IBNR Models

Contents

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.

Basics and Commonalities

Ultimates

All reserving methods determine some ultimate cost of insurance claims. These ultimates are captured in the `ultimate_` property of the estimator.

```
import chainladder as cl
import pandas as pd

cl.Chainladder().fit(cl.load_sample('raa')).ultimate_
```

latest

	2261
1981	18,834
1982	16,858
1983	24,083
1984	28,703
1985	28,927
1986	19,501
1987	17,749
1988	24,019
1989	16,045
1990	18,402

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. This is set globally and can be viewed by referencing the `ULT_VAL` constant.

```
cl.options.get_option('ULT_VAL')
```

```
'2261-12-31 23:59:59.999999999'
```

```
cl.options.set_option('ULT_VAL', '2050-12-31 23:59:59.999999999')
cl.options.get_option('ULT_VAL')
```

```
'2050-12-31 23:59:59.999999999'
```

The `ultimate_` along with most of the other properties of IBNR models are triangles and can be manipulated. However, it is important to note that the model itself is not a Triangle, it is an scikit-learn style Estimator. This distinction is important when wanting to manipulate model attributes.

```
triangle = cl.load_sample('quarterly')
model = cl.Chainladder().fit(triangle)
```

latest

```
# This works since we're slicing the ultimate Triangle
ult = model.ultimate_['paid']
```

```
/home/docs/checkouts/readthedocs.org/user_builds/chainladder-python/conda/lates
  arr = dict(zip(datetime_arg, pd.to_datetime(**item)))
/home/docs/checkouts/readthedocs.org/user_builds/chainladder-python/conda/lates
  arr = dict(zip(datetime_arg, pd.to_datetime(**item)))
```

This throws an error since the model itself is not sliceable:

```
ult = model['paid'].ultimate_
```

IBNR

Any difference between an `ultimate_` and the `latest_diagonal` of a Triangle is contained in the `ibnr_` property of an estimator. While technically, as in the example of a paid triangle, there can be case reserves included in the `ibnr_` estimate, the distinction is not made by the `chainladder` package and must be managed by you.

```
triangle = cl.load_sample('quarterly')
model = cl.Chainladder().fit(triangle)

# Determine outstanding case reserves
case_reserves = (triangle['incurred']-triangle['paid']).latest_diagonal

# Net case reserves off of paid IBNR
true_ibnr = model.ibnr_['paid'] - case_reserves
true_ibnr.sum()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/chainladder-python/conda/lates
  arr = dict(zip(datetime_arg, pd.to_datetime(**item)))
/home/docs/checkouts/readthedocs.org/user_builds/chainladder-python/conda/lates
  arr = dict(zip(datetime_arg, pd.to_datetime(**item)))
```

```
2431.2695585474003
```

latest

Complete Triangles

The `full_triangle_` and `full_expectation_` attributes give a view of the completed `Triangle`. While the `full_expectation_` is entirely based on `ultimate_` values and development patterns, the `full_triangle_` is a blend of the existing triangle. These are useful for conducting an analysis of actual results vs model expectations.

```
model = cl.Chainladder().fit(cl.load_sample('ukmotor'))
residuals = model.full_expectation_ - model.full_triangle_
residuals[residuals.valuation<=model.X_.valuation_date]
```

	12	24	36	48	60	72	84
2007	344.49	557.93	348.77	10.85	-11.41		
2008	-21.88	-185.51	-340.72	-102.58	11.41		
2009	-92.22	-233.62	94.51	91.74			
2010	-303.44	-209.00	-102.57				
2011	67.16	70.21					
2012	5.89						
2013							

Another typical analysis is to forecast the IBNR run-off for future periods.

```
expected_3y_run_off = model.full_triangle_.dev_to_val().cum_to_incr().loc[...,
expected_3y_run_off
```

latest

	2014	2015	2016
2007			
2008	351		
2009	662	376	
2010	1,073	620	352
2011	1,503	1,134	655
2012	2,725	1,820	1,374
2013	5,587	3,352	2,239

Chainladder

The distinguishing characteristic of the :class:`Chainladder` method is that ultimate claims for each accident year are produced from recorded values assuming that future claims' development is similar to prior years' development. In this method, the actuary uses the development triangles to track the development history of a specific group of claims. The underlying assumption in the development technique is that claims recorded to date will continue to develop in a similar manner in the future – that the past is indicative of the future. That is, the development technique assumes that the relative change in a given year's claims from one evaluation point to the next is similar to the relative change in prior years' claims at similar evaluation points.

An implicit assumption in the development technique is that, for an immature accident year, the claims observed thus far tell you something about the claims yet to be observed. This is in contrast to the assumptions underlying the expected claims technique.

Other important assumptions of the development method include: consistent claim processing, a stable mix of types of claims, stable policy limits, and stable reinsurance (or excess insurance) retention limits throughout the experience period.

Though the algorithm underling the basic chainladder is trivial, the properties of the `Chainladder` estimator allow for a concise access to relevant information.

As an example, we can use the estimator to determine actual vs expected latest subsequent valuation period.

[Friedland, 2010]

MackChainladder

The :class:`MackChainladder` model can be regarded as a special form of a weighted linear regression through the origin for each development period. By using a regression framework, statistics about the variability of the data and the parameter estimates allows for the estimation of prediction errors. The Mack Chainladder method is the most basic of stochastic methods.

Compatibility

Because of the regression framework underlying the `MackChainladder`, it is not compatible with all development and tail estimators of the library. In fact, it really should only be used with the `Development` estimator and `TailCurve` tail estimator.

> **Warning**
>
> While the MackChainladder might not error with other options for development and tail, the stochastic properties should be ignored, in which case the basic `Chainladder` should be used.

Examples

[Mack, 1993] [Mack, 1999]

BornhuetterFerguson

The :class:`BornhuetterFerguson` technique is essentially a blend of the development and expected claims techniques. In the development technique, we multiply actual claims by a cumulative claim development factor. This technique can lead to erratic, unreliable projections when the cumulative development factor is large because a relatively small swing in reported claims or the reporting of an unusually large claim could result in a very lar d ultimate claims. In the expected claims technique, the unpaid claim estima difference between a predetermined estimate of expected claims and the actual payments. This has the advantage of stability, but it completely ignores actual results as reported. The Bornhuetter-Ferguson technique combines the two techniques by splitting ultimate claims into two components: actual reported (or paid) claims and expected unreported (or unpaid) claims. As experience matures, more weight is given to the actual claims and the expected claims become gradually less important.

Exposure base

The :class:`BornhuetterFerguson` technique is the first we explore of the Expected Loss techniques. In this family of techniques, we need some measure of exposure. This is handled by passing a `Triangle` representing the exposure to the `sample_weight` argument of the `fit` method of the Estimator.

All scikit-learn style estimators optionally support a `sample_weight` argument and this is used by the `chainladder` package to capture the exposure base of these Expected Loss techniques.

```
raa = cl.load_sample('raa')
sample_weight = raa.latest_diagonal*0+40_000
cl.BornhuetterFerguson(apriori=0.7).fit(
    X=raa,
    sample_weight=sample_weight
).ibnr_.sum()
```

```
75203.23550854485
```

Apriori

We've fit a :class:`BornhuetterFerguson` model with the assumption that our prior belief, or `apriori` is a 70% Loss Ratio. The method supports any constant for the `apriori` hyperparameter. The `apriori` then gets multiplied into our sample weight to determine our prior belief on expected losses prior to considering that actual emerged to date.

Because of the multiplicative nature of `apriori` and `sample_weight` we don't have to limit ourselves to a single constant for the `apriori`. Instead, we can exploit th latest make our `sample_weight` represent our prior belief on ultimates while setting the `apriori` to 1.0.

For example, we can use the :class:`Chainladder` ultimates as our prior belief in the :class:`BornhuetterFerguson` method.

```
cl_ult = cl.Chainladder().fit(raa).ultimate_ # Chainladder Ultimate
apriori = cl_ult*0+(cl_ult.sum()/10) # Mean Chainladder Ultimate
cl.BornhuetterFerguson(apriori=1).fit(raa, sample_weight=apriori).ultimate_
```

	2050
1981	18,834
1982	16,899
1983	24,012
1984	28,282
1985	28,204
1986	19,840
1987	18,840
1988	22,790
1989	19,541
1990	20,986

[Friedland, 2010]

Benktander

The :class:`Benktander` method is a credibility-weighted average of the :class:`BornhuetterFerguson` technique and the development technique. The advantage cited by the authors is that this method will prove more responsive than the Bornhuetter-Ferguson technique and more stable than the development technique.

Iterations

The `Benktander` method is also known as the iterated :class:`Bornhuette` This is because it is a generalization of the :class:`BornhuetterFerguson` technique.

latest

The generalized formula based on `n_iters`, n is:

$Ultimate = Apriori\times (1-\frac{1}{CDF})^{n} + Latest\times \sum_{k=0}^{n-1}(1-\frac{1}{CDF})^{k}$

- `n=0` yields the expected loss method
- `n=1` yields the traditional :class:`BornhuetterFerguson` method
- `n>>1` converges to the traditional :class:`Chainladder` method.

Examples

Expected Loss Method

Setting `n_iters` to 0 will emulate that Expected Loss method. That is to say, the actual emerged loss experience of the Triangle will be completely ignored in determining the ultimate. While it is a trivial calculation, it allows for run-off patterns to be developed, which is useful for new programs new lines of businesses.

```
triangle = cl.load_sample('ukmotor')
exposure = triangle.latest_diagonal*0 + 25_000
cl.Benktander(apriori=0.75, n_iters=0).fit(
    X=triangle,
    sample_weight=exposure
).full_triangle_.round(0)
```

	12	24	36	48	60	72	84	96	9999
2007	3,511	6,726	8,992	10,704	11,763	12,350	12,690	12,690	12,690
2008	4,001	7,703	9,981	11,161	12,117	12,746	18,750	18,750	18,750
2009	4,355	8,287	10,233	11,755	12,993	16,664	18,750	18,750	18,750
2010	4,295	7,750	9,773	11,093	15,112	17,432	18,750	18,750	18,750
2011	4,150	7,897	10,217	13,718	16,359	17,884	18,750	18,750	18,750
2012	5,102	9,650	13,112	15,425	17,170	18,178	18,750	18,750	18,750
2013	6,283	11,121	14,024	15,963	17,426	18,270	18,750	[illegible]	[illegible]

latest

Mack noted the `Benktander` method is found to have almost always a smaller mean squared error than the other two methods and to be almost as precise as an exact Bayesian procedure.

[Friedland, 2010]

CapeCod

The :class:`CapeCod` method, also known as the Stanard-Buhlmann method, is similar to the Bornhuetter-Ferguson technique. The primary difference between the two methods is the derivation of the expected claim ratio. In the Cape Cod technique, the expected claim ratio or apriori is obtained from the triangle itself instead of an independent and often judgmental selection as in the Bornhuetter-Ferguson technique.

```
clrd = cl.load_sample('clrd')[['CumPaidLoss', 'EarnedPremDIR']].groupby('LOB')
loss = clrd['CumPaidLoss']
sample_weight=clrd['EarnedPremDIR'].latest_diagonal
m1 = cl.CapeCod().fit(loss, sample_weight=sample_weight)
m1.ibnr_.sum()
```

```
3030598.384680113
```

Apriori

The default hyperparameters for the :class:`CapeCod` method can be emulated by the :class:`BornhuetterFerguson` method. We can manually derive the `apriori` implicit in the CapeCod estimate.

```
cl_ult = cl.Chainladder().fit(loss).ultimate_
apriori = loss.latest_diagonal.sum() / (sample_weight/(cl_ult/loss.latest_diag
m2 = cl.BornhuetterFerguson(apriori).fit(
    X=clrd['CumPaidLoss'],
    sample_weight=clrd['EarnedPremDIR'].latest_diagonal)
m2.ibnr_.sum()
```

latest

```
3030598.384680113
```

A parameter `apriori_sigma` can also be specified to give sampling variance to the estimated apriori. This along with `random_state` can be used in conjuction with the `BootstrapODPSample` estimator to build a stochastic `CapeCod` estimate.

Trend and On-level

When using data implicit in the Triangle to derive the apriori, it is desirable to bring the different origin periods to a common basis. The `CapeCod` estimator provides a `trend` hyperparameter to allow for trending everything to the latest origin period. However, the apriori used in the actual estimation of the IBNR is the `detrended_apriori_` detrended back to each of the specific origin periods.

```
m1 = cl.CapeCod(trend=0.05).fit(loss, sample_weight=sample_weight)
pd.concat((
    m1.detrended_apriori_.to_frame().iloc[:, 0].rename('Detrended Apriori'),
    m1.apriori_.to_frame().iloc[:, 0].rename('Apriori')), axis=1
)
```

	Detrended Apriori	Apriori
1988-01-01	0.483539	0.750128
1989-01-01	0.507716	0.750128
1990-01-01	0.533102	0.750128
1991-01-01	0.559757	0.750128
1992-01-01	0.587745	0.750128
1993-01-01	0.617132	0.750128
1994-01-01	0.647989	0.750128
1995-01-01	0.680388	0.750128
1996-01-01	0.714407	0.750128
1997-01-01	0.750128	0.750128

Simple one-part trends are supported directly in the hyperparameter selection. If a more complex trend assumption is required or on-leveling, then passing Triangles transformed by the :class:`Trend` and :class:`ParallelogramOLF` estimators will capture these [illegible] this example from the example gallery.

Examples

Decay

The default behavior of the `CapeCod` is to include all origin periods in the estimation of the `apriori_`. A more localized approach, giving lesser weight to origin periods that are farther from a target origin period, can be achieved by flexing the `decay` hyperparameter.

```
cl.CapeCod(decay=0.8).fit(loss, sample_weight=sample_weight).apriori_.T
```

	1988	1989	1990	1991	1992	1993	1994	19
2050	0.617945	0.613275	0.604879	0.591887	0.57637	0.559855	0.548615	0.5422

With a `decay` less than 1.0, we see `apriori_` estimates that vary by origin.

[Friedland, 2010]

Previous
Tail Estimators

Next
Data Adjustments