Zero Inflated Poisson Regression

Eric Chuu
Texas A&M University
ericchuu@tamu.edu

Rajan Kapoor
Texas A&M University
r.kapoor@tamu.edu

May 1, 2018

Abstract

Count data are often saturated with excess zero counts and overdispersion, resulting in inappropriate use of standard regression models and thus to incorrect inference. Zero inflated poisson regression offers a solution to this problem by using two models, one for the excess zero counts and one for the regular counts. Support for this model is available in `R` in the `pscl` package, but support in Python for zero inflated Poisson regression is limited. We sought to bring the convenience of the `R` package into the Python environment so that both communities can use this model technique for better results.

I. Introduction

In count data analysis, one of the most prominent models is Poisson regression, which assumes that $y_i|\mathbf{x}_i \sim \text{Poisson}(\mu_i)$, so that the probability mass function is given as:

$$P\left(Y_i = y_i \mid \mathbf{x}_i\right) = \frac{e^{-\mu_i}\mu_i^{y_i}}{y_i!} \cdot \mathbb{1}(y_i \in \mathbb{N} \cup \{0\}) \tag{1}$$

The mean parameter is given by $\mu_i = \exp(\mathbf{x}_i'\beta)$, which is guaranteed to be non-negative. From our knowledge of the Poisson distribution, we also have the following results,

$$E\left(y_i \mid \mathbf{x}_i\right) = \mu_i \tag{2}$$

$$Var\left(y_i \mid \mathbf{x}_i\right) = \mu_i \tag{3}$$

We observe that in the Poisson regression model, there is an assumption of *equidispersion*: the conditional mean and variance are equal.

In general, this oversimplification of the mean and variance structure does not hold in most practical datasets. In fact, due to the nature of many problems and questions, overdispersion and excess zeros are often the norm. We consider a variety of examples across different fields where this problem is prevalent. Figure 1 shows data from an educational study performed at Iran University of Medical Sciences ([Roudbari and Salehi, 2015]) with a goal of investigating contributing factors to poor academic performance. Researchers found that more than 70% of those selected to participate in the study had never failed a course. It is evident from the figure that the assumption of equidispersion does not hold, so other methods must be employed for more accurate analysis of this data.

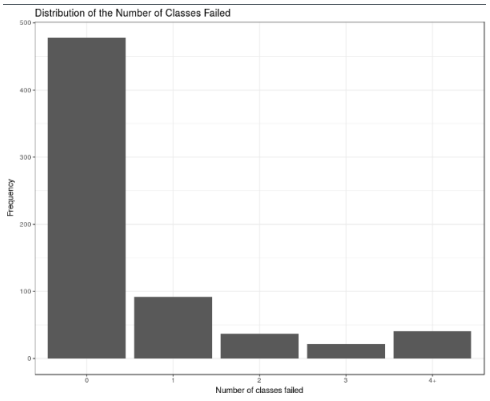

Figure 1: *The distribution of the number of classes failed in a 2008-2009 study at Iran University of Medical Sciences. Of the 670 students selected to partake in the study, 478 (71.3%) failed no courses, 37 (5.5%) failed one course, 22 (3.3%) failed two courses, and 41 (6.2%) failed four or more courses.*

In the context of social sciences as well, there is often the case that datasets are saturated with zero counts. For the dataset that details the frequency of congressional response to Supreme Court decisions in the late nineties, the zero responses dominate the nonzero counts (See table 1).

Table 1: *Number of actions taken by Congress in response to Supreme Court Decisions from 1979 to 1988. More than 95% of the data consists of zero counts, which prevents the traditional use of Poisson regression from being an effective way to model this data.*

# of Actions	Frequency	Percentage
0	**3882**	**95.80**
1	63	1.55
2	38	0.94
3	32	0.79
4	8	0.20
5	12	0.30
6	12	0.30
7	3	0.07
10	1	0.02
11	1	0.02
Total	4052	100

Because of how common these excess-zero scenarios appear in datasets, it makes sense to consider alternative methods that are able to address this data structure.

II. Background of Zero Inflated Poisson Regression

In the Zero Inflated Poisson regression model, instead of assuming a single underlying distribution, we assume two independent data generating processes: one for the excess zeros and one for the regular count data (which may include zeros). More specifically, the response variable y_i is determined by a Bernoulli trial, which can be specified as follows:

$$y_i = \begin{cases} 0 & \text{with probability } \varphi_i \\ \frac{e^{-\mu_i}\mu_i^{y_i}}{y_i!} & \text{with probability 1 - } \varphi_i \end{cases} \quad (4)$$

This yields the corresponding probability mass function:

$$P(Y_i = y_i \mid \mathbf{x}_i, \mathbf{z}_i) = \begin{cases} \varphi_i + (1-\varphi_i)\cdot e^{-\mu_i}, & \text{if } y_i = 0 \\ (1-\varphi_i)\cdot \frac{e^{-\mu_i}\mu_i^{y_i}}{y_i!}, & \text{if } y_i > 0 \end{cases}$$

- $\varphi_i = F\left(z_i'\gamma\right)$, where F is the link function
- $\mathbf{x}_i$ is the vector of count model covariates
- $\mathbf{z}_i$ is the vector of zero inflation covariates
- γ is the vector of zero inflation coefficients
- β is the vector of count model coefficients
- $\mu_i = \exp\left(x_i'\beta\right)$

Under this formulation, the conditional mean and variance are

$$E\left(Y_i \mid \mathbf{x}_i, \mathbf{z}_i\right) = \mu_i(1-\varphi_i)$$

$$Var\left(Y_i \mid \mathbf{x}_i, \mathbf{z}_i\right) = E\left(Y_i \mid \mathbf{x}_i, \mathbf{z}_i\right)\left(1+\mu_i\varphi_i\right)$$

Since $(1+\mu_i\varphi_i) \geq 1$, the expression for the conditional variance implies that $Var\left(Y_i \mid \mathbf{x}_i, \mathbf{z}_i\right) > E\left(Y_i \mid \mathbf{x}_i, \mathbf{z}_i\right)$, so that we no longer assume equidispersion, one of the main issues with the Poisson regression model. When the data is heavily overdispersed, alternative methods may be more suitable, such as zero inflated negative binomial regression and zero inflated geometric regression.

III. Project Proposal and Scope

Initial Goals

In the project proposal, we had a few main goals, described briefly below. Goals 1 and 2 deal with basic functionality, ensuring that analyses done on the same datasets gave the same results across `R`'s `pscl` package and our Python implementation. Goals 3 and 4 share a common theme of extending an already functional modeling tool by adding features such as formula processing and common tests for significance between models, making it more in line with common statistical modeling tools available in `R` and Python.

Goal 1.

- Write functions for likelihood, log-likelihood, gradient and maximum likelihood estimation of parameters using the results of the standard Poisson regression fit (obtained via `GLM()`) as starting values.
- Limit to using logit link to simplify calculations
- Supply starting point initially, later add EM for computing starting point.
- Verify algorithm for different datasets

Goal 2.

- Write functions that are common in modeling packages: `residuals()`, `coeff()`, `predict()`, `fitted()`, `covar()`, `logLik()`, `summary()`, `printModel()`
- Include likelihood ratio test for conducting tests of nested models

Goal 3.

- Add formula processing, sanity checks
- Add compatibility for different link functions
- Add Vuong test for comparing zero-inflated Poisson regression fit with standard Poisson regression

Goal 4.

- Repeat above for zero inflated negative binomial, zero inflated geometric regression
- Make package compatible with scikit-learn, make similar to other Python regression functions

Accomplishments

We were able to accomplish nearly all of the goals, with the exception of some of the additional signifiance tests. We implemented all of the necessary calculations required to run the algorithm (Goal 2), including the incorporation of the Expectation-Maximization algorithm for computing a starting point. The results from this computation gave the same numerical results (with one exception when performing zero inflated negative binomial regression - refer to details in the next section). We wrote all of the functions that one would expect when doing numerical statistical modeling, along with the necessary error-checking to safeguard against unexpected behavior. Once basic functionality was confirmed, we extended the compatibility to accommodate other distributions, namely the negative binomial distribution and the geometric distribution. Although we had initially thought to only allow for a logit link, we ended up including probit, complementary log log, cauchy, and log link functions as well.

Some of the goals that we are still working toward are adding the Vuong test along with other likelihood ratio tests for nested models.

IV. Methods

Algorithm Overview

The first level of the algorithm (Figure 2) is the initialization stage which requires user input: a formula that specifies the model (similar to the `lm(y ~ x1 + x2)` syntax in `R`), choice of a link function (logit by default), and weights and offsets (array of ones and zeros, respectively, by default). A distribution can also be specified, from which the user can choose Poisson (default), geometric, or negative binomial. If the starting value is not given, then a Poisson estimate is used. If the `EM` parameter is `True`, then the starting values can be estimated through the expectation maximization (EM) algorithm, which can result in more numerically stable calculations. In the penultimate level, the `fit()` function provides all of the quantities that one might expect after fitting a statistical model: `coefficients`, `residuals`, `fitted_values`, `loglik`, and `vcov`. These are stored in a `ZeroInflated` object.

One of the problems that we encountered when trying to estimate the starting values when `EM = True` for the zero inflated negative binomial (ZINB) model was that the estimates were not converging to the same values that were shown in the `R` output. This was because the EM algorithm for ZINB uses iterative computation of the parameters via regular negative binomial regression and logit (or other

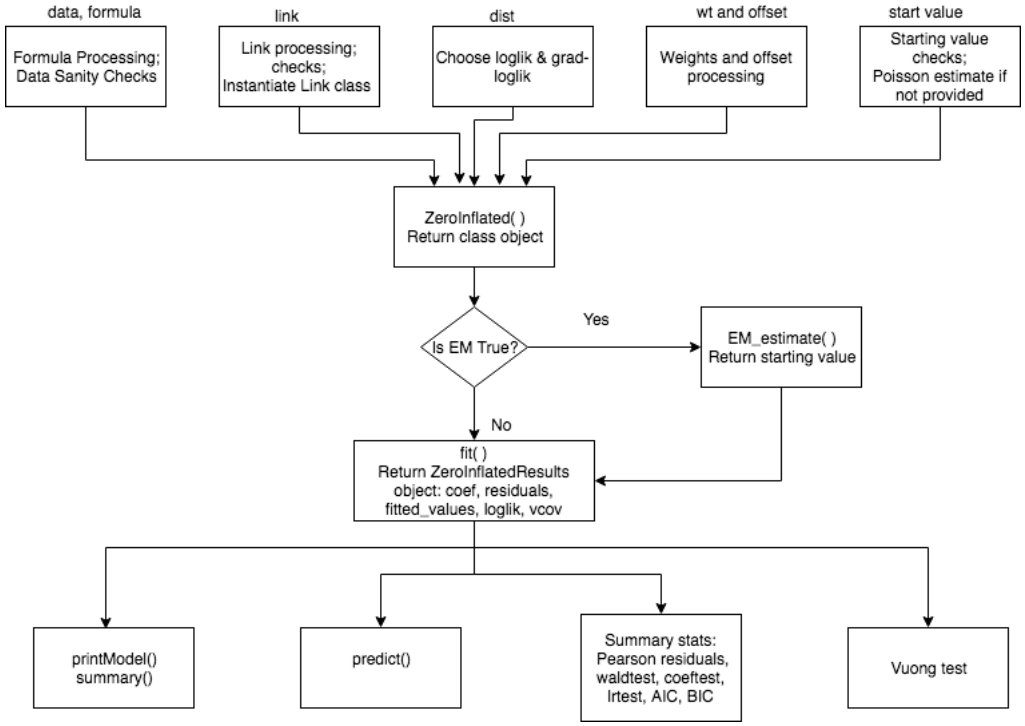

Figure 2: *Algorithm flowchart*

based on the link class) regression in each step. Unlike in `R`, Python's `GLM` regression methods are based on single parameter maximum likelihood estimation. Instead of joint estimation of the dispersion parameter, θ, and μ, maximum likelihood estimation of θ is calculated from the maximum likelihood estimate of μ. When this optimization is done over the course of multiple iterations, the deviance between the joint and marginal estimation remains quite large. Consequently, these differing starting values in Python give results that do not exactly match those provided by `R`. It is worth noting that there is a `NegativeBinomial()` function in Python's `statsmodels` class that supports joint estimation of μ and θ, but it does not, however, have an option to pass in weights, which is needed for EM estimation. Since it is not possible to update the EM equations without changing the weights, we had to use the `GLM` estimates for the EM estimation of parameters.

Code Organization

There were three main classes in total:

1. The `LinkClass` class is the parent class of the 5 classes, namely, `Logit`, `Probit`, `CLogLog`, `Cauchit`, `Log`, corresponding to the binomial link functions. Logit is the default link function.

2. A `ZeroInflated` object is instantiated when the user passes in the dataframe accompanied with the formula string. The string input is processed so that pertinent information about the variables is stored. At this point, no model fitting is done.

3. A `ZeroInflatedResults` object is instantiated when `fit()` is called on a `ZeroInflated` object. The `ZeroInflatedResults` object contains the optimization results: fitted values, log-likelihood, coefficients, residuals, and other summary statistics. For more organized display of the model details and re-

sults, `printModel()` and `summary()` are two such functions that provide very similar output the R's `summary()` and `print()` functions when called on a model object. Note that both `printModel()` and `summary()` must be called on `ZeroInflatedResults` objects and not on `ZeroInflated` objects.

Usage

In general, the classes and member functions can be used as follows:

1. Create an instance of the class `ZeroInflated`, with default params: `dist = 'poisson', link = 'logit'`

- `ZeroInflated(formula, d, params)`

2. Fit the zero-inflated regression model by calling `fit` **on a** `ZeroInflated` **object,** `m`**, and create a** `ZeroInflatedResults` **object**, with default params: `EM = True, method = BFGS`

- `m.fit(params)`

3. Call member functions after fitting the model (`method = summary(), printModel(), predict(), vcov()`)

- `m.fit(params).method`

The argument `formula` is a `string` that specifies the model, similar to the `formula` argument used in R's `lm()` function. The argument `d` is the design matrix.

V. Demonstration

We tested the functionality and accuracy of our zero-inflated Poisson regression package on a dataset of 4406 observations (individuals at least 66 years old who are covered by Medicare). The data can be found at `http://qed.econ.queensu.ca/jae/1997-v12.3/deb-trivedi/`. We use the number of physician office visits, represented by the variable `ofp`, as a means of quantifying the demand for medical care for the elderly, with the following covariates:

- `hosp`: number of hospital stays (0 to 8)
- `health`: self-assessed health status (average, excellent, poor)
- `numchron`: number of chronic conditions (0 to 8)
- `gender`: male/female
- `school`: years of education (0 to 9)
- `privins`: private insurance (yes/no)

As seen in the histogram of the the number of physician office visits in Figure (3), the data set is saturated with zero counts, with more than 50% of the `ofp` values being less than 4.

We omit some of the exploratory analysis of the data (a more thorough study of this dataset can be found in the paper by [Deb and Trivedi, 1997] and in the article by [Jackman, Kleiber, and Zeileis, 2008]), and demonstrate the use of zero-inflated regression in this context. In particular, we compare the usage and results of the routines that we implemented in Python to the counterparts offered in the `pscl` package in `R`.

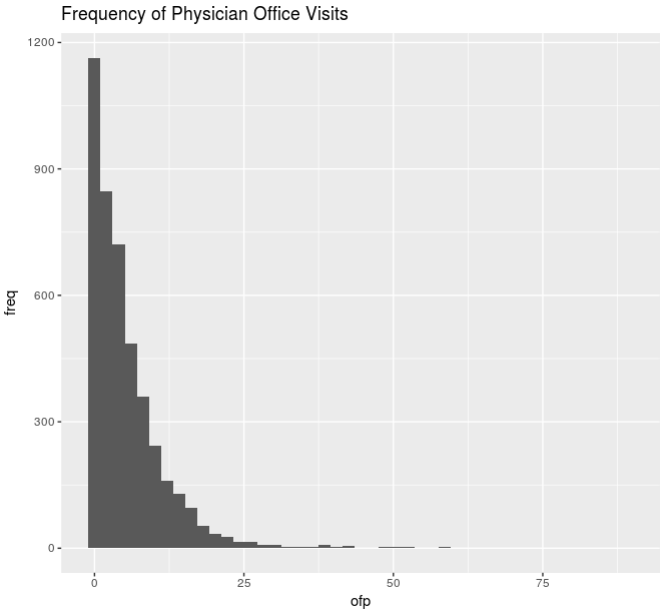

Figure 3: *The distribution of the physician office visits is for the 4406 individuals in the study is heavily skewed right with a disproportionate number of subjects who had zero visits.*

The `ZeroInflated` object requires a `string` argument for the model formula whose format is parallel to that of `R`'s modeling syntax. For example, the appropriate formula to fit a zero inflated Poisson regression model with the response variable and covariates as described above is written as:

```
[1] formula = 'ofp ~ hosp + health +
      numchron + gender + school +
      privins | health'
```

After instantiating a `ZeroInflated` object using the above formula and fitting the model, we can call `summary()` on the `ZeroInflatedResults` object to obtain the coefficients for each of the predictors and corresponding p-values. In Figure 4 and Figure 5 (in the appendix), we see that the results from our Python routine match those calculated by the `zeroinfl()` function in `R`'s `pscl` package.

Other suitable methods for this dataset are zero-inflated geometric regression and zero-inflated negative binomial regression, both of which are supported by our implementation by passing in `dist = 'geometric'` and `dist = 'negbin'`, respectively, when instantiating the `ZeroInflated` object. Fitting the corresponding model, we then obtain the coefficient estimates as shown in Table 2 and Table 3. First considering the zero-inflated geometric regression model, we see that the count model coefficients match with the `R` output up to at least 4 decimal points, whereas the zero counts model's coefficients show a little less accuracy. For the zero-negative binomial regression model, we see accuracy up to 2 decimal places for the count model, and 1 decimal place for the zero counts model. As mentioned before, these differences arise from the support for joint parameter estimation in `R`, whereas we settled for using the single parameter `GLM` estimates.

As is standard with modeling packages in `R`, our Python implementation of zero-inflated Poisson regression also supports prediction. The `predict()` function takes a `ZeroInflatedResults` object, a `type` argument that indicates the desired prediction scale (`'response'`, `'count'`, `'prob'`), `newdata_X` (matrix of new data for the count model), and `newdata_Z` (matrix of new data for the zero-count model). A more detailed overview of individual member functions can be found in the code's documentation.

VI. Conclusion

Our `zeroinfl` package contains all the necessary tools to more easily conduct zero inflated count analysis in Python. Moreover, with the similar argument syntax that it shares with `R`'s modeling functions, we make it easy to compare results across platforms and languages. One area that still requires further development is inference on the models. One such way to test if the zero inflated model is an improvement over standard models is by performing a Vuong test.

For future development, we also note some design considerations. In Python, regression packages usually inherit from the `LikelihoodModel` class which internally performs the necessary otpimization, likelihood ratio tests, and standard error computations. This effectively removes the need to repeat these calculations in any subsequent regression functions themselves. However, we were not able to inherit these methods for this project because many of the attribute names of classes that we defined do not match with those of the `LikelihoodModel` class. Since our project prioritized functionality and accuracy over uniformity, our primary focus was not making our project immediately adaptable with existing Python packages. However, now that most of the project's features are working correctly, it makes sense for our next step to begin making our code more consistent with Python's package conventions.

Finally, in order to fully integrate our code and make it accessible, we would publish our package on Python to assist other statististicians with their analyses and to allow other developers to further extend our work. The `zeroinfl` package that we developed aims to fill in the gaps in Python's modeling toolkit

and add more flexibility to the algorithms currently available. In addition, by implementing zero inflated regression in a manner that is similar to existing R modeling packages and functions, we hope to bridge the gap between R and Python users and allow the developers in both user bases to simultaneously advance scientific computing and make the languages more accessible to other communities. The current state of our development can be accessed on GitHub.

References

[Deb and Trivedi, 1997] Deb P., Trivedi P. K. (1997). Demand for Medical Care by the Elderly: A Finite Mixture Approach. *Journal of Applied Econometrics*, 12:31–336.

[Jackman, Kleiber, and Zeileis, 2008] Zeileis, A., Kleiber, C., Jackman, C. (2009). Regression Models for Count Data in R *Journal of Statistical Software*, 27:1–25.

[Roudbari and Salehi, 2015] Salehi, M., and Roudbari, M. (2015). Zero inflated Poisson and negative binomial regression models: application in education *Medical Journal of the Islamic Republic of Iran*, 29:297

Appendix

Figure 4: *Summary output for the zero-inflated Poisson regression model using the Medicare dataset. Comparing this to the summary output generated by* `R` *for the same model, we see that the computations yield very similar results. Note the Python routine only required 13 iterations to reach convergence in the BFGS optimization.*

```
Count model cefficients (poisson log link):
                     |  Estimate|Std. Error|   z value|  Pr(>|z|)
------------------------------------------------------------------
           Intercept|    1.3937|    0.0245|     56.86|    <2e-16
 health[T.excellent]|   -0.3078|    0.0314|    -9.791|    <2e-16
      health[T.poor]|    0.2542|    0.0177|    14.339|    <2e-16
      gender[T.male]|   -0.0649|    0.0131|    -4.946| 7.5608e-7
      privins[T.yes]|    0.0854|    0.0173|     4.936| 7.9923e-7
                hosp|    0.1591|    0.0061|    26.262|    <2e-16
            numchron|    0.1033|    0.0047|    21.805|    <2e-16
              school|    0.0196|    0.0019|    10.393|    <2e-16


Zero-inflation model coefficients (binomial with logit link):
                     |  Estimate|Std. Error|   z value|  Pr(>|z|)
------------------------------------------------------------------
           Intercept|   -1.7336|    0.0481|    -36.04|    <2e-16
 health[T.excellent]|    0.4749|    0.1455|     3.265| 1.0960e-3
      health[T.poor]|   -0.3992|    0.1466|    -2.724| 6.4570e-3
---
Number of iterations in BFGS optimization: 13
Log-likelihood: -1.629e+4 on 11 Df.
```

Figure 5: *Summary output for the zero-inflated Poisson regression model using* `zeroinfl()` *in* `R`

```
Count model coefficients (poisson with log link):
                Estimate Std. Error z value Pr(>|z|)
(Intercept)      1.393672   0.024506  56.870  < 2e-16 ***
healthpoor       0.254201   0.017726  14.340  < 2e-16 ***
healthexcellent -0.307754   0.031428  -9.792  < 2e-16 ***
gendermale      -0.064859   0.013111  -4.947 7.54e-07 ***
privinsyes       0.085441   0.017310   4.936 7.97e-07 ***
hosp             0.159127   0.006059  26.262  < 2e-16 ***
numchron         0.103280   0.004736  21.806  < 2e-16 ***
school           0.019591   0.001885  10.395  < 2e-16 ***

Zero-inflation model coefficients (binomial with logit link):
                Estimate Std. Error z value Pr(>|z|)
(Intercept)     -1.73364    0.04809 -36.050  < 2e-16 ***
healthpoor      -0.39916    0.14655  -2.724  0.00646 **
healthexcellent  0.47490    0.14542   3.266  0.00109 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Number of iterations in BFGS optimization: 17
Log-likelihood: -1.629e+04 on 11 Df
```

Table 2: *Zero Inflated Geometric Regression (logit link). All of the count estimates are accurate up to at least 4 decimal places. There are minor discrepancies in the zero model's coefficient estimates.*

Count:	**Python**	**R**
Intercept	0.92523	0.92523
health[T.excellent]	-0.34160	-0.34160
health[T.poor]	0.31351	0.31352
gender[T.male]	-0.12678	-0.12678
privins[T.yes]	0.22483	0.22483
hosp	0.22057	0.22057
numchron	0.17603	0.17603
school	0.02689	0.02689
Zero:	**Python**	**R**
Intercept	-24.34361	-24.33868
health[T.excellent]	10.00308	9.99858
health[T.poor]	19.44489	19.43996

Table 3: *Zero Inflated Negative Binomial Regression (logit link). Note the discrepancy between the Python and R coefficient estimates. This is due to the different methods of the MLE estimation.*

Count:	**Python**	**R**
Intercept	0.94181	0.94206
health[T.excellent]	-0.32944	-0.32945
health[T.poor]	0.32891	0.32884
gender[T.male]	-0.12468	-0.12465
privins[T.yes]	0.21839	0.21832
hosp	0.22252	0.22250
numchron	0.17366	0.17363
school	0.02667	0.02667
theta	1.25876	1.25925
Zero:	**Python**	**R**
Intercept	-5.04317	-5.02321
health[T.excellent]	1.08234	1.06984
health[T.poor]	1.61952	1.60087