Article from

The Modeling Platform

November 2019

Issue 10

An R Package for Experience Studies

By Matthew Caseres

Significant effort is required to transform actuarial data from a raw format into a format that can be used for the calculation of decrement rates or average premiums. The expstudies R package was developed to aid in transforming raw data into assumptions.

This package relies on dplyr, a popular R package with highly optimized algorithms for data manipulation. Functions that had no obvious dplyr implementation were written in C++ for high performance. Using this package on a desktop computer, I have been able to handle millions of rows of data without issue.

The package is open source and is available on GitHub and can be installed using Comprehensive R Archive Network (CRAN) with `install.packages('expstudies')`. The package is in version 0.0.5 so there are still improvements to make. The official package documentation is at *https:/actuarialanalyst.github.io/expstudies/.*

HOW TO READ THIS ARTICLE

There is some R code in this article, in grey boxes. If you don't read the grey boxes at all, everything will make sense from just reading the non-code text so don't worry if you don't know R. The table below the code displays the output from running the code. In this case, "records" is a variable in our environment, and we are displaying its value.

```
records
```

Policy Number	Issue Date	Termination Date	Issue Age	Gender
B10251C8	2010-04-10	2019-04-04	35	M
D68554D5	2005-01-01	2019-04-04	30	F

These records are used for demonstration purposes in this article. We assume a data format with a unique policy number, issue date, termination date, issue age and gender.

CREATING EXPOSURES

The raw data has a single row per policy. For calculations, we would like multiple rows per policy where each row represents an interval of time where the policy was in force. The addExposures function does this. By default, exposure rows are created for each policy year.

```
addExposures(records)
```

Policy Number	Policy Date	Start Interval	End Interval
B10251C8	1	2010-04-10	2011-04-09
B10251C8	2	2010-04-10	2011-04-09
B10251C8	3	2010-04-10	2011-04-09

Note: Table has been truncated

ADDEXPOSURES() ARGUMENTS

To allow for greater user control, there are arguments that control the creation of the exposure data frame.

Type

We can partition experience by policy month using the type argument with type = "PM".

```
addExposures(records, type = "PM")
```

Policy Number	Policy Year	Policy Month	Start Interval	End Interval
B10251C8	1	1	2010-04-10	2010-05-09
B10251C8	1	2	2010-05-10	2010-06-09
B10251C8	1	3	2010-06-10	2010-07-09

Note: Table has been truncated

Policy years cross multiple calendar years and we might need to do an analysis filtering by both exact calendar year and policy year. This can be accomplished using type = "PYCY".

```
addExposures(records, type = "PYCY")
```

Policy Number	Policy Year	Start Interval	End Interval
B10251C8	1	2010-04-10	2010-12-31
B10251C8	1	2011-01-01	2011-04-09
B10251C8	2	2011-04-10	2011-12-31

Note: Table has been truncated

There are also options for policy year with calendar month, policy month with calendar year, and policy month with calendar month. The following table shows output for policy month with calendar year.

```
addExposures(records, type = "PMCY")
```

Policy Number	Policy Year	Policy Month	Start Interval	End Interval
Note: Table has been truncated				
B10251C8	1	8	2010-11-10	2010-12-09
B10251C8	1	9	2010-12-10	2010-12-31
B10251C8	1	9	2011-01-01	2011-01-09
B10251C8	1	10	2011-01-10	2011-02-09
Note: Table has been truncated				

Lower_Year

The lower_year argument allows for left truncation by calendar year. Exposure rows will only be created if the interval contains no dates from years below the lower_year argument. This can reduce computation time and memory use.

```
addExposures(records, type="PY", lower_year=2017)
```

Policy Number	Policy Year	Start Interval	End Interval
B10251C8	8	2017-04-10	2018-04-09
B10251C8	9	2018-04-10	2019-04-04
D68554D5	13	2017-01-01	2017-12-31
D68554D5	14	2018-01-01	2018-12-31
D68554D5	15	2019-01-01	2019-04-04

Determine Output Size Before Calling addExposures() Using expSize()

The expSize function calculates an upper bound for the number of rows created by a call to addExposures but doesn't create the exposures. The expSize function runs faster and uses less memory than addExposures for large outputs so it can be useful.

```
expSize(records, type = "PY")
```

Upper Bound on Output Size
25

Joining Additional Information to Exposures

The call to addExposures removed the issue age and gender fields. We add these fields back by joining our original records to the created exposures using the policy number as the join criterion. In the next section, we discuss how to join by both a policy number and date.

Policy Number	Policy Year	Start Interval	End Interval	Issue Age	Gender
B10251C8	1	2010-04-10	2011-04-09	35	M
B10251C8	2	2011-04-10	2012-04-09	35	M
B10251C8	3	2012-04-10	2013-04-09	35	M
Note: Table has been truncated					

PREMIUM PATTERN

Suppose we would like to analyze premium pattern by policy month for some transaction data.

```
trans
```

Policy Number	Transaction Date	Amount
B10251C8	2012-12-04	199
B10251C8	2013-12-28	197
B10251C8	2015-12-30	177
Note: Table has been truncated		

We create monthly exposures called exposures_PM using the addExposures function. Later we join aggregated transaction data to these exposures.

```
exposures_PM <- addExposures(records, type = "PM")
exposures_PM
```

Policy Number	Policy Year	Policy Month	Start Interval	End Interval
B10251C8	1	1	2010-04-10	2010-05-09
B10251C8	1	2	2010-05-10	2010-06-09
B10251C8	1	3	2010-06-10	2010-07-09
Note: Table has been truncated				

Allocating Transactions

The addStart function adds a start interval column to the transaction data that corresponds to the correct start interval from the exposure data frame. The start interval and policy number columns specify what row of the exposures_PM data frame a transaction should be allocated to.

```
trans_with_interval <- addStart(trans, exposures_PM)
trans_with_interval
```

Start Interval	Policy Number	Transaction Date	Amount
2010-05-10	B10251C8	2010-05-28	190
2010-06-10	B10251C8	2010-07-04	189
2010-11-10	B10251C8	2010-11-21	179
Note: Table has been truncated			

We can group and aggregate transactions by policy number and issue date to get transaction totals for intervals in exposures_PM.

```
trans_to_join <- trans_with_interval %>%
  group_by(`Start Interval`, `Policy Number`) %>%
  summarise(`Total Amount` = sum(Amount))
trans_to_join
```

Start Interval	Policy Number	Total Amount
2005-06-01	D68554D5	97
2005-10-01	D68554D5	169
2005-12-01	D68554D5	96

Note: Table has been truncated

Then we can left join the aggregated transactions to the exposures data frame with join criteria of policy number and start interval.

Policy Number	Policy Year	Policy Month	Start Interval	End Interval	Total Amount
B10251C8	1	1	2010-04-10	2010-05-09	NA
B10251C8	1	2	2010-05-10	2010-06-09	190
B10251C8	1	3	2010-06-10	2010-07-09	189
B10251C8	1	4	2010-07-10	2010-08-09	NA

Note: Table has been truncated

We then fill in NA values with zero.

Policy Number	Policy Year	Policy Month	Start Interval	End Interval	Total Amount
B10251C8	1	1	2010-04-10	2010-05-09	0
B10251C8	1	2	2010-05-10	2010-06-09	190
B10251C8	1	3	2010-06-10	2010-07-09	189
B10251C8	1	4	2010-07-10	2010-08-09	0

Note: Table has been truncated

Now we are done with the data wrangling; from here it is not hard to calculate things like average premium per policy year or policy month. We can even export this data as a .csv to make dashboards in a business intelligence tool.

Other Uses for addStart()

Suppose we were interested in the last non-zero monthly premium paid by a policy. We left join the aggregated premiums to the exposure frame as we did before. This time we fill in NA values with the previous paid premium instead of 0. The first interval is NA because there are no prior premiums.

Policy Number	Policy Year	Policy Month	Start Interval	End Interval	Total Amount
B10251C8	1	1	2010-04-10	2010-05-09	NA
B10251C8	1	2	2010-05-10	2010-06-09	190
B10251C8	1	3	2010-06-10	2010-07-09	189
B10251C8	1	4	2010-07-10	2010-08-09	189
B10251C8	1	5	2010-08-10	2010-09-09	189
B10251C8	1	6	2010-09-10	2010-10-09	189

Note: Table has been truncated

Data manipulations like this can engineer features for anything varying with time like account values, guarantees or planned premiums.

DECREMENT RATES

Calculating mortality and lapse rates is not difficult once we have created the exposure data frame. In the following example, we calculate the exposure as

$$\frac{(\#\text{Days in Interval})}{365.25}.$$

It is not difficult to add a death indicator and use a full exposure in the year of death for performing a mortality study.

Policy Number	Policy Year	Start Interval	End Interval	Exposure	Death Count
Note: Table has been truncated					
D68554D5	12	2016-01-01	2016-12-31	1.002	0
D68554D5	13	2017-01-01	2017-12-31	0.9993	0
D68554D5	14	2018-01-01	2018-12-31	0.9993	0
D68554D5	15	2019-01-01	2019-04-04	1	1

We then aggregate by duration to calculate mortality rates.

```
exposures_mort %>%
  group_by(`Policy Year`) %>%
  summarise(q = sum(`Death Count`)/
  sum(`Exposure`))
```

Policy Year	q
Note: Table has been truncated	
8	0
9	0.5002
10	0
11	0
12	0
13	0
14	0
15	1

Expected Mortality

Some mortality tables have been loaded to the package in an easy-to-join format so that users can conduct an actual to expected analysis.

```
expstudies::mortality_tables
            mortality_tables
```

```
AM92
GAM1983
UP1984
VBT2015_SmokerDistinct_ALB
CSO2017_Loaded_PreferredStructure_ALB
```

We view the AM92 Ultimate table.

```
mortality_tables$AM92$AM92_Ultimate
```

Attained Age	q Ultimate
19	0.000587
20	0.000582
21	0.000577
22	0.000572
23	0.000569

Note: Table has been truncated

We can join the mortality table to a data frame using the attained age as the join criterion for actual to expected analysis of calculated rates.

OPPORTUNITIES

It would not be difficult to implement the methods in this package in Python using pandas. In R, Python or Apache Spark, there is potential for running really large experience studies by parallelizing calculations. It would be nice if there was a framework for experience study calculations that has been reviewed by many people so that others are comfortable relying on the framework.

There is a question I am curious about the answer to. For a given data set and product specification, about would different organizations produce materially different models? I don't think there is much room for difference in lapse/mortality rate implementations, but there are many approaches that can be taken for something like premium pattern. Should we classify policies into separate premium schedules based on some combination of characteristics? Should we model future premiums as a percentage of planned premium? I think it would be interesting to have some sample data sets and people can produce and share simple universal life models in Excel or Python to compare different modeling practices. I don't think there is much research in the field that is fully reproducible due to data privacy concerns and proprietary modeling systems, so there is lots of work that can be done. In the future maybe actuarial organizations will sponsor things like open source economic scenario generators or open source models. ■

Matthew Caseres is an actuarial analyst at Ameritas. He can be reached at *matthew.caseres@ameritas.com* and on GitHub at *github.com/ActuarialAnalyst*.