedPremD...
```

Now we are ready. The four indices are: * `index` = 21, as shown from `clrd.index[clrd.index["GRNAME"] == "Allstate Ins Co Grp"]`. * `columns` = 1, because `IncurLoss` is 0, `CumPaidLoss` is 1, etc. * `origin` = 2, because the years start in 1988, which is index 0. * `development` = `:`, we use `:` to indicate "all" development periods.

[22]:
```
clrd.iloc[21, 1, 2, :]
```

[22]:
```
          12      24       36       48       60       72       84       96   108  120
1990  1373.0  4002.0  10829.0  16695.0  21788.0  25332.0  34875.0  34893.0  NaN  NaN
```

We can also use other `pandas` filter functions. For example, getting the four `CumPaidLoss` diagonals between 1990 and 1993 with the help of `.valuation`.

[23]:
```
paid_tri = clrd.iloc[21, 1, :, :]
paid_tri[(paid_tri.valuation >= "1990") & (paid_tri.valuation < "1994")]["CumPaidLoss"]
```

[23]:
```
           12      24       36       48       60       72
1988      NaN     NaN   8834.0  17450.0  22495.0  28687.0
1989      NaN  5717.0  10442.0  18125.0  23284.0      NaN
1990   1373.0  4002.0  10829.0  16695.0      NaN      NaN
1991   1069.0  4594.0   6920.0      NaN      NaN      NaN
1992   1134.0  3068.0      NaN      NaN      NaN      NaN
1993    979.0     NaN      NaN      NaN      NaN      NaN
```

Another commonly used filter is `.development`, let's get the three columns between age 36 and age 60.

[24]:
```
paid_tri[(paid_tri.development >= 36) & (paid_tri.development <= 60)]
```

[24]:
```
            36       48       60
1988   8834.0  17450.0  22495.0
1989  10442.0  18125.0  23284.0
1990  10829.0  16695.0  21788.0
1991   6920.0   9996.0  13249.0
1992   5412.0   8210.0  19164.0
1993   6407.0  16113.0  16131.0
1994  25688.0  26030.0      NaN
1995  22095.0      NaN      NaN
1996      NaN      NaN      NaN
1997      NaN      NaN      NaN
```

With complete flexibility in the ability to slice subsets of triangles, we can use basic arithmetic to derive new triangles, which are commonly used as diagnostics to explore trends. Recall that `IncurLoss` is actually financial incurred, which includes paid + case + IBNR. Let's derive a new column, `CaseIncurLoss`, that is actually just paid + case, but without the IBNR.

[25]:
```
clrd["CaseIncurLoss"] = clrd["IncurLoss"] - clrd["BulkLoss"]
clrd["CaseIncurLoss"]
```

[25]:
```
            Triangle Summary
Valuation:           1997-12
```

(continues on next page)

(continued from previous page)

```
Grain:                  OYDY
Shape:          (775, 1, 10, 10)
Index:             [GRNAME, LOB]
Columns:        [CaseIncurLoss]
```

Note that even though `clrd["CaseIncurLoss"]` is declared as a new single variable, it actually comes with all 775 "indexes", i.e. we have 775 `clrd["CaseIncurLoss"]` triangles. But we can use `.sum()` to see the sum of them.

[26]:
```
clrd["CaseIncurLoss"].sum()
```

[26]:
```
             12          24          36          48          60          72          84 ␣
↪         96          108         120
1988   7778398.0   9872876.0  10537707.0  10973808.0  11175391.0  11265524.0  11288288.0␣
↪ 11305023.0  11323995.0  11327627.0
1989   8734319.0  10844720.0  11822136.0  12279311.0  12481505.0  12567543.0  12608487.0␣
↪ 12633539.0  12639258.0         NaN
1990   9325252.0  11913461.0  12985113.0  13459843.0  13646077.0  13718445.0  13755879.0␣
↪ 13768960.0         NaN         NaN
1991   9564486.0  12159826.0  13216383.0  13659541.0  13821032.0  13903084.0  13964163.0␣
↪         NaN         NaN         NaN
1992  10539619.0  13125930.0  14120971.0  14563964.0  14755405.0  14850140.0         NaN␣
↪         NaN         NaN         NaN
1993  11402448.0  14043343.0  15095232.0  15576086.0  15775057.0         NaN         NaN␣
↪         NaN         NaN         NaN
1994  12411107.0  15005424.0  16095699.0  16650937.0         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
1995  12686394.0  15140099.0  16223016.0         NaN         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
1996  12627293.0  14956778.0         NaN         NaN         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
1997  12705993.0         NaN         NaN         NaN         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
```

Let's look at the (sum of) Paid to (sum of) Incurred ratio triangle. Does it look like the ratios are changing over time? Using `.heatmap()` usually helps with spotting trends.

[27]:
```
(clrd["CumPaidLoss"].sum() / clrd["CaseIncurLoss"].sum()).heatmap()
```

[27]:
```
<IPython.core.display.HTML object>
```

	12	24	36	48	60	72	84	96	108	120
1988	0.4600	0.7151	0.8376	0.8987	0.9373	0.9600	0.9739	0.9811	0.9865	0.9891
1989	0.4683	0.7344	0.8406	0.9038	0.9427	0.9643	0.9765	0.9843	0.9884	
1990	0.4910	0.7394	0.8460	0.9086	0.9438	0.9650	0.9780	0.9848		
1991	0.4860	0.7370	0.8440	0.9085	0.9473	0.9672	0.9770			
1992	0.4876	0.7434	0.8518	0.9125	0.9483	0.9661				
1993	0.4958	0.7548	0.8581	0.9176	0.9512					
1994	0.5033	0.7594	0.8602	0.9158						
1995	0.5103	0.7670	0.8636							
1996	0.5220	0.7671								
1997	0.5078									

Compare the result to `(clrd["CumPaidLoss"] / clrd["CaseIncurLoss"]).sum()`, which looks odd, why is that? This is because we are summing the quotients of paid losses over incurred losses at each index.

[28]:
```
(clrd["CumPaidLoss"] / clrd["CaseIncurLoss"]).sum()
```

[28]:
```
             12          24          36          48          60          72          84 ␣
↪         96          108         120
1988  183.174954  292.568472  331.870839  398.920441  494.883976  473.435946  485.361460␣
↪ 490.862840  489.800188  491.31401
1989  200.685859  303.319078  373.150572  420.302929  463.582968  484.426264  497.129633␣
↪ 505.378356  503.292688         NaN
1990  215.342110  369.604469  378.755996  476.510210  518.929698  505.843304  513.896098␣
↪ 520.722478         NaN         NaN
1991  164.685591  327.687927  396.143934  444.785517  485.415433  503.483137  514.193116␣
↪         NaN         NaN         NaN
1992  218.254666  340.385259  411.630112  476.672321  499.918913  516.520918         NaN␣
↪         NaN         NaN         NaN
1993  231.650900  352.995731  423.386563  483.572803  517.404169         NaN         NaN␣
↪         NaN         NaN         NaN
1994  235.049595  355.944571  436.773937  498.180201         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
1995  235.370730  369.357522  445.787686         NaN         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
1996  245.990104  384.577758         NaN         NaN         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
1997  268.778113         NaN         NaN         NaN         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
```

4.1.5 Triangle Adjustments

Another adjustment we can make to the triangle is to apply a trend. We can do that by calling `chainladder.Trend()`, which is actually an estimator. It takes in a few variables: - `trends`: the list containing the annual trends expressed as a decimal. For example, 5% decrease should be stated as -0.05. - `dates`: a list-like of (start, end) dates to correspond to the trend list. - `axis` (options: [`origin`, `valuation`]): the axis on which to apply the trend.

Let's say we want a 5% trend from `1992-12-31` to `1991-01-01`. You can then call `.trend_` attribute to view the trend factors.

[29]:
```
trend_factors = (
    cl.Trend(trends=[0.05], dates=[("1992-12-31", "1991-01-01")], axis="origin")
    .fit(clrd["CumPaidLoss"].sum())
    .trend_
)
trend_factors
```

[29]:

	12	24	36	48	60	72	84	96	108	120
1988	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.0	1.0	1.0
1989	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.0	1.0	NaN
1990	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.0	NaN	NaN
1991	0.95254	0.95254	0.95254	0.95254	0.95254	0.95254	0.95254	NaN	NaN	NaN
1992	0.90709	0.90709	0.90709	0.90709	0.90709	0.90709	NaN	NaN	NaN	NaN
1993	0.90709	0.90709	0.90709	0.90709	0.90709	NaN	NaN	NaN	NaN	NaN
1994	0.90709	0.90709	0.90709	0.90709	NaN	NaN	NaN	NaN	NaN	NaN
1995	0.90709	0.90709	0.90709	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1996	0.90709	0.90709	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1997	0.90709	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Then we can apply the `trend_factors` to get our trended loss triangle.

[30]:
```
clrd["CumPaidLoss"].sum() * trend_factors
```

[30]:

	12	24	36	48	60	72	84	96	108	120
1988	3.577780e+06	7.059966e+06	8.826151e+06	9.862687e+06	1.047470e+07	1.081458e+07	1.099401e+07	11091363.0	11171590.0	11203949.0
1989	4.090680e+06	7.964702e+06	9.937520e+06	1.109859e+07	1.176649e+07	1.211879e+07	1.231163e+07	12434826.0	12492899.0	NaN
1990	4.578442e+06	8.808486e+06	1.098535e+07	1.222900e+07	1.287854e+07	1.323867e+07	1.345299e+07	13559557.0	NaN	NaN
1991	4.428126e+06	8.536430e+06	1.062486e+07	1.182063e+07	1.247069e+07	1.280926e+07	1.299494e+07	NaN	NaN	NaN
1992	4.661665e+06	8.851112e+06	1.091046e+07	1.205476e+07	1.269275e+07	1.301427e+07	NaN	NaN	NaN	NaN
1993	5.128124e+06	9.614631e+06	1.175027e+07	1.296460e+07	1.361101e+07	NaN	NaN	NaN	NaN	NaN
1994	5.666090e+06	1.033625e+07	1.255935e+07	1.383251e+07	NaN	NaN	NaN	NaN	NaN	NaN
1995	5.872359e+06	1.053327e+07	1.270842e+07	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1996	5.979174e+06	1.040787e+07	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1997	5.852451e+06	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Multipart trend is also possible, since `trends` and `dates` can accept lists.

[31]:
```
cl.Trend(
    trends=[0.05, -0.10],
    dates=[("1992-12-31", "1991-01-01"), ("1990-12-31", "1989-01-01")],
    axis="origin",
).fit(clrd["CumPaidLoss"].sum()).trend_
```

[31]:
```
           12        24        36        48        60        72        84        96      ␣
↪  108  120
1988  1.000000  1.000000  1.000000  1.000000  1.000000  1.000000  1.000000  1.000000  1.
↪000000  1.0
1989  1.110711  1.110711  1.110711  1.110711  1.110711  1.110711  1.110711  1.110711  1.
↪110711  NaN
1990  1.234034  1.234034  1.234034  1.234034  1.234034  1.234034  1.234034  1.234034    ␣
↪  NaN  NaN
1991  1.175467  1.175467  1.175467  1.175467  1.175467  1.175467  1.175467       NaN    ␣
↪  NaN  NaN
1992  1.119380  1.119380  1.119380  1.119380  1.119380  1.119380       NaN       NaN    ␣
↪  NaN  NaN
1993  1.119380  1.119380  1.119380  1.119380  1.119380       NaN       NaN       NaN    ␣
↪  NaN  NaN
1994  1.119380  1.119380  1.119380  1.119380       NaN       NaN       NaN       NaN    ␣
↪  NaN  NaN
1995  1.119380  1.119380  1.119380       NaN       NaN       NaN       NaN       NaN    ␣
↪  NaN  NaN
1996  1.119380  1.119380       NaN       NaN       NaN       NaN       NaN       NaN    ␣
↪  NaN  NaN
1997  1.119380       NaN       NaN       NaN       NaN       NaN       NaN       NaN    ␣
↪  NaN  NaN
```

`chainladder.Triangle()` objects are awesome, but what if you need to get back out to `pandas`? `.to_frame()` is a very handy method to know. It converts the `chainladder.Triangle()` object back to a `pandas.DataFrame()` object.

[32]:
```
clrd["CumPaidLoss"].sum().to_frame()
```

[32]:
```
                 12          24          36          48          60   \
1988-01-01  3577780.0   7059966.0   8826151.0   9862687.0  10474698.0
1989-01-01  4090680.0   7964702.0   9937520.0  11098588.0  11766488.0
1990-01-01  4578442.0   8808486.0  10985347.0  12229001.0  12878545.0
1991-01-01  4648756.0   8961755.0  11154244.0  12409592.0  13092037.0
1992-01-01  5139142.0   9757699.0  12027983.0  13289485.0  13992821.0
1993-01-01  5653379.0  10599423.0  12953812.0  14292516.0  15005138.0
1994-01-01  6246447.0  11394960.0  13845764.0  15249326.0         NaN
1995-01-01  6473843.0  11612151.0  14010098.0         NaN         NaN
1996-01-01  6591599.0  11473912.0         NaN         NaN         NaN
1997-01-01  6451896.0         NaN         NaN         NaN         NaN

                 72          84          96          108         120
1988-01-01  10814576.0  10994014.0  11091363.0  11171590.0  11203949.0
1989-01-01  12118790.0  12311629.0  12434826.0  12492899.0         NaN
1990-01-01  13238667.0  13452993.0  13559557.0         NaN         NaN
1991-01-01  13447481.0  13642414.0         NaN         NaN         NaN
1992-01-01  14347271.0         NaN         NaN         NaN         NaN
```

(continues on next page)

(continued from previous page)

```
1993-01-01        NaN        NaN        NaN        NaN        NaN
1994-01-01        NaN        NaN        NaN        NaN        NaN
1995-01-01        NaN        NaN        NaN        NaN        NaN
1996-01-01        NaN        NaN        NaN        NaN        NaN
1997-01-01        NaN        NaN        NaN        NaN        NaN
```

Sometimes you just want the content copied to your clipboard, you can call `.to_clipboard()` and paste the result anywhere you like, such as a spreadsheet software or another analytics tool.

[33]:
```
clrd["CumPaidLoss"].sum().to_clipboard()
```

Other data I/O methods that you might want to know are `.to_json()` and `.to_pickle`. The inverse `chainladder.read_json()` and `chainladder.read_pickle()` are also available, but we won't explore them anymore here as they are not commonly used. You may want to visit the documentation site for more info.

Now that we feel comfortable going in and out of `chainladder`, let's jump back in `chainladder` and explore some of the functions that a reserving actuary often performs when working with triangles.

4.2 Triangle Development

4.2.1 Compute Loss Development Factors

Actuaries often spend lots of time trying to fine-tune their development factors, so let's explore the ways that `chainladder` can help us do that. `chainladder.Developent()` is a very helpful estimator, and has many useful attributes such as `.ldf_` or `.cdf_`.

Let's look at the dataset that we are already familiar with `clrd["CumPaidLoss"].sum()`.

You can call `chainladder.Development()` and the `.fit()` method to the dataset, then call any attribute that it has, for example, the `.ldf_` or `.cdf_` attributes. What is being done here is that we are asking the `chainladder.Development()` estimator to set the model with all the default options, and applying such model to a triangle dataset, `clrd["CumPaidLoss"].sum()`. We are then asking the estimator to give us the `.ldf_` attribute from such dataset computed by the estimator.

[34]:
```
cl.Development().fit(clrd["CumPaidLoss"].sum()).ldf_
```

[34]:
```
         12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
(All)  1.86453  1.230856  1.109122  1.055039  1.028329  1.015751  1.008899  1.005879  1.
↪002897
```

[35]:
```
cl.Development().fit(clrd["CumPaidLoss"].sum()).cdf_
```

[35]:
```
         12-Ult    24-Ult    36-Ult    48-Ult    60-Ult  72-Ult    84-Ult    96-Ult   108-
↪Ult
(All)  2.854913  1.53117  1.243988  1.121597  1.063086  1.0338  1.017769  1.008792  1.
↪002897
```

Remember `incr_to_cum()` from earlier? It works with development factors too!

[36]:
```
cl.Development().fit(clrd["CumPaidLoss"].sum()).ldf_.incr_to_cum()
```

[36]:
```
         12-Ult    24-Ult    36-Ult    48-Ult    60-Ult  72-Ult    84-Ult    96-Ult    108-
↪Ult
(All)  2.854913  1.53117  1.243988  1.121597  1.063086  1.0338  1.017769  1.008792  1.
↪002897
```

You may have noticed that these attributes have a trailing underscore ("_"). This is scikit-learn's API convention. As its documentation states, "attributes that have been estimated from the data must always have a name ending with trailing underscore". For example, the coefficients of some regression estimator would be stored in a `coef_` attribute after `.fit()` has been called. In essence, the trailing underscore in class attributes is a scikit-learn's convention to denote that the attributes are estimated, or to denote that they are fitted attributes.

Now, one might wonder, how are the averages calculated? By default, `chainladder.Development()` calculates these averages using all data and volume-average to set the development factors. Note that since the `.average` attribute is not estimated, but just an estimator's parameter, it has no underscore following it.

[37]:
```
cl.Development().fit(clrd["CumPaidLoss"].sum()).average
```

[37]:
```
'volume'
```

Other useful averaging methods are `simple` and `regression`. The Ordinary Least Squares `regression` estimates the development factors using the regression equation $Y = mX$.

[38]:
```
simple_ldf = cl.Development(average="simple").fit(clrd["CumPaidLoss"].sum()).ldf_
simple_ldf
```

[38]:
```
          12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
(All)  1.878245  1.233252  1.109947  1.055521  1.028566  1.015797  1.008927  1.005952  1.
↪002897
```

[39]:
```
regression_ldf = (
    cl.Development(average="regression").fit(clrd["CumPaidLoss"].sum()).ldf_
)
regression_ldf
```

[39]:
```
          12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
(All)  1.851565  1.228516  1.108323  1.054585  1.028105  1.015706  1.008867  1.005806  1.
↪002897
```

Remember, we can always perform simple arithmetic with any `chainladder.Triangle()` object.

[40]:
```
simple_ldf - regression_ldf
```

[40]:
```
         12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  108-
↪120
(All)  0.02668  0.004736  0.001624  0.000936  0.000461  0.000092  0.00006  0.000146     ␣
↪NaN
```

We can also vary the `average` used for each age-to-age factors. In our dataset, we have 9 age-to-age factors, so we can supply an array of averages to use. Let's set the averages to `volume`, `simple`, `regression`, `volume`, `simple`, `regression`, `volume`, `simple`, `regression` for 12-24, 24-36, etc, respectively. You can use `["volume", "simple", "regression"] * 3` as a shortcut since the pattern is repeating.

[41]:
```
cl.Development(average=["volume", "simple", "regression"] * 3).fit(
    clrd["CumPaidLoss"].sum()
).ldf_
```

[41]:
```
         12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
(All)  1.86453  1.233252  1.108323  1.055039  1.028566  1.015706  1.008899  1.005952  1.
↪002897
```

`chainladder.Development()` estimator has another parameter `n_periods`, which is the number of most recent periods to include in the calculation of averages. The default value is `-1`, which means use all data. Let's try using only the most recent 3 periods.

[42]:
```
cl.Development(average="simple", n_periods=3).fit(clrd["CumPaidLoss"].sum()).ldf_
```

[42]:
```
          12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
(All)  1.786207  1.214568  1.103199  1.052592  1.026814  1.015533  1.008927  1.005952  1.
↪002897
```

4.2.2 Discarding Problematic Link Ratios

From time to time, there might be certain data points that we may want to exclude from the calculation of loss development factors. For example, let's say we want to discard valuation 1991 (which is a diagonal).

[43]:
```
cl.Development(drop_valuation="1991").fit(clrd["CumPaidLoss"].sum()).ldf_
```

[43]:
```
          12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
(All)  1.857588  1.228727  1.108023  1.053946  1.028329  1.015751  1.008899  1.005879  1.
↪002897
```

Or that we want to drop a specific `origin` year's `age`. For example, year `1992`'s age `24-36`'s factor. Note that only the beginning period needs to be declared.

[44]:
```
cl.Development(drop=("1992", 24)).fit(clrd["CumPaidLoss"].sum()).ldf_
```

[44]:
```
         12-24    24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
(All)  1.86453  1.23059  1.109122  1.055039  1.028329  1.015751  1.008899  1.005879  1.
↪002897
```

Or calculating the averages using the Olympic Average method, discarding the highest or lowest value, or the highest or lowest n values.

[45]:
```
cl.Development(drop_high=[True, True, False, True], drop_low=[1, 2, 0, 3]).fit(
    clrd["CumPaidLoss"].sum()
).ldf_
```

[45]:
```
         12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
(All)  1.87613  1.237933  1.109122  1.057441  1.028329  1.015751  1.008899  1.005879  1.
↪002897
```

4.2.3 Setting Development Factors Manually

Sometimes, a practitioner might be forced to use a specific development pattern, or that they might want to manually set the age-to-age factors. `chainladder.DevelopmentConstant()` does exactly that. This estimator requires that we must specify the `style` as `ldf` or `cdf`.

[46]:
```
manual_patterns = {
    12: 1.800,
    24: 1.250,
    36: 1.100,
    48: 1.050,
    60: 1.030,
    72: 1.020,
    84: 1.010,
    96: 1.005,
    108: 1.002,
}
cl.DevelopmentConstant(patterns=manual_patterns, style="ldf").fit(
    clrd["CumPaidLoss"].sum()
).ldf_
```

[46]:
```
       12-24  24-36  36-48  48-60  60-72  72-84  84-96  96-108  108-120
(All)    1.8   1.25    1.1   1.05   1.03   1.02   1.01   1.005    1.002
```

Finally, before we go further, in `scikit-learn`, there are two types of estimators: transformers and predictors. A transformer transforms the input data, X, in some ways, and a predictor predicts a new value, or values, Y, by using the input data X.

`chainladder.Development()` and `chainladder.DevelopmentConstant()` are both transformers. The returned object is a means to create development patterns, which is used to estimate ultimates. However, itself is not a IBNR estimation model, or a predictor.

In addition to `.fit()`, transformers come with the `.transform()` and `.fit_transform()` methods. These will return a `chainladder.Triangle` object, but augment it with additional information for use in a subsequent IBNR model (a predictor). For example, as we've explored previously, `drop_high` can take an array of boolean variables, indicating if the highest factor should be dropped for each of the LDF calculation.

Look at this example, calling `cl.Development().fit(clrd["CumPaidLoss"].sum()).ldf_` again actually doesn't give the transformed loss development factors that we had manually set, but using `fit_transform()` will actually modify the underlying attribute of the triangle, so we get the updated loss development factors, that will stay with the said triangle object.

[47]:
```
cl.Development().fit(clrd["CumPaidLoss"].sum()).ldf_
```

[47]:
```
         12-24     24-36     36-48     48-60     60-72     72-84     84-96    96-108  ␣
↪108-120
(All)  1.86453  1.230856  1.109122  1.055039  1.028329  1.015751  1.008899  1.005879  1.
↪002897
```

[48]:
```
transformed_paid = cl.DevelopmentConstant(
    patterns=manual_patterns, style="ldf"
).fit_transform(clrd["CumPaidLoss"].sum())
transformed_paid.ldf_
```

[48]:
```
       12-24  24-36  36-48  48-60  60-72  72-84  84-96  96-108  108-120
(All)    1.8   1.25    1.1   1.05   1.03   1.02   1.01   1.005    1.002
```

One of the major benefits of `chainladder` is that it can handle several (or all) triangles simultaneously. While this can be a convenient shorthand, all these estimators will use the same assumptions across every triangle, as expected. We can group the data by LOBs, and estimate their development patterns in one go.

[49]:
```
clrd_lob = cl.load_sample("clrd").groupby("LOB").sum()["CumPaidLoss"]
cl.Development().fit_transform(clrd_lob).ldf_
```

[49]:
```
           Triangle Summary
Valuation:          2261-12
Grain:                 OYDY
Shape:          (6, 1, 1, 9)
Index:                 [LOB]
Columns:       [CumPaidLoss]
```

Let's get `medmal`'s LDFs and see how much higher it is compared to `wkcomp`'s LDFs.

[50]:
```
clrd_lob.index
```

[50]:
```
        LOB
0   comauto
1    medmal
2   othliab
3    ppauto
4  prodliab
5    wkcomp
```

[51]:
```
(
    cl.Development().fit_transform(clrd_lob).ldf_.iloc[1, :, :, :]
    - cl.Development().fit_transform(clrd_lob).ldf_.iloc[5, :, :, :]
)
```

[51]:
```
          12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
(All)  3.654978  0.647406  0.22609  0.117203  0.052247  0.034519  0.014174  0.008478  0.
↪007935
```

4.2.4 Correlation Tests

`chainladder` also has functionality to tests for possible violation of assumptions. The two main tests that are commonly used are:

1. The `valuation_correlation` test:
   - This test tests for the assumption of independence of `origin` years. In fact, it tests for correlation across calendar periods (diagonals), and by extension, origin periods (rows).
   - An additional parameter, `total`, can be passed, depending on if we want to calculate valuation correlation in total across all origins (`True`), or for each origin separately (`False`).
   - The test uses Z-statistic.
2. The `development_correlation` test:
   - This test tests for the assumption of independence of the chain ladder method that assumes that subsequent development factors are not correlated (columns).
   - The test uses T-statistic.

[52]:
```
print(
    "Are valuation years correlated? I.e., are origins years correlated?",
    clrd["CumPaidLoss"]
    .sum()
    .valuation_correlation(p_critical=0.1, total=True)
    .z_critical.values,
)
print(
    "Are development periods correlated?",
    clrd["CumPaidLoss"].sum().development_correlation(p_critical=0.5).t_critical.values,
)
```

```
Are valuation years correlated? I.e., are origins years correlated? [[False]]
Are development periods correlated? [[ True]]
```

Now that we are done with setting up our loss development factors and are comfortable with the assumptions implied, how about adding in a tail factor?

4.3 Extending Development Patterns with Tail

4.3.1 Setting Tail Factors Manually

Often, a tail factor is necessary to supplement our loss development factors, since our triangle is too "small". `chainladder.TailConstant()` is a useful estimator that has two model parameters for us to fine-tune our tail factors.

- `tail`: The constant to apply to all LDFs within a triangle object.
- `decay`: An exponential decay constant that allows for decay over future development periods. A decay rate of 0.5 sets the development portion of each successive LDF to 50% of the previous LDF.

[53]:
```
cl.TailConstant(tail=1.005, decay=1).fit(clrd["CumPaidLoss"].sum()).cdf_
```

[53]:
```
         12-Ult    24-Ult    36-Ult    48-Ult    60-Ult    72-Ult    84-Ult    96-Ult  ␣
↪108-Ult  120-Ult   132-Ult
(All)  2.869188  1.538826  1.250208  1.127205  1.068402  1.038969  1.022858  1.013836  1.
↪007911    1.005  1.004995
```

[54]:
```
cl.TailConstant(tail=1.005, decay=0.50).fit(clrd["CumPaidLoss"].sum()).cdf_
```

[54]:
```
         12-Ult    24-Ult    36-Ult    48-Ult    60-Ult    72-Ult    84-Ult    96-Ult  ␣
↪108-Ult  120-Ult   132-Ult
(All)  2.869188  1.538826  1.250208  1.127205  1.068402  1.038969  1.022858  1.013836  1.
↪007911    1.005  1.002504
```

[55]:
```
cl.TailConstant(tail=1.005, decay=0.50).fit(
    clrd["CumPaidLoss"].sum()
).cdf_ / cl.Development().fit(clrd["CumPaidLoss"].sum()).cdf_
```

[55]:
```
       12-Ult  24-Ult  36-Ult  48-Ult  60-Ult  72-Ult  84-Ult  96-Ult  108-Ult  120-Ult ␣
↪132-Ult
(All)   1.005   1.005   1.005   1.005   1.005   1.005   1.005   1.005    1.005      NaN ␣
↪    NaN
```

4.3.2 Modeled Tail Factors

`chainladder.TailCurve()` is another class of tail transformers. Similar to the `chainladder.Development()` or `chainladder.TailConstant()` estimator, it comes with `fit`, `transform` and `fit_transform` methods. Also, like our `chainladder.Development()` estimator, you can define a tail in the absence of data or if you believe development will continue beyond your latest evaluation period.

Here, we can extend our development factors from 120 months (really 120-132) to 144 months (really 132-144).

[56]:
```
clrd["CumPaidLoss"].sum().development.max()
```

[56]:
```
120
```

[57]:
```
tail = cl.TailCurve()
tail.fit(clrd["CumPaidLoss"].sum()).ldf_
```

[57]:
```
         12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120   120-132   132-144
(All)  1.86453  1.230856  1.109122  1.055039  1.028329  1.015751  1.008899  1.005879  1.
↪002897  1.001245  1.001311
```

These extra twelve months (144 - 132, or one year) of development patterns are included, as it is typical for actuaries to track IBNR run-off over a 1-year time horizon from the valuation date. The tail extension is currently fixed at one year and there is no ability to extend it even further. However, a subsequent version of `chainladder` may address this limitation.

Curve fitting takes selected development patterns and extrapolates them using either an `exponential` or `inverse_power` fit. In most cases, the `inverse_power` produces a thicker, or a more conservative tail.

[58]:
```
exp = cl.TailCurve(curve="exponential").fit(clrd["CumPaidLoss"].sum())
exp.tail_
```

[58]:
```
        120-Ult
(All)  1.002558
```

[59]:
```
inv_power = cl.TailCurve(curve="inverse_power").fit(clrd["CumPaidLoss"].sum())
inv_power.tail_
```

[59]:
```
        120-Ult
(All)  1.026366
```

When fitting a tail, by default, all of the data will be used; however, we can specify which period of development patterns we want to begin including in the curve fitting process with `fit_period`, which takes a tuple of `start` and `stop` period. `None` can be used to ignore `start` or `stop`. For example, `(48, None)` will use development factors for age 48 and beyond. Alternatively, passing a list of booleans `[True, False, ...]` will allow for including (`True`) or excluding (`False`) any development patterns from fitting.

Patterns will also be generated for 100 periods beyond the end of the triangle by default, or we can specify how far beyond the triangle to project the tail factor to before dropping the age-to-age factor down to 1.0 using `extrap_periods`.

Note that even though we can extrapolate the curve many years beyond the end of the triangle for computational purposes, the resultant development factors will compress all ldf_ beyond one year into a single age-ultimate factor.

Let's ignore the first 3 development patterns for curve fitting but including the rest. Let's also allow our tail extrapolation to go 50 periods beyond the end of the triangle. Note that both `fit_period` and `extrap_periods` follow the `development_grain` (which is annual in our example) of the underlying triangle being fit.

[60]:
```
cl.TailCurve(fit_period=(36, None), extrap_periods=50).fit(
    clrd["CumPaidLoss"].sum()
).ldf_
```

[60]:
```
         12-24     24-36     36-48     48-60     60-72     72-84     84-96    96-108  ␣
↪108-120   120-132   132-144
(All)  1.86453  1.230856  1.109122  1.055039  1.028329  1.015751  1.008899  1.005879  1.
↪002897  1.001603  1.001996
```

Once the development patterns are set for the triangles, we are ready to fit them through some IBNR models.

4.4 IBNR Models

4.4.1 Chainladder Model

Now that we have set and transformed the triangles' loss development factors, the IBNR estimators are the final stage in analyzing reserve estimates in the `chainladder` package. These estimators have a `predict` method as opposed to a `transform` method.

The most popular method, the chainladder method, can be called with `chainladder.Chainladder()`. The basic chainladder method is entirely specified by its development pattern selections. For this reason, the `chainladder.Chainladder()` estimator takes no additional assumptions, i.e. no additional arguments is needed.

[61]:
```
cl_mod = cl.Chainladder().fit(clrd["CumPaidLoss"].sum())
cl_mod
```

[61]:
```
Chainladder()
```

All IBNR models come with common attributes. First, the `.ultimate_` attribute, which gives the ultimate estimates from using the underlying model.

[62]:
```
cl_mod.ultimate_
```

[62]:
```
              2261
1988  1.120395e+07
1989  1.252909e+07
1990  1.367877e+07
1991  1.388483e+07
1992  1.483220e+07
1993  1.595176e+07
1994  1.710361e+07
1995  1.742840e+07
1996  1.756851e+07
1997  1.841960e+07
```

Note that ultimates are measured at a valuation date way into the future. The library is extraordinarily conservative in picking this date, and sets it to December 31, 2261 by default. This is set globally and can be viewed by referencing the `ULT_VAL` constant. This is a very common maximum time value across multiple python packages and holds no additional meaning other than that is commonly chosen.

[63]:
```
cl.options.get_option("ULT_VAL")
```

[63]:
```
'2261-12-31 23:59:59.999999999'
```

If for some reason, year `2261` is not far enough out the future for you, you can change this to whatever value you like.

[64]:
```
cl.options.set_option("ULT_VAL", "3000-12-31 23:59:59.999999999")
print(cl.options.get_option("ULT_VAL"))
cl.options.set_option("ULT_VAL", "2261-12-31 23:59:59.999999999")  # Resetting it back
```

```
3000-12-31 23:59:59.999999999
```

Another commonly used attribute that is shared across all IBNR models is the `.ibnr_` attribute, which is calculated as the difference between `.ultimate_` and `.latest_diagonal`.

[65]:
```
cl_mod.ibnr_
```

[65]:
```
             2261
1988          NaN
1989  3.618623e+04
1990  1.192173e+05
1991  2.424152e+05
1992  4.849332e+05
1993  9.466176e+05
1994  1.854279e+06
1995  3.418299e+06
1996  6.094599e+06
1997  1.196771e+07
```

[66]:
```
cl_mod.ultimate_ - cl_mod.latest_diagonal
```

[66]:
```
             2261
1988          NaN
1989  3.618623e+04
1990  1.192173e+05
1991  2.424152e+05
1992  4.849332e+05
1993  9.466176e+05
1994  1.854279e+06
1995  3.418299e+06
1996  6.094599e+06
1997  1.196771e+07
```

Other attributes that actuaries might be interested in are the `.full_triangle_` and `.full_expectation_` attributes. While the `.full_expectation_` is entirely based on `.ultimate_` values and development patterns, the `.full_triangle_` is a blend of the existing triangle (i.e. not modifying the upper left portion of the triangle). These are useful for conducting an analysis of actual results vs model expectations.

[67]:
```
cl_mod.full_triangle_
```

[67]:
```
          12            24            36            48            60            72      ␣
↪       84            96           108           120           132          9999
1988  3577780.0  7.059966e+06  8.826151e+06  9.862687e+06  1.047470e+07  1.081458e+07  1.
↪099401e+07  1.109136e+07  1.117159e+07  1.120395e+07  1.120395e+07  1.120395e+07
1989  4090680.0  7.964702e+06  9.937520e+06  1.109859e+07  1.176649e+07  1.211879e+07  1.
↪231163e+07  1.243483e+07  1.249290e+07  1.252909e+07  1.252909e+07  1.252909e+07
1990  4578442.0  8.808486e+06  1.098535e+07  1.222900e+07  1.287854e+07  1.323867e+07  1.
↪345299e+07  1.355956e+07  1.363927e+07  1.367877e+07  1.367877e+07  1.367877e+07
1991  4648756.0  8.961755e+06  1.115424e+07  1.240959e+07  1.309204e+07  1.344748e+07  1.
↪364241e+07  1.376382e+07  1.384473e+07  1.388483e+07  1.388483e+07  1.388483e+07
1992  5139142.0  9.757699e+06  1.202798e+07  1.328948e+07  1.399282e+07  1.434727e+07  1.
```

(continues on next page)

(continued from previous page)

```
↪457325e+07  1.470293e+07  1.478937e+07  1.483220e+07  1.483220e+07  1.483220e+07
1993  5653379.0  1.059942e+07  1.295381e+07  1.429252e+07  1.500514e+07  1.543022e+07  1.
↪567325e+07  1.581273e+07  1.590568e+07  1.595176e+07  1.595176e+07  1.595176e+07
1994  6246447.0  1.139496e+07  1.384576e+07  1.524933e+07  1.608863e+07  1.654441e+07  1.
↪680499e+07  1.695454e+07  1.705421e+07  1.710361e+07  1.710361e+07  1.710361e+07
1995  6473843.0  1.161215e+07  1.401010e+07  1.553891e+07  1.639415e+07  1.685858e+07  1.
↪712411e+07  1.727650e+07  1.737806e+07  1.742840e+07  1.742840e+07  1.742840e+07
1996  6591599.0  1.147391e+07  1.412273e+07  1.566383e+07  1.652595e+07  1.699412e+07  1.
↪726178e+07  1.741539e+07  1.751777e+07  1.756851e+07  1.756851e+07  1.756851e+07
1997  6451896.0  1.202976e+07  1.480689e+07  1.642265e+07  1.732654e+07  1.781738e+07  1.
↪809801e+07  1.825907e+07  1.836640e+07  1.841960e+07  1.841960e+07  1.841960e+07
```

[68]:
```
cl_mod.full_expectation_
```

[68]:
```
              12            24            36            48            60            72  ␣
↪          84            96           108           120           132          9999
1988  3.924445e+06  7.317247e+06  9.006475e+06  9.989278e+06  1.053908e+07  1.083764e+07␣
↪ 1.100834e+07  1.110630e+07  1.117159e+07  1.120395e+07  1.120395e+07  1.120395e+07
1989  4.388605e+06  8.182687e+06  1.007171e+07  1.117075e+07  1.178558e+07  1.211945e+07␣
↪ 1.231034e+07  1.241989e+07  1.249290e+07  1.252909e+07  1.252909e+07  1.252909e+07
1990  4.791310e+06  8.933543e+06  1.099590e+07  1.219580e+07  1.286704e+07  1.323155e+07␣
↪ 1.343996e+07  1.355956e+07  1.363927e+07  1.367877e+07  1.367877e+07  1.367877e+07
1991  4.863486e+06  9.068117e+06  1.116154e+07  1.237951e+07  1.306087e+07  1.343087e+07␣
↪ 1.364241e+07  1.376382e+07  1.384473e+07  1.388483e+07  1.388483e+07  1.388483e+07
1992  5.195326e+06  9.686843e+06  1.192311e+07  1.322418e+07  1.395202e+07  1.434727e+07␣
↪ 1.457325e+07  1.470293e+07  1.478937e+07  1.483220e+07  1.483220e+07  1.483220e+07
1993  5.587475e+06  1.041802e+07  1.282308e+07  1.422235e+07  1.500514e+07  1.543022e+07␣
↪ 1.567325e+07  1.581273e+07  1.590568e+07  1.595176e+07  1.595176e+07  1.595176e+07
1994  5.990937e+06  1.117028e+07  1.374901e+07  1.524933e+07  1.608863e+07  1.654441e+07␣
↪ 1.680499e+07  1.695454e+07  1.705421e+07  1.710361e+07  1.710361e+07  1.710361e+07
1995  6.104703e+06  1.138240e+07  1.401010e+07  1.553891e+07  1.639415e+07  1.685858e+07␣
↪ 1.712411e+07  1.727650e+07  1.737806e+07  1.742840e+07  1.742840e+07  1.742840e+07
1996  6.153781e+06  1.147391e+07  1.412273e+07  1.566383e+07  1.652595e+07  1.699412e+07␣
↪ 1.726178e+07  1.741539e+07  1.751777e+07  1.756851e+07  1.756851e+07  1.756851e+07
1997  6.451896e+06  1.202976e+07  1.480689e+07  1.642265e+07  1.732654e+07  1.781738e+07␣
↪ 1.809801e+07  1.825907e+07  1.836640e+07  1.841960e+07  1.841960e+07  1.841960e+07
```

And of course, you can back test to see how close the actuals are compared to what the model thinks in the upper left side of our triangle.

[69]:
```
errors = cl_mod.full_expectation_ - cl_mod.full_triangle_
errors[errors.valuation < clrd.valuation_date]
```

[69]:
```
               12             24             36             48             60          ␣
↪72            84            96   108
1988  346664.831949  257280.559690  180324.113059  126591.261484   64380.299520  23064.
↪221102  14325.313154  14937.996844  NaN
1989  297924.747991  217984.817928  134187.244310   72162.484363   19089.590029    661.
↪634062  -1289.003906 -14937.996844  NaN
1990  212868.212760  125057.385629   10556.379469  -33204.396438 -11503.761841  -7114.
↪895903 -13036.309247           NaN  NaN
1991  214729.726101  106361.821952    7299.869403  -30079.911240 -31168.757261 -16610.
↪959261           NaN           NaN  NaN
```

(continues on next page)

(continued from previous page)

```
1992   56183.954712  -70855.995074 -104876.441230  -65307.098905 -40797.370446
↪NaN           NaN           NaN  NaN
1993  -65904.023231 -181406.235367 -130736.003765  -70162.339263           NaN
↪NaN           NaN           NaN  NaN
1994 -255509.812661 -224675.692141  -96755.161245           NaN           NaN
↪NaN           NaN           NaN  NaN
1995 -369139.883519 -229746.662616           NaN           NaN           NaN
↪NaN           NaN           NaN  NaN
1996 -437817.754101           NaN           NaN           NaN           NaN
↪NaN           NaN           NaN  NaN
```

With this, we can also force the IBNR run-off of future periods, let's say we want the next three years'.

[70]:
```
cl_mod.full_triangle_.dev_to_val().cum_to_incr().loc[..., "1998":"2000"]
```

[70]:
```
              1998          1999          2000
1988           NaN           NaN           NaN
1989  3.618623e+04           NaN           NaN
1990  7.971060e+04  3.950674e+04           NaN
1991  1.214019e+05  8.091135e+04  4.010186e+04
1992  2.259778e+05  1.296853e+05  8.643201e+04
1993  4.250811e+05  2.430349e+05  1.394741e+05
1994  8.393079e+05  4.557755e+05  2.605840e+05
1995  1.528808e+06  8.552461e+05  4.644305e+05
1996  2.648819e+06  1.541098e+06  8.621217e+05
1997  5.577860e+06  2.777139e+06  1.615756e+06
```

Most of the above methods from the `chainladder.Chainladder()` model apply to all other IBNR models inside `chainladder`, which we will not repeatedly demonstrate. ## Expected Loss Model

Let's look at another model, the `chainladder.ExpectedLoss()` model, which is when we know the ultimate loss already. The Expected Loss model requires one input assumption, the `aprior`, which is a scalar multiplier that will be applied to an exposure vector, that will produce an a priori ultimate estimate vector that we can use for the model.

Let's assume that our `aprior` is 80% of the scalar multiplier, and that this multiplier should be applied to `clrd["EarnedPremDIR"]`. But first, let's revisit `clrd["EarnedPremDIR"].sum()`.

[71]:
```
clrd["EarnedPremDIR"].sum()
```

[71]:
```
             12          24          36          48          60          72          84
↪         96          108         120
1988  14759891.0  14759891.0  14759891.0  14759891.0  14759891.0  14759891.0  14759891.0
↪ 14759891.0  14759891.0  14759891.0
1989  16251494.0  16251494.0  16251494.0  16251494.0  16251494.0  16251494.0  16251494.0
↪ 16251494.0  16251494.0         NaN
1990  17967080.0  17967080.0  17967080.0  17967080.0  17967080.0  17967080.0  17967080.0
↪ 17967080.0         NaN         NaN
1991  19662971.0  19662971.0  19662971.0  19662971.0  19662971.0  19662971.0  19662971.0
↪         NaN         NaN         NaN
1992  21208358.0  21208358.0  21208358.0  21208358.0  21208358.0  21208358.0         NaN
↪         NaN         NaN         NaN
1993  22951940.0  22951940.0  22951940.0  22951940.0  22951940.0         NaN         NaN
↪         NaN         NaN         NaN
1994  24758613.0  24758613.0  24758613.0  24758613.0         NaN         NaN         NaN
```

(continues on next page)

(continued from previous page)

```
↪         NaN          NaN          NaN
1995  26121518.0  26121518.0  26121518.0         NaN         NaN         NaN         NaN␣
↪         NaN          NaN          NaN
1996  26810956.0  26810956.0         NaN         NaN         NaN         NaN         NaN␣
↪         NaN          NaN          NaN
1997  27076444.0         NaN          NaN         NaN         NaN         NaN         NaN␣
↪         NaN          NaN          NaN
```

Remember that `chainladder.ExpectedLoss()` applies the scaler to a vector, and not a triangle. But we also see that the premium does not develop over time, so we can just get any vector we want. With that said, we will use the `.latest_diagonal` premium vector.

[72]:
```
cl.ExpectedLoss(apriori=0.80).fit(
    clrd["CumPaidLoss"].sum(), sample_weight=clrd["EarnedPremDIR"].latest_diagonal.sum()
).ultimate_
```

[72]:
```
            2261
1988  11807912.8
1989  13001195.2
1990  14373664.0
1991  15730376.8
1992  16966686.4
1993  18361552.0
1994  19806890.4
1995  20897214.4
1996  21448764.8
1997  21661155.2
```

A very common question that one might ask is what is the difference between:

```
cl.ExpectedLoss(apriori=0.80).fit(..., sample_weight=weight_vector)
```

versus

```
cl.ExpectedLoss(apriori=1.00).fit(..., sample_weight=weight_vector*0.80)
```

Here is where `chainladder` follows `scikit-learn`'s implementation philosophy closely. The `apriori=...` inside the estimator, `chainladder.ExpectedLoss()` is a model parameter, whereas the inputs inside `fit()` are the data that the model will be applied to. With that said, only the first code block's syntax is theoretically correct, because in the second scenario, we are technically modifying our data, while not using the model assumption (i.e. `apriori`) correctly, even though they will yield identical results.

Let's apply the same model to `clrd["IncurLoss"].sum()` just to make sure that we get the exact same ultimates.

[73]:
```
cl.ExpectedLoss(apriori=0.80).fit(
    clrd["IncurLoss"].sum(), sample_weight=clrd["EarnedPremDIR"].latest_diagonal.sum()
).ultimate_
```

[73]:
```
            2261
1988  11807912.8
1989  13001195.2
1990  14373664.0
1991  15730376.8
1992  16966686.4
```

(continues on next page)

(continued from previous page)

```
1993  18361552.0
1994  19806890.4
1995  20897214.4
1996  21448764.8
1997  21661155.2
```

4.4.2 Bornhuetter-Ferguson

The `chainladder.BornhuetterFerguson()` estimator is another method having many of the same attributes as the `chainladder.Chainladder()` estimator. It comes with one input assumption, the a priori (`apriori`), a scalar multiplier that will be applied to an exposure vector, which will produce an a priori ultimate estimate vector that we can use for the model, which works exactly like `chainladder.ExpectedLoss()`.

[74]:
```
cl.BornhuetterFerguson(apriori=0.80).fit(
    clrd["CumPaidLoss"].sum(), sample_weight=clrd["EarnedPremDIR"].latest_diagonal.sum()
).ultimate_
```

[74]:
```
              2261
1988  1.120395e+07
1989  1.253045e+07
1990  1.368483e+07
1991  1.391705e+07
1992  1.490199e+07
1993  1.609476e+07
1994  1.739668e+07
1995  1.810875e+07
1996  1.891459e+07
1997  2.052573e+07
```

4.4.3 Benktander

The `chainladder.Benktander()` method is similar to the Bornhuetter-Ferguson method, but allows for the specification of one additional assumption, `n_iters`, the number of iterations to recalculate the estimates. The Benktander method generalizes both the Bornhuetter-Ferguson and the Chainladder estimator through this assumption.

- When `n_iters = 1`, the result is equivalent to the Bornhuetter-Ferguson estimator.
- When `n_iters` is sufficiently large, the result converges to the Chainladder estimator.

(You already know that!)

Let's try with `n_iters = 1`, and verify the result is the same as Bornhuetter-Ferguson's estimate.

[75]:
```
cl.Benktander(apriori=0.80, n_iters=1).fit(
    clrd["CumPaidLoss"].sum(), sample_weight=clrd["EarnedPremDIR"].latest_diagonal.sum()
).ultimate_
```

[75]:
```
              2261
1988  1.120395e+07
1989  1.253045e+07
1990  1.368483e+07
1991  1.391705e+07
1992  1.490199e+07
```

(continues on next page)

(continued from previous page)

```
1993  1.609476e+07
1994  1.739668e+07
1995  1.810875e+07
1996  1.891459e+07
1997  2.052573e+07
```

Now `n_iters` sufficiently large, like 100, the result is (nearly) the same as Chainladder's estimate.

[76]:
```
cl.Benktander(apriori=0.80, n_iters=100).fit(
    clrd["CumPaidLoss"].sum(), sample_weight=clrd["EarnedPremDIR"].latest_diagonal.sum()
).ultimate_
```

[76]:
```
              2261
1988  1.120395e+07
1989  1.252909e+07
1990  1.367877e+07
1991  1.388483e+07
1992  1.483220e+07
1993  1.595176e+07
1994  1.710361e+07
1995  1.742840e+07
1996  1.756851e+07
1997  1.841960e+07
```

4.4.4 Cape Cod

The `chainladder.CapeCod()` method is similar to the Bornhuetter-Ferguson method, except its a priori is computed from the Triangle itself. Instead of specifying an a priori, `decay` and `trend` need to be specified.

- `decay` is the rate that gives weights to earlier origin periods, this parameter is required by the Generalized Cape Cod Method, as discussed in *Using Best Practices to Determine a Best Reserve Estimate* by Struzzieri and Hussain.
  - As the `decay` factor approaches 1 (the default value), the result approaches the traditional Cape Cod method.
  - As the decay factor approaches 0, the result approaches the Chainladder method.
- `trend` is the trend rate along the origin axis to reflect systematic inflationary impacts on the a priori.

When we fit a Cape Cod method, we can see the a priori it computes with the given decay and trend assumptions. Since it is an array of estimated parameters, this CapeCod attribute is called the `apriori_`, with a trailing underscore, since they are estimated.

[77]:
```
cl.CapeCod().fit(
    clrd["CumPaidLoss"].sum(), sample_weight=clrd["EarnedPremDIR"].latest_diagonal.sum()
).apriori_
```

[77]:
```
          2261
1988  0.706934
1989  0.706934
1990  0.706934
1991  0.706934
1992  0.706934
1993  0.706934
```

(continues on next page)

(continued from previous page)

```
1994  0.706934
1995  0.706934
1996  0.706934
1997  0.706934
```

With `decay=0`, the `.apriori_` for each origin period gets their unique apriori.

[78]:
```
cl.CapeCod(decay=0).fit(
    clrd["CumPaidLoss"].sum(), sample_weight=clrd["EarnedPremDIR"].latest_diagonal.sum()
).apriori_
```

[78]:
```
          2261
1988  0.759081
1989  0.770950
1990  0.761324
1991  0.706141
1992  0.699357
1993  0.695007
1994  0.690814
1995  0.667205
1996  0.655274
1997  0.680281
```

And we can apply `trend`, of say -1% annually, to further fine-tune the a prioris.

[79]:
```
cl.CapeCod(decay=0, trend=-0.01).fit(
    clrd["CumPaidLoss"].sum(), sample_weight=clrd["EarnedPremDIR"].latest_diagonal.sum()
).apriori_
```

[79]:
```
          2261
1988  0.693438
1989  0.711390
1990  0.709599
1991  0.664809
1992  0.665086
1993  0.667621
1994  0.670292
1995  0.653918
1996  0.648725
1997  0.680281
```

You can also view the trended aprioris without the trend with `.detrended_apriori_`.

[80]:
```
cl.CapeCod(decay=0, trend=-0.01).fit(
    clrd["CumPaidLoss"].sum(), sample_weight=clrd["EarnedPremDIR"].latest_diagonal.sum()
).detrended_apriori_
```

[80]:
```
          2261
1988  0.759081
1989  0.770950
1990  0.761324
1991  0.706141
1992  0.699357
1993  0.695007
```

(continues on next page)

(continued from previous page)

```
1994  0.690814
1995  0.667205
1996  0.655274
1997  0.680281
```

As previously mentioned, all the deterministic estimators have `.ultimate_`, `.ibnr_`, `.full_expecation_` and `.full_triangle_` attributes that are themselves `chainladder.Triangles`. These can be manipulated in a variety of ways to gain additional insights from our model.

4.4.5 Workflow Pipelines

Because of the way that the estimators are designed, they can all work seamlessly together. Let's look at the compositional nature of these estimators and how `chainladder.Pipeline()` can chain multiple operations together quickly. Recall the dataset that we've been working with, but grouping the dataset at the line of business (`LOB`) level.

[81]:
```
clrd_lob = clrd.groupby("LOB").sum()
clrd_lob
```

[81]:
```
                                                Triangle Summary
Valuation:                                              1997-12
Grain:                                                     OYDY
Shape:                                          (6, 7, 10, 10)
Index:                                                    [LOB]
Columns:    [IncurLoss, CumPaidLoss, BulkLoss, EarnedPremD...
```

We now have 6 LOBs and each with 7 columns (`IncurLoss`, `CumPaidLoss`, `BulkLoss`, `EarnedPremDIR`, `EarnedPremCeded`, `EarnedPremNet`, and `CaseIncurLoss`.

[82]:
```
clrd_lob["IncurLoss"].sum()
```

[82]:
```
             12          24          36          48          60          72          84 ␣
↪         96          108         120
1988  11644995.0  11674240.0  11653597.0  11630882.0  11593868.0  11551625.0  11463312.0␣
↪ 11420238.0  11415560.0  11396981.0
1989  13123290.0  13118789.0  13113024.0  13050144.0  12959037.0  12866709.0  12787372.0␣
↪ 12757420.0  12743440.0         NaN
1990  14776079.0  14670690.0  14479699.0  14324680.0  14183178.0  14033498.0  13948139.0␣
↪ 13925679.0         NaN         NaN
1991  15318373.0  15112547.0  14877662.0  14615540.0  14380205.0  14205778.0  14154882.0␣
↪         NaN         NaN         NaN
1992  16828857.0  16457307.0  15999385.0  15538214.0  15249286.0  15161066.0         NaN␣
↪         NaN         NaN         NaN
1993  18169370.0  17590902.0  17080187.0  16485467.0  16281774.0         NaN         NaN␣
↪         NaN         NaN         NaN
1994  19414898.0  18609089.0  17854178.0  17521037.0         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
1995  19502850.0  18668388.0  17901550.0         NaN         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
1996  19142090.0  17910743.0         NaN         NaN         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
1997  18113581.0         NaN         NaN         NaN         NaN         NaN         NaN␣
↪         NaN         NaN         NaN
```

Since we have many sets of 10x10 triangles, we can apply a "selected" set of development patterns to all of them, say we want to: - Use `volume`-weighted for the first 5 factors, and `simple` average for the next 4 factors. - Use no more than 7 periods. - Drop the 1995 valuation period. - Use the `inverse_power` curve and extrapolate 80 periods in the tail.

[83]:
```
patterns = cl.Pipeline(
    [
        (
            "dev",
            cl.Development(
                average=["volume"] * 5 + ["simple"] * 4,
                n_periods=7,
                drop_valuation="1995",
            ),
        ),
        ("tail", cl.TailCurve(curve="inverse_power", extrap_periods=80)),
    ]
)
```

Let's fit the Cape Cod model to all the `IncurLoss` triangles, `clrd_lob["IncurLoss"]`.

[84]:
```
incurred_lob = cl.CapeCod(decay=0, trend=-0.01).fit(
    X=patterns.fit_transform(clrd_lob["IncurLoss"]),
    sample_weight=clrd_lob["EarnedPremNet"].latest_diagonal,
)
```

And now we can call each LOB's incurred ultimate easily. This vastly simplified our workflow.

[85]:
```
clrd_lob.index
```

[85]:
```
        LOB
0   comauto
1    medmal
2   othliab
3    ppauto
4  prodliab
5    wkcomp
```

[86]:
```
incurred_lob.ultimate_.loc["othliab", :, :, :]
```

[86]:
```
                2261
1988  328473.000000
1989  367994.093378
1990  397774.260542
1991  485918.271360
1992  501166.803876
1993  568930.997730
1994  647890.072761
1995  610951.931101
1996  679568.821347
1997  669403.512799
```

In just a few lines of code, we have Cape Cod estimates with customized development patterns for each of the 6 LOBs that we have in our portfolio.

4.4.6 Voting Chainladder

Actuaries often don't rely on a single model, but a combination of models and they will weight in each of their pros and cons. `chainlaader.VotingChainladder()` is an ensemble meta-estimator, that works taking in a matrix of weights to form a final ultimate estimate. Let's see how it works.

First, we begin by declaring a bunch of models that we want, note that none of the data is passed in. This is the beauty of separating model and data.

[87]:
```
cl_mod = cl.Chainladder()
el_mod = cl.ExpectedLoss(apriori=0.8)
bf_mod = cl.BornhuetterFerguson(apriori=0.8)
cc_mod = cl.CapeCod(decay=1, trend=0.05)
bk_mod = cl.Benktander(apriori=0.8, n_iters=2)
```

Next, we need to prepare the `estimators` variable. This `estimators` parameter in `chainladder.VotingChainladder()` must be in an array of tuples, with (estimator_name, estimator) pairing.

[88]:
```
estimators = [
    ("cl", cl_mod),
    ("el", el_mod),
    ("bf", bf_mod),
    ("cc", cc_mod),
    ("bk", bk_mod),
]
```

Now, let's bring in the data.

Recall that some estimators (in this case, `chainladder.BornhuetterFerguson()`, `chainladder.CapeCod()`, and `chainladder.Benktander()`) require the variable `sample_weight`, let's set that up.

[89]:
```
sample_weight = clrd["EarnedPremNet"].sum().latest_diagonal
sample_weight
```

[89]:
```
            1997
1988  13861176.0
1989  15227729.0
1990  16782295.0
1991  18305690.0
1992  19802657.0
1993  21372028.0
1994  23057846.0
1995  24420839.0
1996  25020256.0
1997  25281654.0
```

Finally, we are ready to fit the data into various models, and applying the weights based on `origin` years.

[90]:
```
model_weights = np.array(
    [[0.6, 0.0, 0.2, 0.2, 0.0]] * 4
    + [[0.0, 0.0, 0.5, 0.5, 0.0]] * 3
    + [[0.0, 0.0, 0.0, 1.0, 0.0]] * 3
)

vot_mod = cl.VotingChainladder(estimators=estimators, weights=model_weights).fit(
```

(continues on next page)

(continued from previous page)

```
    clrd["IncurLoss"].sum(), sample_weight=sample_weight
)
vot_mod.ultimate_
```

[90]:
```
              2261
1988  1.139698e+07
1989  1.272397e+07
1990  1.389398e+07
1991  1.408674e+07
1992  1.500015e+07
1993  1.596923e+07
1994  1.695630e+07
1995  1.689910e+07
1996  1.628824e+07
1997  1.568684e+07
```

`chainladder.VotingChainladder()` has an additional attribute, `weights` that shows the weight matrix that is applied.

[91]:
```
vot_mod.weights
```

[91]:
```
array([[0.6, 0. , 0.2, 0.2, 0. ],
       [0.6, 0. , 0.2, 0.2, 0. ],
       [0.6, 0. , 0.2, 0.2, 0. ],
       [0.6, 0. , 0.2, 0.2, 0. ],
       [0. , 0. , 0.5, 0.5, 0. ],
       [0. , 0. , 0.5, 0.5, 0. ],
       [0. , 0. , 0.5, 0.5, 0. ],
       [0. , 0. , 0. , 1. , 0. ],
       [0. , 0. , 0. , 1. , 0. ],
       [0. , 0. , 0. , 1. , 0. ]])
```

Let's make a graph to wrap everything up, comparing all of the model's estimates vs what we have selected in `chainladder.VotingChainladder()`.

We will leave it up to the reader to clean up the axis in the graph.

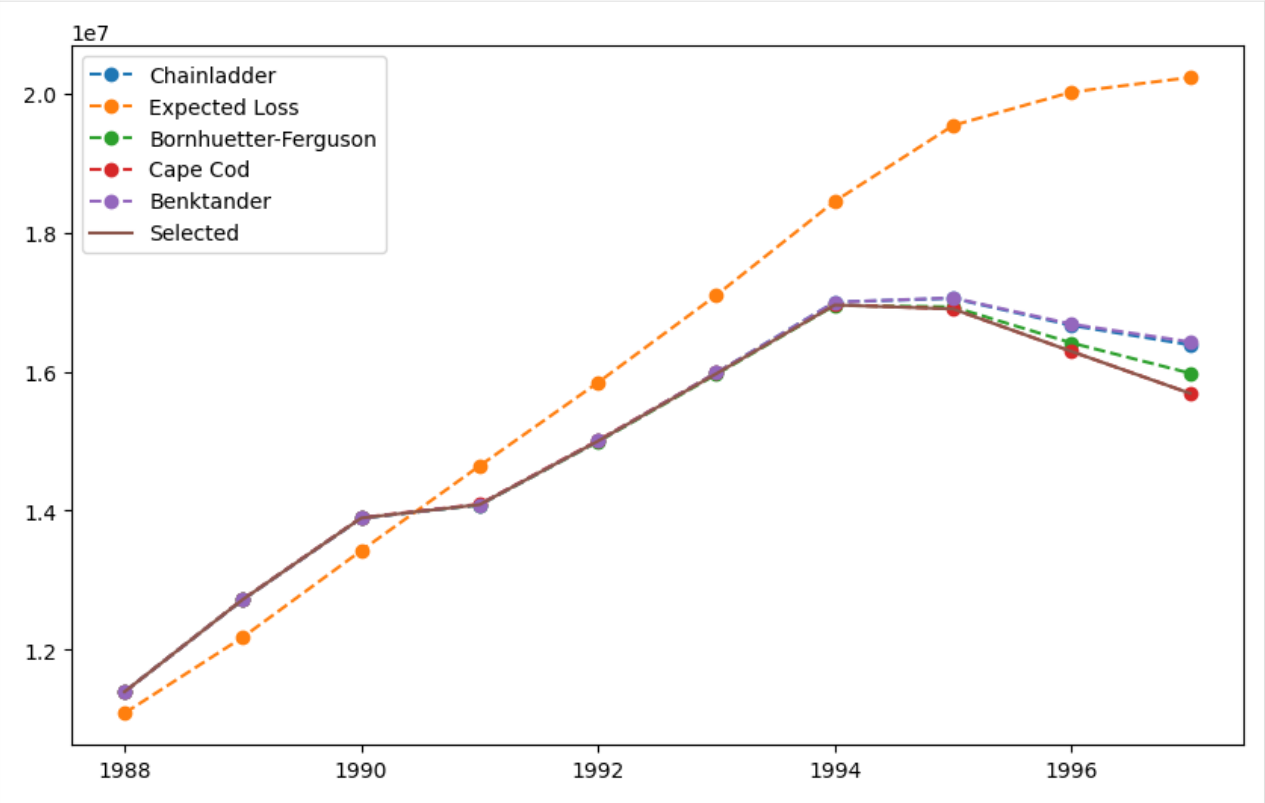
[92]:
```
loss_data = clrd["IncurLoss"].sum()

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(
    cl_mod.fit(loss_data).ultimate_.to_frame().index.year,
    cl_mod.fit(loss_data).ultimate_.to_frame(),
    label="Chainladder",
    linestyle="dashed",
    marker="o",
)
plt.plot(
    el_mod.fit(loss_data, sample_weight=sample_weight).ultimate_.to_frame().index.year,
    el_mod.fit(loss_data, sample_weight=sample_weight).ultimate_.to_frame(),
    label="Expected Loss",
    linestyle="dashed",
```

(continues on next page)

(continued from previous page)

```
    marker="o",
)
plt.plot(
    bf_mod.fit(loss_data, sample_weight=sample_weight).ultimate_.to_frame().index.year,
    bf_mod.fit(loss_data, sample_weight=sample_weight).ultimate_.to_frame(),
    label="Bornhuetter-Ferguson",
    linestyle="dashed",
    marker="o",
)
plt.plot(
    cc_mod.fit(loss_data, sample_weight=sample_weight).ultimate_.to_frame().index.year,
    cc_mod.fit(loss_data, sample_weight=sample_weight).ultimate_.to_frame(),
    label="Cape Cod",
    linestyle="dashed",
    marker="o",
)
plt.plot(
    bk_mod.fit(loss_data, sample_weight=sample_weight).ultimate_.to_frame().index.year,
    bk_mod.fit(loss_data, sample_weight=sample_weight).ultimate_.to_frame(),
    label="Benktander",
    linestyle="dashed",
    marker="o",
)
plt.plot(
    vot_mod.ultimate_.to_frame().index.year,
    vot_mod.ultimate_.to_frame(),
    label="Selected",
)
plt.legend(loc="best")
```

[92]:
```
<matplotlib.legend.Legend at 0x1692ea110>
```



Note that it is not possible to use `chainladder.VotingChainladder()` with multiple datasets. For example, blending the ultimate estimate using paid data and the ultimate estimate using incurred data. This strictly follow `scikit-learn` and its model and data implementation philosophy. However, a practitioner is of course free to perform any additional weighting outside of `chainladder`.

4.4.7 MackChainladder

Now that we know how to run deterministic models from `chainladder`, `chainladder` also has stochastic models. This is where an open-source tool like Python can excel compared to traditional spreadsheet-like analytics tools due to its computational efficiency.

Like the basic Chainladder method, the `chainladder.MackChainladder()` is entirely specified by its selected development pattern. In fact, it is the basic Chainladder model, but with extra features.

[93]:
```
cl.Chainladder().fit(clrd["CumPaidLoss"].sum()).ultimate_ == cl.MackChainladder().fit(
    clrd["CumPaidLoss"].sum()
).ultimate_
```

[93]:
```
True
```

Let's store the model result as `mack_mod` and look at its additional attributes.

[94]:
```
mack_mod = cl.MackChainladder().fit(clrd["CumPaidLoss"].sum())
mack_mod
```

[94]:
```
MackChainladder()
```

`chainladder.MackChainladder()` has the following additional fitted features that the deterministic `chainladder.Chainladder()` does not: - `full_std_err_`: The full standard error. - `total_process_risk_`: The total process

error. - `total_parameter_risk_`: The total parameter error. - `mack_std_err_`: The total prediction error by `origin` period. - `total_mack_std_err_`: The total prediction error across all `origin` periods.

[95]:
```
mack_mod.mack_std_err_
```

[95]:
```
      12             24             36             48             60             72      ␣
↪       84             96            108            120           9999
1988   NaN            NaN            NaN            NaN            NaN            NaN   ␣
↪        NaN            NaN            NaN            NaN            NaN
1989   NaN            NaN            NaN            NaN            NaN            NaN   ␣
↪        NaN            NaN            NaN    7612.690223    7612.690223
1990   NaN            NaN            NaN            NaN            NaN            NaN   ␣
↪        NaN            NaN   28691.407720   29904.992210   29904.992210
1991   NaN            NaN            NaN            NaN            NaN            NaN   ␣
↪        NaN   16178.602115   33242.040280   34341.474494   34341.474494
1992   NaN            NaN            NaN            NaN            NaN            NaN  ␣
↪13979.389430   21992.890793   37543.594231   38638.874014   38638.874014
1993   NaN            NaN            NaN            NaN            NaN   40597.438138  ␣
↪43751.707004   47552.710115   57499.871103   58393.849416   58393.849416
1994   NaN            NaN            NaN            NaN   68320.462843   82023.501630  ␣
↪84702.714537   87438.353020   94123.060189   94894.193048   94894.193048
1995   NaN            NaN            NaN   90704.313340  118024.302156  128698.338240 ␣
↪131635.018550  134122.292695  139121.818776  139872.884208  139872.884208
1996   NaN            NaN  191747.066733  231370.426260  253779.828508  264491.695092 ␣
↪269105.844778  272153.518557  275877.015450  276854.086589  276854.086589
1997   NaN  483823.825083  627242.770754  701962.818458  744029.006561  766386.657312 ␣
↪778622.006993  785791.370313  791200.371859  793559.126941  793559.126941
```

Note that these are all measures of uncertainty, and can be extremely useful when you want to run various triangle diagnostics.

Let's start by examining the link ratios underlying the triangle between age 12 and 24.

[96]:
```
clrd_first_lag = clrd[clrd.development <= 24][clrd.origin < "1997"]["CumPaidLoss"].sum()
clrd_first_lag
```

[96]:
```
             12          24
1988  3577780.0   7059966.0
1989  4090680.0   7964702.0
1990  4578442.0   8808486.0
1991  4648756.0   8961755.0
1992  5139142.0   9757699.0
1993  5653379.0  10599423.0
1994  6246447.0  11394960.0
1995  6473843.0  11612151.0
1996  6591599.0  11473912.0
```

A simple average link-ratio can be directly computed.

[97]:
```
clrd_first_lag.link_ratio.to_frame().mean().iloc[0]
```

[97]:
```
1.8782447151772095
```

Which can be verified with the `chainladder.Development()` object, ignoring the very minor rounding difference.

[98]:
```
cl.Development(average="simple").fit(clrd["CumPaidLoss"].sum()).ldf_.to_frame().iloc[
    0, 0
]
```

[98]:
```
1.8782447151772093
```

Linear Regression Framework

Mack noted that the estimate for the LDF is really just a linear regression fit. In the case of using the simple average, it is a weighted regression where the weight is $\left(\frac{1}{X}\right)^2$.

Let's take a look at the fitted coefficient and verify that this ties to the direct calculations that we made earlier. With the regression framework in hand, we can get more information about our LDF estimate than just the coefficient.

[99]:
```
import statsmodels.api as sm

y = clrd_first_lag.to_frame().values[:, 1]
x = clrd_first_lag.to_frame().values[:, 0]

sm.WLS(y, x, weights=(1 / x) ** 2).fit().summary()
```

```
/Users/kennethhsu/opt/anaconda3/envs/cl_dev/lib/python3.11/site-packages/scipy/stats/_
↪stats_py.py:1971: UserWarning: kurtosistest only valid for n>=20 ... continuing anyway,
↪ n=9
  k, _ = kurtosistest(a, axis)
```

[99]:

Dep. Variable:	y	**R-squared (uncentered):**	0.998
Model:	WLS	**Adj. R-squared (uncentered):**	0.998
Method:	Least Squares	**F-statistic:**	5323.
Date:	Fri, 14 Jun 2024	**Prob (F-statistic):**	1.39e-12
Time:	15:17:32	**Log-Likelihood:**	-128.23
No. Observations:	9	**AIC:**	258.5
Df Residuals:	8	**BIC:**	258.7
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P> |t|	[0.025	0.975]
x1	1.8782	0.026	72.958	0.000	1.819	1.938

Omnibus:	1.016	**Durbin-Watson:**	0.188
Prob(Omnibus):	0.602	**Jarque-Bera (JB):**	0.777
Skew:	-0.563	**Prob(JB):**	0.678
Kurtosis:	2.103	**Cond. No.**	1.00

Notes:
[1] R^2 is computed without centering (uncentered) since the model does not contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly specified.

By toggling the weights of our regression, we can handle the most common types of averaging used in picking loss development factors.

- For simple average, the weights are $\left(\frac{1}{X}\right)^2$
- For volume-weighted average, the weights are $\left(\frac{1}{X}\right)$
- For "regression" average, the weights are 1

Let's check to see if everything reconciles.

[100]:
```
print(
    "Simple average:",
    round(
        cl.Development(average="simple")
        .fit(clrd_first_lag)
        .ldf_.to_frame()
        .values[0, 0],
        10,
    )
    == round(sm.WLS(y, x, weights=(1 / x) ** 2).fit().params[0], 10),
)

print(
    "Volume-weighted average:",
    round(
        cl.Development(average="volume")
        .fit(clrd_first_lag)
        .ldf_.to_frame()
        .values[0, 0],
        10,
    )
    == round(sm.WLS(y, x, weights=(1 / x)).fit().params[0], 10),
)

print(
    "Regression average:",
    round(
        cl.Development(average="regression")
        .fit(clrd_first_lag)
        .ldf_.to_frame()
        .values[0, 0],
        10,
    )
    == round(sm.OLS(y, x).fit().params[0], 10),
)
```

```
Simple average: True
Volume-weighted average: True
Regression average: True
```

The regression framework is what the `chainladder.Development()` estimator uses to set development patterns. Although we discard the information in the deterministic methods, in stochastic methods, `chainladder.Development()` has two useful statistics for estimating reserve variability, both of which come from the regression framework. These statistics are `.sigma_` and `.std_err_`, and they are used by the `chainladder.MackChainladder()` estimator to determine the prediction error of our reserves.

[101]:
```
dev = cl.Development(average="simple").fit(clrd["CumPaidLoss"].sum())
dev.sigma_
```

[101]:
```
          12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  ␣
↪108-120
(All)  0.077233  0.016849  0.006631  0.004674  0.002729  0.000912  0.001045  0.001812  0.
↪000375
```

[102]:
```
dev.std_err_
```

[102]:
```
          12-24     24-36     36-48     48-60     60-72     72-84     84-96     96-108  
↪108-120
(All)  0.025744  0.005957  0.002506  0.001908  0.001221  0.000456  0.000603  0.001282  0.
↪000375
```

Since the regression framework uses the weighting method, we can easily "turn on and off" any observation we want to include or exclude using the dropping capabilities such as `drop_valuation` in the `chainladder.Development()` estimator. Dropping link ratios not only affects the `.ldf_` and `.cdf_`, but also the `.sigma_` and `.std_err_` of the estimates.

Can we eliminate the 1988 valuation from our triangle, which is identical to eliminating the first observation from our 12-24 regression fit? Let's calculate the `.std_err_` for the LDF of ages 12-24, and compare it to the value calculated using the weighted least squares regression.

[103]:
```
cl.Development(average="volume", drop_valuation="1988").fit(
    clrd["CumPaidLoss"].sum()
).std_err_.to_frame().values[0, 0]
```

[103]:
```
0.02624835706250002
```

[104]:
```
sm.WLS(y[1:], x[1:], weights=(1 / x[1:])).fit().bse[0]
```

[104]:
```
0.026248357062500012
```

With `.sigma_` and `.std_err_` in hand, Mack goes on to develop recursive formulas to estimate `.parameter_risk_` and `.process_risk_`.

[105]:
```
mack_mod.parameter_risk_
```

[105]:
```
      12             24             36             48             60             72  
↪        84             96            108            120           9999
1988   0.0       0.000000       0.000000       0.000000       0.000000       0.000000  
↪   0.000000       0.000000       0.000000       0.000000       0.000000
1989   0.0       0.000000       0.000000       0.000000       0.000000       0.000000  
↪   0.000000       0.000000       0.000000    5531.223492    5531.223492
1990   0.0       0.000000       0.000000       0.000000       0.000000       0.000000  
↪   0.000000       0.000000   17348.866209   18417.278286   18417.278286
1991   0.0       0.000000       0.000000       0.000000       0.000000       0.000000  
↪   0.000000    8417.183878   19539.800200   20532.719926   20532.719926
1992   0.0       0.000000       0.000000       0.000000       0.000000       0.000000  
↪6620.573212   11201.007805   21927.712932   22945.373563   22945.373563
1993   0.0       0.000000       0.000000       0.000000       0.000000   17897.098043  
↪19523.685261   21943.130444   29941.616526   30843.061763   30843.061763
1994   0.0       0.000000       0.000000       0.000000   28370.917272   34919.811664  
↪36282.128164   38045.113572   43989.397687   44758.317984   44758.317984
1995   0.0       0.000000       0.000000   35065.813858   46951.660897   52091.072038  
↪53480.371208   54980.972066   59558.062541   60224.091293   60224.091293
1996   0.0       0.000000   69382.590822   84683.768219   93977.300631   98629.275738  
↪100489.196199  101941.304597  104933.632333  105522.997072  105522.997072
1997   0.0  168092.794063  219313.548204  246052.431689  261386.797434  269584.906199  
↪273954.428315  276617.776201  279222.904022  280149.727111  280149.727111
```

[106]:
```
mack_mod.process_risk_
```

[106]:
```
      12             24             36             48             60             72      ␣
↪       84             96            108            120           9999
1988   0.0       0.000000       0.000000       0.000000       0.000000       0.000000   ␣
↪  0.000000       0.000000       0.000000       0.000000       0.000000
1989   0.0       0.000000       0.000000       0.000000       0.000000       0.000000   ␣
↪  0.000000       0.000000       0.000000    5230.546731    5230.546731
1990   0.0       0.000000       0.000000       0.000000       0.000000       0.000000   ␣
↪  0.000000       0.000000   22851.995935   23560.823831   23560.823831
1991   0.0       0.000000       0.000000       0.000000       0.000000       0.000000   ␣
↪  0.000000   13816.590822   26892.925652   27527.155372   27527.155372
1992   0.0       0.000000       0.000000       0.000000       0.000000       0.000000   ␣
↪12312.243466   18926.824076   30474.528271   31088.139492   31088.139492
1993   0.0       0.000000       0.000000       0.000000       0.000000   36439.619441   ␣
↪39154.023798   42187.193146   49089.049457   49583.739176   49583.739176
1994   0.0       0.000000       0.000000       0.000000   62151.240504   74219.010860   ␣
↪76538.598274   78727.599431   83211.077090   83675.568988   83675.568988
1995   0.0       0.000000       0.000000   83652.024225  108283.320223  117685.098800   ␣
↪120281.453284  122335.122138  125728.746299  126243.742675  126243.742675
1996   0.0       0.000000  178754.003289  215315.892464  235738.135064  245414.186102   ␣
↪249639.494353  252340.064358  255141.255894  255955.235051  255955.235051
1997   0.0  453685.250258  587652.159902  657426.801521  696603.262074  717406.779209   ␣
↪728835.510236  735493.496605  740292.238443  742463.614160  742463.614160
```

Assumption of Independence

The Mack model makes a lot of assumptions about independence (i.e. the covariance between random processes is 0). This means that many of the variance estimates in the `chainladder.MackChainladder()` model follow the form of $Var(A + B) = Var(A) + Var(B)$.

First, `.mack_std_err_`^{2} $= $ `.parameter_risk_`$\$^2$ $\$ +$ `.process_risk_`^{2}, the parameter risk and process risk are assumed to be independent.

[107]:
```
mack_mod.parameter_risk_**2 + mack_mod.process_risk_**2 - mack_mod.mack_std_err_**2
```

[107]:
```
      12    24        36    48        60            72            84            96      ␣
↪     108           120           9999
1988   NaN   NaN       NaN   NaN       NaN           NaN           NaN           NaN    ␣
↪      NaN           NaN           NaN
1989   NaN   NaN       NaN   NaN       NaN           NaN           NaN           NaN    ␣
↪      NaN -7.450581e-09 -7.450581e-09
1990   NaN   NaN       NaN   NaN       NaN           NaN           NaN           NaN    ␣
↪      NaN  1.192093e-07  1.192093e-07
1991   NaN   NaN       NaN   NaN       NaN           NaN           NaN -2.980232e-08  2.
↪384186e-07           NaN           NaN
1992   NaN   NaN       NaN   NaN       NaN           NaN           NaN           NaN -2.
↪384186e-07  2.384186e-07  2.384186e-07
1993   NaN   NaN       NaN   NaN       NaN  2.384186e-07           NaN           NaN  4.
↪768372e-07           NaN           NaN
1994   NaN   NaN       NaN   NaN       NaN           NaN  9.536743e-07  9.536743e-07  1.
↪907349e-06           NaN           NaN
1995   NaN   NaN       NaN   NaN       NaN           NaN -3.814697e-06  3.814697e-06    ␣
```

(continues on next page)

(continued from previous page)

```
↪       NaN  3.814697e-06  3.814697e-06
1996    NaN    NaN       NaN    NaN  0.000008  1.525879e-05           NaN           NaN -1.
↪525879e-05           NaN           NaN
1997    NaN    NaN  0.000061    NaN       NaN -1.220703e-04           NaN -1.220703e-04    ␣
↪       NaN           NaN           NaN
```

Second, `.total_process_risk_`^{2} = $:nbsphinx-math:*sum*_{origin} $ `.process_risk_`^{2}, the process risk is assumed to be independent between origins.

[108]:
```
mack_mod.total_process_risk_**2 - (mack_mod.process_risk_**2).sum(axis="origin")
```

[108]:
```
      12    24    36    48    60        72    84        96    108   120   9999
1988   NaN   NaN   NaN   NaN   NaN  0.000122   NaN -0.000122   NaN   NaN   NaN
```

Lastly, independence is also assumed to apply to the overall standard error of reserves.

[109]:
```
(mack_mod.parameter_risk_**2 + mack_mod.process_risk_**2).sum(axis=2).sum(
    axis=3
) - (mack_mod.mack_std_err_**2).sum(axis=2).sum(axis=3)
```

[109]:
```
0.0
```

This over-reliance on independence is one of the weaknesses of the `chainladder.MackChainladder()` method. Nevertheless, if the data align with this assumption, then `.total_mack_std_err_` is a reasonable estimator of reserve variability.

Mack Reserve Variability

The `.mack_std_err_` at ultimate is the reserve variability for each origin period.

[110]:
```
mack_mod.mack_std_err_[
    mack_mod.mack_std_err_.development == mack_mod.mack_std_err_.development.max()
]
```

[110]:
```
              9999
1988           NaN
1989   7612.690223
1990  29904.992210
1991  34341.474494
1992  38638.874014
1993  58393.849416
1994  94894.193048
1995 139872.884208
1996 276854.086589
1997 793559.126941
```

With the `.summary_` attribute, we can more easily look at the result of the `chainladder.MackChainladder()` model.

[111]:
```
mack_mod.summary_
```

[111]:
```
         Latest          IBNR      Ultimate   Mack Std Err
1988  11203949.0           NaN  1.120395e+07            NaN
1989  12492899.0  3.618623e+04  1.252909e+07   7612.690223
1990  13559557.0  1.192173e+05  1.367877e+07  29904.992210
```

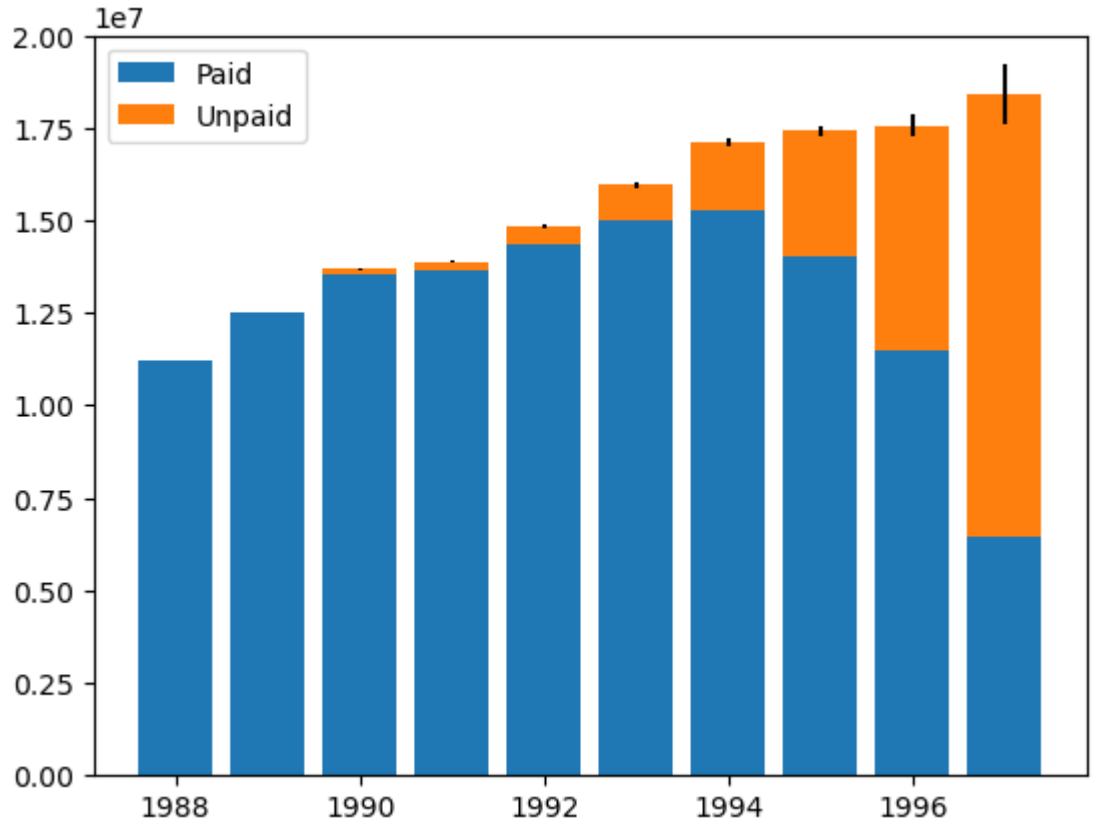
(continues on next page)

(continued from previous page)

```
1991  13642414.0  2.424152e+05  1.388483e+07   34341.474494
1992  14347271.0  4.849332e+05  1.483220e+07   38638.874014
1993  15005138.0  9.466176e+05  1.595176e+07   58393.849416
1994  15249326.0  1.854279e+06  1.710361e+07   94894.193048
1995  14010098.0  3.418299e+06  1.742840e+07  139872.884208
1996  11473912.0  6.094599e+06  1.756851e+07  276854.086589
1997   6451896.0  1.196771e+07  1.841960e+07  793559.126941
```

[112]:
```
plt.bar(
    mack_mod.summary_.to_frame().index.year,
    mack_mod.summary_.to_frame()["Latest"],
    label="Paid",
)
plt.bar(
    mack_mod.summary_.to_frame().index.year,
    mack_mod.summary_.to_frame()["IBNR"],
    bottom=mack_mod.summary_.to_frame()["Latest"],
    yerr=mack_mod.summary_.to_frame()["Mack Std Err"],
    label="Unpaid",
)
plt.legend(loc="upper left")
plt.ylim(0, 20000000)
```

[112]:
```
(0.0, 20000000.0)
```



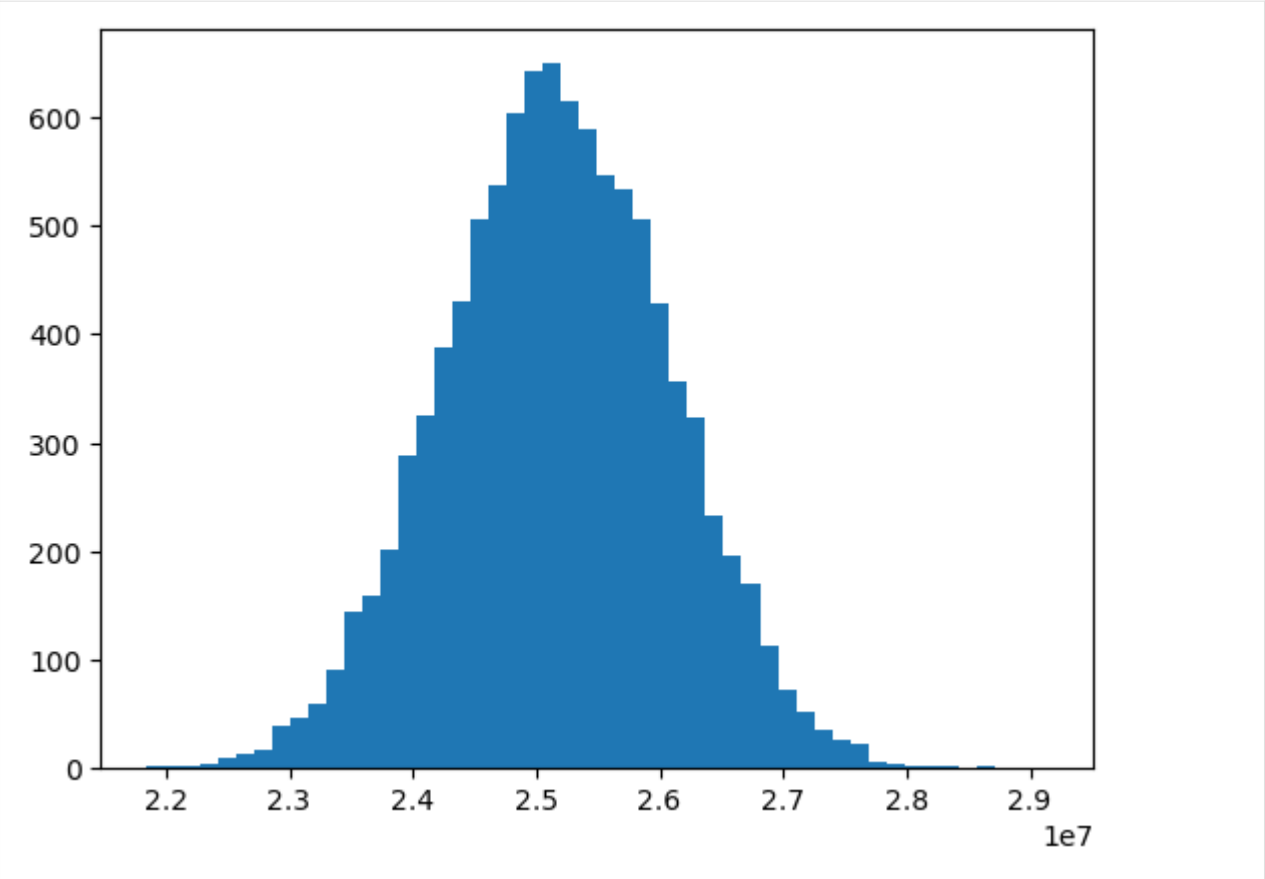
[113]:
```
mack_mod.summary_
```

[113]:
```
          Latest          IBNR      Ultimate   Mack Std Err
1988  11203949.0           NaN  1.120395e+07            NaN
1989  12492899.0  3.618623e+04  1.252909e+07    7612.690223
1990  13559557.0  1.192173e+05  1.367877e+07   29904.992210
1991  13642414.0  2.424152e+05  1.388483e+07   34341.474494
1992  14347271.0  4.849332e+05  1.483220e+07   38638.874014
1993  15005138.0  9.466176e+05  1.595176e+07   58393.849416
1994  15249326.0  1.854279e+06  1.710361e+07   94894.193048
1995  14010098.0  3.418299e+06  1.742840e+07  139872.884208
1996  11473912.0  6.094599e+06  1.756851e+07  276854.086589
1997   6451896.0  1.196771e+07  1.841960e+07  793559.126941
```

We can also simulate the (assumed) normally distributed IBNR with `np.random.normal()`.

[114]:
```
ibnr_mean = mack_mod.ibnr_.sum()
ibnr_sd = mack_mod.total_mack_std_err_.values[0, 0]
n_trials = 10000

np.random.seed(2024)
dist = np.random.normal(ibnr_mean, ibnr_sd, size=n_trials)

plt.hist(dist, bins=50)
```

[114]:
```
(array([  2.,   2.,   2.,   4.,   9.,  14.,  17.,  39.,  47.,  59.,  90.,
        144., 159., 202., 288., 325., 388., 430., 507., 537., 604., 642.,
        650., 615., 590., 546., 534., 507., 429., 357., 323., 233., 196.,
        171., 113.,  73.,  51.,  36.,  26.,  22.,   5.,   3.,   2.,   2.,
          2.,   0.,   2.,   0.,   0.,   1.]),
 array([21838558.78255578, 21984792.64512774, 22131026.5076997 ,
        22277260.37027166, 22423494.23284362, 22569728.09541558,
        22715961.95798754, 22862195.82055949, 23008429.68313145,
        23154663.54570341, 23300897.40827537, 23447131.27084733,
        23593365.13341929, 23739598.99599125, 23885832.85856321,
        24032066.72113517, 24178300.58370713, 24324534.44627909,
        24470768.30885104, 24617002.171423  , 24763236.03399497,
        24909469.89656692, 25055703.75913888, 25201937.62171084,
        25348171.4842828 , 25494405.34685476, 25640639.20942672,
        25786873.07199868, 25933106.93457064, 26079340.7971426 ,
        26225574.65971456, 26371808.52228651, 26518042.38485847,
        26664276.24743043, 26810510.11000239, 26956743.97257435,
        27102977.83514631, 27249211.69771827, 27395445.56029023,
        27541679.42286219, 27687913.28543415, 27834147.1480061 ,
        27980381.01057807, 28126614.87315002, 28272848.73572198,
        28419082.59829394, 28565316.4608659 , 28711550.32343786,
        28857784.18600982, 29004018.04858178, 29150251.91115374]),
 <BarContainer object of 50 artists>)
```



4.4.8 ODP Bootstrap Model

The `chainladder.MackChainladder()` focuses on a regression framework for determining the variability of reserve estimates. An alternative approach is to use the statistical bootstrapping, or sampling from a triangle with replacement to simulate new triangles, which is what `chainladder.BootstrapODPSample()` does.

Bootstrapping imposes less model constraints than the `chainladder.MackChainladder()`, which allows for greater applicability in different scenarios. Sampling new triangles can be accomplished through the `chainladder.BootstrapODPSample()` estimator. This estimator will take a single triangle and simulate new ones from it. To simulate new triangles randomly from an existing triangle, we specify `n_sims` with how many triangles we want to simulate, and access the `resampled_triangles_` attribute to get the simulated triangles. Notice that the shape of `resampled_triangles_` matches `n_sims` at the first index.

[115]:
```
samples = (
    cl.BootstrapODPSample(n_sims=10000)
    .fit(clrd_lob.sum()["CumPaidLoss"])
    .resampled_triangles_
)
samples
```

[115]:
```
              Triangle Summary
Valuation:              1997-12
Grain:                     OYDY
Shape:      (10000, 1, 10, 10)
```

(continues on next page)

(continued from previous page)

```
Index:                  [LOB]
Columns:          [CumPaidLoss]
```

Alternatively, we could use `chainladder.BootstrapODPSample()` to transform our triangle into a resampled set.

The notion of the ODP Bootstrap is that as our simulations approach infinity, we should expect our mean simulation to converge on the basic Chainladder estimate of reserves.

Let's apply the basic Chainladder to our original triangle and also to our simulated triangles to see whether this holds true.

[116]:
```
ibnr_cl = cl.Chainladder().fit(clrd["CumPaidLoss"].sum()).ibnr_
ibnr_bootstrap = cl.Chainladder().fit(samples).ibnr_

print(
    "Chainladder's IBNR estimate:",
    ibnr_cl.sum(),
)
print(
    "BootstrapODPSample's mean IBNR estimate:",
    ibnr_bootstrap.sum("origin").mean(),
)
print("Difference $: $", ibnr_cl.sum() - ibnr_bootstrap.sum("origin").mean())
print(
    "Difference %:",
    abs(ibnr_cl.sum() - ibnr_bootstrap.sum("origin").mean()) / ibnr_cl.sum() * 100,
    "%",
)
```

```
/Users/kennethhsu/Documents/GitHub/chainladder-python/chainladder/tails/base.py:120:␣
↪RuntimeWarning: overflow encountered in exp
  sigma_ = xp.exp(time_pd * reg.slope_ + reg.intercept_)
/Users/kennethhsu/Documents/GitHub/chainladder-python/chainladder/tails/base.py:124:␣
↪RuntimeWarning: overflow encountered in exp
  std_err_ = xp.exp(time_pd * reg.slope_ + reg.intercept_)
/Users/kennethhsu/Documents/GitHub/chainladder-python/chainladder/tails/base.py:127:␣
↪RuntimeWarning: invalid value encountered in multiply
  sigma_ = sigma_ * 0
/Users/kennethhsu/Documents/GitHub/chainladder-python/chainladder/tails/base.py:128:␣
↪RuntimeWarning: invalid value encountered in multiply
  std_err_ = std_err_* 0
```

```
Chainladder's IBNR estimate: 25164252.77609498
BootstrapODPSample's mean IBNR estimate: 25175248.530119807
Difference $: $ -10995.75402482599
Difference %: 0.04369592899365369 %
```

The difference is small, as expected.

The last thing we will do, is to compare `chainladder.MackChainladder()` against `chainladder.BootstrapODPSample()`. Recall the Mack model is stored as `mack_mod`. We can get its total `ibnr_` and its standard error with `.total_mack_std_err_`. We then simulate 10,000 trials to form a Mack stimulated total IBNR.

[117]:
```
ibnr_mean = mack_mod.ibnr_.sum()
ibnr_sd = mack_mod.total_mack_std_err_.values[0, 0]
```

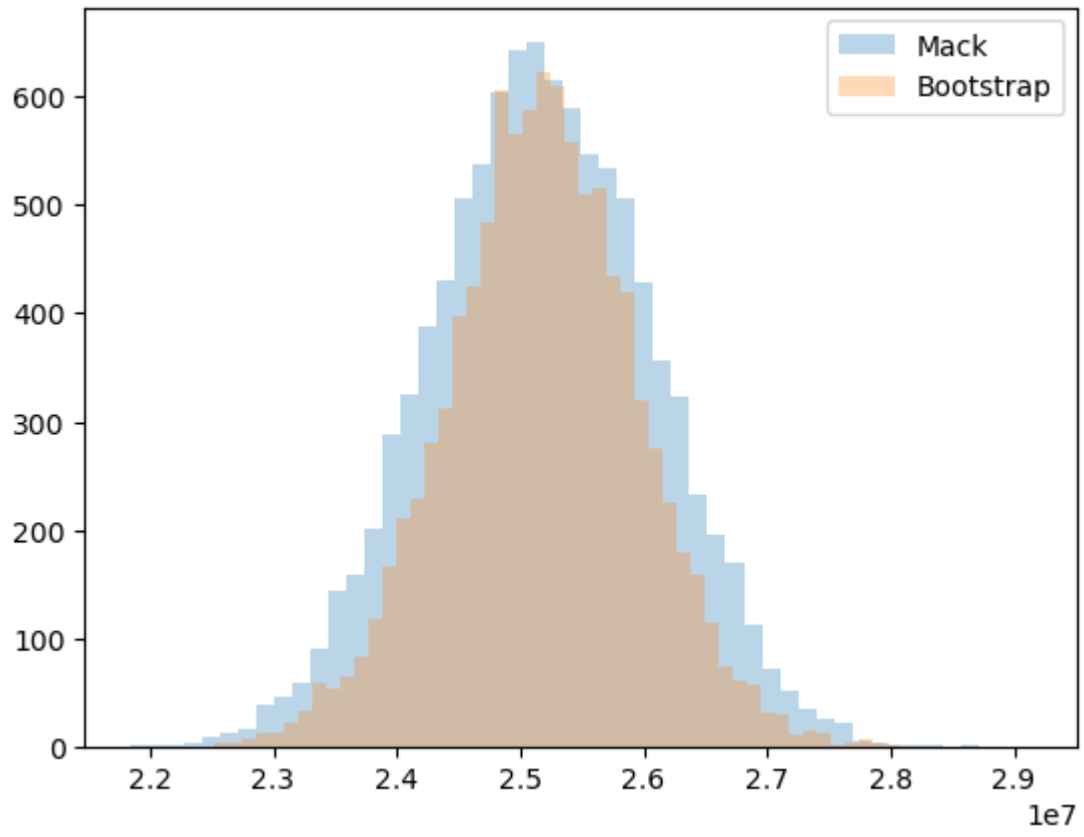
(continues on next page)

(continued from previous page)

```
np.random.seed(2024)
dist = np.random.normal(ibnr_mean, ibnr_sd, size=10000)
```

[118]:
```
plt.hist(dist, bins=50, label="Mack", alpha=0.3)
plt.hist(
    ibnr_bootstrap.sum("origin").to_frame(),
    bins=50,
    label="Bootstrap",
    alpha=0.3,
)
plt.legend(loc="upper right")
```

[118]:
```
<matplotlib.legend.Legend at 0x17765fa50>
```



4.5 Data Preparation Considerations

Even though data preparation is probably the first thing that the practicing actuary needs to perform, we decided to put the discussion here as it will be the piece that will vary the most between users. Here we offer a couple of ideas and suggestions on things that you might want to consider.

4.5.1 Triangle Data Format

One of the most commonly asked questions is that if the data needs to be in the tabular long format as opposed to the more commonly used triangle format when we are loading the data for use.

Unfortunately, the `chainladder` package requires the data to be in long form.

Suppose you have a wide triangle.

[119]:
```
df_raa = cl.load_sample("raa").to_frame()
df_raa
```

[119]:
```
                12       24       36       48       60       72       84   \
1981-01-01  5012.0   8269.0  10907.0  11805.0  13539.0  16181.0  18009.0
1982-01-01   106.0   4285.0   5396.0  10666.0  13782.0  15599.0  15496.0
1983-01-01  3410.0   8992.0  13873.0  16141.0  18735.0  22214.0  22863.0
1984-01-01  5655.0  11555.0  15766.0  21266.0  23425.0  26083.0  27067.0
1985-01-01  1092.0   9565.0  15836.0  22169.0  25955.0  26180.0      NaN
1986-01-01  1513.0   6445.0  11702.0  12935.0  15852.0      NaN      NaN
1987-01-01   557.0   4020.0  10946.0  12314.0      NaN      NaN      NaN
1988-01-01  1351.0   6947.0  13112.0      NaN      NaN      NaN      NaN
1989-01-01  3133.0   5395.0      NaN      NaN      NaN      NaN      NaN
1990-01-01  2063.0      NaN      NaN      NaN      NaN      NaN      NaN

                96       108      120
1981-01-01  18608.0  18662.0  18834.0
1982-01-01  16169.0  16704.0      NaN
1983-01-01  23466.0      NaN      NaN
1984-01-01      NaN      NaN      NaN
1985-01-01      NaN      NaN      NaN
1986-01-01      NaN      NaN      NaN
1987-01-01      NaN      NaN      NaN
1988-01-01      NaN      NaN      NaN
1989-01-01      NaN      NaN      NaN
1990-01-01      NaN      NaN      NaN
```

You can use `pandas` to `.unstack()` the data into the wide long format.

[120]:
```
df_raa = df_raa.unstack().dropna().reset_index()
df_raa.columns = ["age", "origin", "values"]
df_raa.head(10)
```

[120]:
```
   age     origin  values
0   12 1981-01-01  5012.0
1   12 1982-01-01   106.0
2   12 1983-01-01  3410.0
3   12 1984-01-01  5655.0
4   12 1985-01-01  1092.0
```

(continues on next page)

(continued from previous page)

```
5   12 1986-01-01  1513.0
6   12 1987-01-01   557.0
7   12 1988-01-01  1351.0
8   12 1989-01-01  3133.0
9   12 1990-01-01  2063.0
```

Next, we will need a valuation column (think Schedule P style triangle).

[121]:
```
df_raa["valuation"] = (df_raa["origin"].dt.year + df_raa["age"] / 12 - 1).astype(int)
df_raa.head(10)
```

[121]:
```
   age     origin  values  valuation
0   12 1981-01-01  5012.0       1981
1   12 1982-01-01   106.0       1982
2   12 1983-01-01  3410.0       1983
3   12 1984-01-01  5655.0       1984
4   12 1985-01-01  1092.0       1985
5   12 1986-01-01  1513.0       1986
6   12 1987-01-01   557.0       1987
7   12 1988-01-01  1351.0       1988
8   12 1989-01-01  3133.0       1989
9   12 1990-01-01  2063.0       1990
```

Now, we are ready to load it into the `chainladder` package!

[122]:
```
cl.Triangle(
    df_raa, origin="origin", development="valuation", columns="values", cumulative=True
)
```

[122]:
```
         12       24       36       48       60       72       84       96       108  ␣
↪ 120
1981  5012.0   8269.0  10907.0  11805.0  13539.0  16181.0  18009.0  18608.0  18662.0 ␣
↪18834.0
1982   106.0   4285.0   5396.0  10666.0  13782.0  15599.0  15496.0  16169.0  16704.0  ␣
↪ NaN
1983  3410.0   8992.0  13873.0  16141.0  18735.0  22214.0  22863.0  23466.0      NaN  ␣
↪ NaN
1984  5655.0  11555.0  15766.0  21266.0  23425.0  26083.0  27067.0      NaN      NaN  ␣
↪ NaN
1985  1092.0   9565.0  15836.0  22169.0  25955.0  26180.0      NaN      NaN      NaN  ␣
↪ NaN
1986  1513.0   6445.0  11702.0  12935.0  15852.0      NaN      NaN      NaN      NaN  ␣
↪ NaN
1987   557.0   4020.0  10946.0  12314.0      NaN      NaN      NaN      NaN      NaN  ␣
↪ NaN
1988  1351.0   6947.0  13112.0      NaN      NaN      NaN      NaN      NaN      NaN  ␣
↪ NaN
1989  3133.0   5395.0      NaN      NaN      NaN      NaN      NaN      NaN      NaN  ␣
↪ NaN
1990  2063.0      NaN      NaN      NaN      NaN      NaN      NaN      NaN      NaN  ␣
↪ NaN
```

4.5.2 Incremental vs Cumulative Triangles

While actuaries are working with cumulative triangles more often, cumulative triangles are actually inferior to incremental triangles when they come to data storage and processing. This is because incremental triangles only have to contain information on any differences, whereas cumulative triangles contain all the information that the incremental triangle has, but will take up more storage space since a data point needs to exist for all valuation period.

Consider the `prism` dataset, which is a claim level dataset with over 13,000 triangles.

[123]:
```
prism = cl.load_sample("prism")
prism
```

[123]:
```
                                          Triangle Summary
Valuation:                                         2017-12
Grain:                                                OMDM
Shape:                                  (34244, 4, 120, 120)
Index:      [ClaimNo, Line, Type, ClaimLiability, Limit, D...
Columns:     [reportedCount, closedPaidCount, Paid, Incurred]
```

We see that the dataset has a shape of `(34244, 4, 120, 120)` with its size being only approximately `2.8M`.

[124]:
```
prism.values
```

[124]:
```
<COO: shape=(34244, 4, 120, 120), dtype=float64, nnz=121178, fill_value=nan>
```

But converting the dataset into cumulative triangle, the size is significantly larger, by nearly 80 times!

[125]:
```
prism.incr_to_cum().values
```

[125]:
```
<COO: shape=(34244, 4, 120, 120), dtype=float64, nnz=5750047, fill_value=nan>
```

4.5.3 Sparse Triangles

By default, the `chainladder.Triangle()` is a wrapper around a `numpy` array. `numpy` is optimized for high performance, and this allows `chainladder` to achieve decent computational speeds. However, despite being fast, `numpy` can become memory inefficient with triangle data because triangles are inherently sparse (when memory is being allocated yet no data is stored). This is because the lower half of an incomplete triangle is generally blank and that means about 50% of an array size is allocated yet wasted to store nothing. As we include granular index and column values in our Triangle, the sparsity of the triangle increases, further consuming RAM unnecessarily. `chainladder` automatically eliminates this extraneous consumption of memory by resorting to a sparse array representation when the `chainladder.Triangle()` becomes sufficiently large. Unless you are an expert or are running into RAM issues, it will be uncommon for an end user to have to make these adjustments manually.

4.5.4 Claim-Level Data

The sparse representation of triangles allows for substantially more data to be pushed through `chainladder`. This gives us some nice capabilities that we would not otherwise be able to do with aggregated data.

This will allow you to drill into the individual claim makeup of any cell in our `chainladder.Triangle()`.

For example, we can look at all the claims happened in January of 2017, at age 12.

[126]:
```
claims = prism[prism.origin == "2017-01"][prism.development == 12].to_frame()
claims[abs(claims).sum(axis="columns") != 0].reset_index()
```

[126]:
```
    ClaimNo  Line Type  ClaimLiability    Limit  Deductible  reportedCount  \
0     38147  Auto   PD            True  20000.0        1000            1.0
1     38157  Auto   PD            True  20000.0        1000            1.0
2     38163  Auto   PD            True   8000.0        1000            1.0
3     38192  Auto   PD            True  20000.0        1000            1.0
4     38197  Auto   PD            True  20000.0        1000            1.0
5     38215  Auto   PD            True   8000.0        1000            1.0
6     38221  Auto   PD            True  20000.0        1000            1.0
7     38228  Auto   PD            True  20000.0        1000            1.0
8     38263  Auto   PD            True  15000.0        1000            1.0
9     38307  Auto   PD            True  15000.0        1000            1.0
10    38367  Auto   PD            True  20000.0        1000            1.0
11    38373  Auto   PD            True  20000.0        1000            1.0
12    38377  Auto   PD            True  20000.0        1000            1.0
13    38393  Auto   PD            True   8000.0        1000            1.0
14    38396  Auto   PD            True   8000.0        1000            1.0
15    38455  Auto   PD            True  20000.0        1000            1.0
16    38457  Auto   PD            True  15000.0        1000            1.0
17    38460  Auto   PD            True  15000.0        1000            1.0

    closedPaidCount          Paid      Incurred
0               1.0  14540.067710  14540.067710
1               1.0   3873.094721   3873.094721
2               1.0   7000.000000   7000.000000
3               1.0   6451.718210   6451.718210
4               1.0  18222.149160  18222.149160
5               1.0   7000.000000   7000.000000
6               1.0  17844.211120  17844.211120
7               1.0  10262.441970  10262.441970
8               1.0  14000.000000  14000.000000
9               1.0  11901.151660  11901.151660
10              1.0   5207.251558   5207.251558
11              1.0   4900.780565   4900.780565
12              1.0   5741.776419   5741.776419
13              1.0   7000.000000   7000.000000
14              1.0   7000.000000   7000.000000
15              1.0   9927.351224   9927.351224
16              1.0   7874.879070   7874.879070
17              1.0   3125.840672   3125.840672
```

Or that if we want to cop large losses or create an excess triangle on the fly.

[127]:
```
prism["Capped 100k Paid"] = cl.minimum(prism["Paid"], 100000)
prism["Excess 100k Paid"] = prism["Paid"] - prism["Capped 100k Paid"]
```

We can take this even further, we can create claim-level IBNR estimates.

But first, let's fit a simple Chainladder model using the aggregated data.

[128]:
```
agg_data = prism.sum()[["Paid", "reportedCount"]]
cl_mod = cl.Chainladder().fit(agg_data)
```

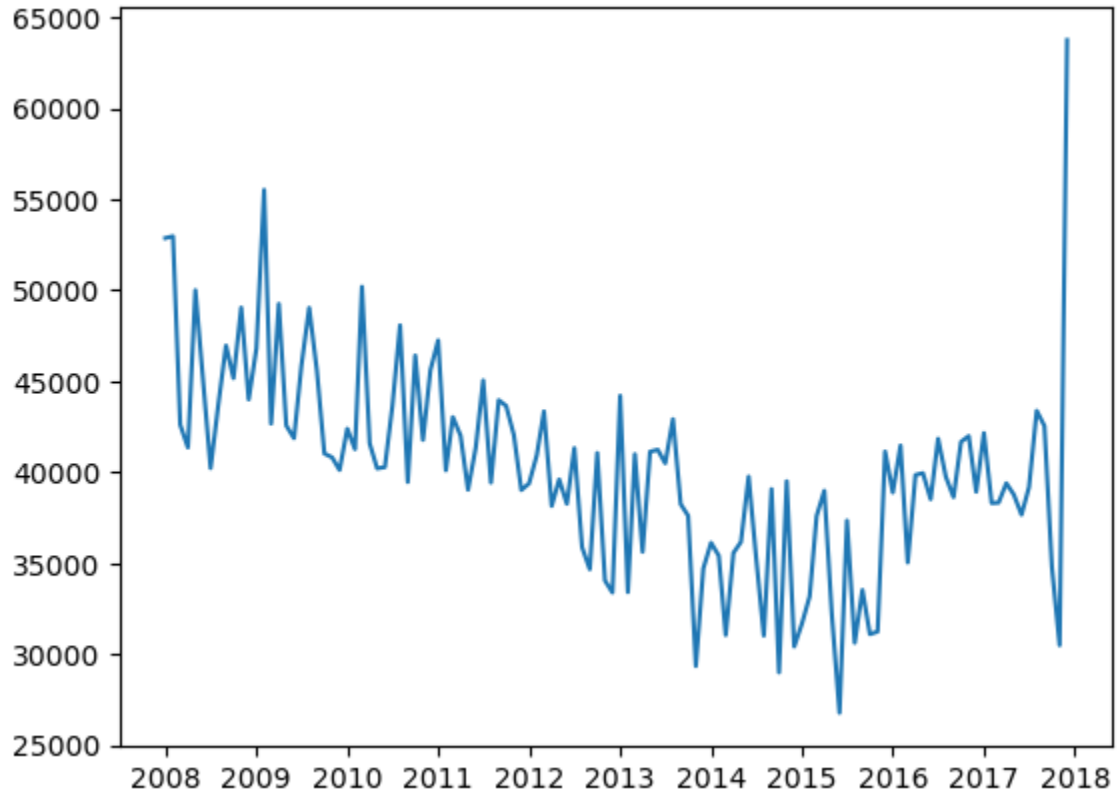
[129]:
```
cl_ults = cl_mod.predict(prism[["Paid", "reportedCount"]]).ultimate_
cl_ults
```

[129]:
```
                                          Triangle Summary
Valuation:                                        2261-12
Grain:                                               OMDM
Shape:                                  (34244, 2, 120, 1)
Index:      [ClaimNo, Line, Type, ClaimLiability, Limit, D...
Columns:                              [Paid, reportedCount]
```

Let's try a Bornhuetter-Ferguson method as well. We will infer an a priori severity from our chainladder model, `cl_mod` above.

[130]:
```
plt.plot(
    (cl_mod.ultimate_["Paid"] / cl_mod.ultimate_["reportedCount"]).to_frame(),
)
```

[130]:
```
[<matplotlib.lines.Line2D at 0x17889cf50>]
```



There are some trends in the severity, but $40,000 doesn't look like a bad a priori (at least for the last two years or so, between 2016 and 2018).

Now, let's fit an aggregate Bornhuetter-Ferguson model. Like the Chainladder example, we fit the model in aggregate (summing all claims) to create a stable model from which we can generate granular predictions. We will use our Chainladder ultimate claim counts as our sample_weight (exposure) for the Bornhuetter-Ferguson method. Essentially, we are saying for each `reportedCount`, we will assume `40000` each of ultimate severity as the a priori.

[131]:
```
paid_bf = cl.BornhuetterFerguson(apriori=40000).fit(
    X=prism["Paid"].sum().incr_to_cum(), sample_weight=cl_ults["reportedCount"].sum()
)
```

We can now create claim-level Bornhuetter-Ferguson predictions using our claim-level Triangle.

[132]:
```
paid_bf.predict(
    prism["Paid"].incr_to_cum(), sample_weight=cl_ults["reportedCount"]
).ultimate_
```

[132]:
```
                                          Triangle Summary
Valuation:                                          2261-12
Grain:                                                OMDM
Shape:                                   (34244, 1, 120, 1)
Index:      [ClaimNo, Line, Type, ClaimLiability, Limit, D...
Columns:                                             [Paid]
```

4.5.5 Consolidating Exposure and Loss Data

Very often, actuaries will use both exposure data and loss data together, instead of just studying and analyzing loss data by itself. Because of this, the practitioner will need to figure out a way to consolidate the data together into a single dataset. This is often more challenging in practice as one would have to first worry about pulling data from difference data systems, and consolidating them together after.

When it comes to consolidating the data, you most likely will choose one of these two options.

Exposure and Loss Data as Columns

The `clrd` dataset is prepared this way, where exposure and loss data sit side-by-side.

[133]:
```
cl.load_sample("clrd").to_frame().reset_index()
```

[133]:

	GRNAME	LOB	origin	development	IncurLoss	CumPaidLoss	BulkLoss	EarnedPremDIR	EarnedPremCeded	EarnedPremNet
0	Adriatic Ins Co	othliab	1995-01-01	12	8.0	NaN	8.0	139.0	131.0	8.0
1	Adriatic Ins Co	othliab	1995-01-01	24	11.0	NaN	4.0	139.0	131.0	8.0
2	Adriatic Ins Co	othliab	1995-01-01	36	7.0	3.0	4.0	139.0	131.0	8.0
3	Adriatic Ins Co	othliab	1996-01-01	12	40.0	NaN	40.0	410.0	359.0	51.0
4	Adriatic Ins Co	othliab	1996-01-01	24	40.0	NaN	40.0	410.0	359.0	51.0
...	...	...	...	...	...	...	...	...	...	...
33084	Yel Co Ins	comauto	1995-01-01	36	282.0	37.0	102.0	403.0	NaN	403.0
33085	Yel Co Ins	comauto	1996-01-01	12	707.0	40.0	495.0	996.0	NaN	996.0
33086	Yel Co Ins	comauto	1996-01-01	24	699.0	60.0	502.0	996.0	NaN	996.0
33087	Yel Co Ins	comauto	1997-01-01	12	698.0	25.0	506.0	996.0	NaN	996.0
33088	Zurich Ins (Guam) Inc	wkcomp	1997-01-01	12	NaN	NaN	NaN	55.0	NaN	55.0

(continues on next page)

(continued from previous page)

```
[33089 rows x 10 columns]
```

Some benefits of organizing the data this way include:

- This method makes the organization of data a bit easier to follow, since the data is basically organized by time period.
- You can use `.head()` or `.tail()` to inspect the data that includes both exposure data and loss data.

But there are drawbacks:

- The practitioner may want to consider if they have "exposure" periods where no data is available, and how that should be handled.
- Merging the two data sources, which might come from different databases, might be challenging.

Stacking Exposure and Loss Data on Top of Each Other

Another option is that you can stack the loss data on top of exposure data, or vice versa.

Some of the benefits include: - Not having to allocate or map exposure and loss amounts one to one. - The data can be prepared easier, since exposure and loss data generally sits in two separate data warehouses.

However, some drawbacks include: - The table can be twice as long. - Your table will contain a lot of empty cells, which may or may not be a problem. - You already have to think about how to map exposures and losses later, as required by some models.

4.5.6 How Much Data Should I Load?

Of course, there's no need to have "everything" loaded into the RAM, even though `chainladder` is designed to handle multiple triangles at once.

The amount of data you should load into your system for analysis depends on several factors, including the objectives of your analysis, and the computational resources available. Generally, it is advisable to start with the simplest dataset that is just enough to capture the problem that you are trying to analyze. For example, even though loading claim level data *might* be useful, there's no need to keep the data at that level of detail in your first go.

Keeping things small allows you to test your analysis methods, identify potential issues, and make adjustments without committing excessive resources upfront. As you refine your methods and increase your system's capacity, you can progressively scale up the amount of data you analyze. Ultimately, the goal is to use a dataset that is sufficient to achieve reliable and statistically significant results, while efficiently using your resources.

CHAPTER

FIVE

REFERENCES

1. Wickham, Hadley, 2014. "Tidy Data," Journal of Statistical Software, Foundation for Open Access Statistics, vol. 59(i10)
2. Shapland, Mark. 2016. "CAS Monograph No. 4: Using The ODP Bootstrap Model: A Practitioner's Guide" Casualty Actuarial Society
3. Powell, et. al. 2009. "Errors in Operational Spreadsheets" Journal of Organizational and End User Computing
4. Struzzieri Hussain,1998. "Using Best Practices to Determine a Best Reserve Estimate"
5. Thomas Mack, 2014. "Distribution-free Calculation of the Standard Error of Chain Ladder Reserve Estimates"
6. Fannin, Brian, 2022. "CAS Actuarial Technology Survey"
7. Astin Working Party 2016, "Non Life Reserving Practices"
8. Buitinck et al., 2013. "API design for machine learning software: experiences from the scikit-learn project"
9. Mckinney, Wes 2011. "pandas: a Foundational Python Library for Data Analysis and Statistics"
10. Caseres, Matthew. https://www.actuarialopensource.org/
11. Mack, Thomas 1994. "Which Stochastic Model is Underlying the Chain Ladder Method?" RAA triangle
12. Meyers, Glenn and Shi, Peng. 2011 "Loss Reserving Data Pulled from NAIC Schedule P" CLRD triangle