Journal of Data Science 23 (2), 399–415
April 2025
DOI: 10.6339/25-JDS1183
Computing in Data Science

The Journey to Improve LCPM: An R Package for Ordinal Regression

Roland DePratti[1,*] and Gurbakhshash Singh[1]
[1]Central Connecticut State University, New Britain, Connecticut, USA

Abstract

Recently, the log cumulative probability model (LCPM) and its special case the proportional probability model (PPM) was developed to relate ordinal outcomes to predictor variables using the log link instead of the logit link. These models permit the estimation of probability instead of odds, but the log link requires constrained maximum likelihood estimation (cMLE). An algorithm that efficiently handles cMLE for the LCPM is a valuable resource as these models are applicable in many settings and its output is easy to interpret. One such implementation is in the R package `lcpm`. In this era of big data, all statistical models are under pressure to meet the new processing demands. This work aimed to improve the algorithm in `R` package `lcpm` to process more input in less time using less memory.

Keywords *constrained maximum likelihood estimation; log link; ordinal outcomes; proportional probability model*

1 Introduction

Considerable work and research has been done for models involving ordinal outcomes. McCullagh studied Generalized Linear Models with the logit link which estimates odds McCullagh (1980). An alternative approach is models that estimate probabilities for an ordinal outcome rather than odds. These models use log links versus logit links to permit easier interpretation of results. However, such models introduce a challenge not seen with the logit link models. The log link models have constraints on the parameter space that must be satisfied by the objective function's maximum likelihood estimate (MLE).

Blizzard et al. (2013) introduced the proportional probability model (PPM):

$$\log[P(y \leqslant j|x_k)] = \alpha_j + \sum_{k=1}^{p} x_k\beta_k,$$

which used the log link instead of the logit link to relate ordinal outcomes (y with $j > 2$ categories) to predictor variables (x_k). The PPM assumes that the β_k do not change for each category j with the cumulative probability constraint satisfied by $\alpha_j \leqslant \alpha_{j+1}$ and the probability constraint satisfied by link constraint $\log[P(y \leqslant j|x_k)] \leqslant 0$. Blizzard et al. also provided `SAS` code for maximum likelihood estimation (MLE) subject to these constraints induced by the log link and cumulative probability. Williams (2010) produced a solution using `Stata` and Yee (2024) created a solution in the R `VGAM` package. However, both the Williams and Yee solutions

*Corresponding author. Email: roland.depratti@my.ccsu.edu.

Received October 3, 2024; Accepted April 4, 2025

did not implement the MLE constraint requirement, leaving the burden of the constraint on the users of these packages. Singh and Fick (2020a,b) introduced the function `ppm` in the `lcpm` package in R which uses the function `constrOptim` for constrained MLE of the PPM. The use of `constrOptim` naturally followed from its extensive use in the log-binomial regression model which had similar constraints in the use of a log link to relate predictors to binary outcomes Luo et al. (2014). Andrade eventually developed this approach into an R package called `lbreg` Andrade (2019).

In analyzing a diabetic dataset Clore et al. (2014) that had 100,000+ records and 16 (of 47) selected predictors of outcome length of hospitalization (converted to ordinal scale of no stay, short and long term), package `lcpm` took time to provide constrained MLEs. Using a complete case approach to handle missing data, it was noted that as predictors (with varying % of missingness) changed in the model so did the number of records and runtime of `lcpm`. Similar issues arose with other larger datasets. This started the inquiry into the elements of the package that were leading to increased use of time and memory. Initial thoughts focused on data-specific elements such as the number of predictors, sample size of dataset, and percentage of missing data. Eventually, it grew to profile the R code in `lcpm` to address code bottlenecks.

Several researchers have identified that providing additional data to fit a model will improve the accuracy of the results more than using a different algorithm. In trying to predict the classification of a bank customer's complaint using data from the Consumer Financial Protection Bureau, Schnoebelen found that the increase in accuracy was greater when increasing training data rather than changing the features or algorithm Schnoebelen (2016). When changing the features and algorithm, the greatest mean accuracy increase was 0.02%, but by increasing the training data size the mean accuracy increase was 1.87%. Two other research teams took a broader approach Halevy et al. (2009); Rajaraman (2008). Both papers describe cases where algorithm performance improved by adding additional features. Of course, in the latter case, the quality of the data is important to prevent the curse of dimensionality issues.

The issues with the large diabetes dataset and the work that described the benefits of processing with larger training data raise the importance of an algorithm's ability to handle large amounts of data efficiently. This work aimed to explore ways to improve Singh and Fick's algorithm to process more data in less time and with less memory.

2 Methods

To improve the `ppm` function in package `lcpm`, we: a) used profiling to identify high response time and memory-consuming code blocks; b) developed alternative code; c) tested alternative code; d) confirmed that the changes produced similar quality and accuracy of MLE.

2.1 Profile Code and Alternative Approaches

The `ppm` function has three main tasks: a) preparing the problem data for the optimizer (marshaling), b) optimizing, and c) the objective function used by the optimizer (Figure 1).

The `Rprof` package was used to identify runtime and memory usage for each code block in `ppm`. The initial profiling was completed by assessing the performance of `ppm` on simulated datasets with 16 predictors (p) and five different sample sizes (n): 5,000; 50,000; 500,000; 1 million; and 1.5 million. Details of the simulation are discussed in Section 2.3 for $p > 2$. These tests were performed in both a Linux and Windows environment. Hardware details of each

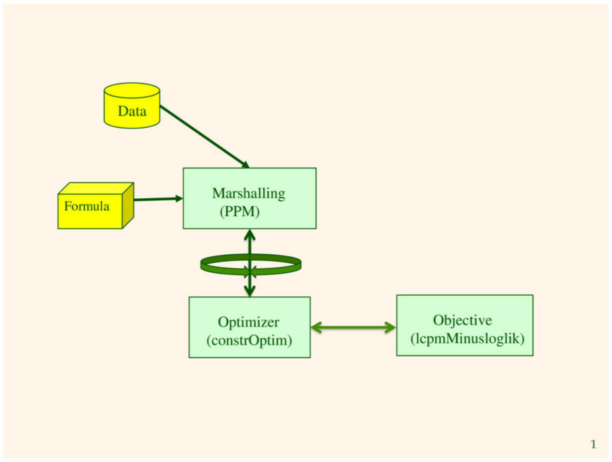

Figure 1: Flow of control.

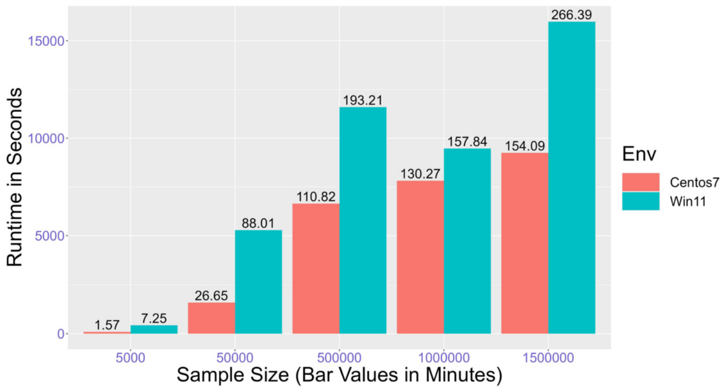

Figure 2: Original code (Test Case A) runtime comparisons.

environment are available in Section 2.4. Profile results can be found in Figure 2. The y-axis represents seconds, while the bar labels represent minutes.

The tests demonstrated that runtimes increased significantly as the sample size increased. On a Windows laptop, it took $\approx$ 4.5 hours to fit a model for a sample size of 1.5 million. The longer runtimes of the profiling in a Windows environment made us realize that the remainder of our experiment needed to be run on an AWS Linux cluster Amazon (2025), which would allow us to run multiple tests in parallel. All future performance comparisons were produced on the AWS environment. By examining the AWS profile data, we identified three bottlenecks: a marshaling bottleneck, an objective function bottleneck, and an optimizer bottleneck.

2.1.1 Marshaling Bottleneck

The `R` package `lcpm` receives two inputs: a dataframe with predictors and responses; and the log likelihood function. The internal optimizer must obtain this data as a summary matrix (see Appendix A and B). Marshaling begins this reformatting by generating a main summary matrix that provides counts for each combination of predictor values and responses. Profiling showed that for the smaller sample sizes ($<$ 50,000), the marshaling code took the most time and used the most memory. The original code used the `model.frame` and `model.matrix` functions in the `stats` package to split the dataframe parameter into a vector of responses and a matrix of predictor rows. Finally, count in the `plyr` package was used to generate the summary matrix. As an alternative, labeled as Test Case B, we used `model.frame` to generate one data structure with both predictors and responses. A group by summary from the `dplyr` library was used to generate the summary matrix. Initial testing showed that the new marshaling code for sample sizes of 50k with 16 predictors decreased marshaling runtime by 13%. Both the original and the alternative code are available in Appendix A.

2.1.2 Objective Function Bottleneck

For larger sample sizes (> 50,000), the optimizer and objective function code took the most time and memory. For our profile test of 10 minutes, the optimizer and objective function consumed 99% of the runtime. The objective function receives a series of matrices containing the data and generates a loss value that the internal optimizer uses to evaluate a series of coefficients. The original objective function was coded in R. Many applications have improved performance by converting problem code from R to C++. The `R` package `RcppArmadillo` provides functions that simplify what is required to interface C++ code with an R application.

To improve the performance of the objective function, the code was converted to C++ code. Using the `RcppArmadillo` package, we built two approaches: a) the C++ compile happens either at runtime (Case C) and b) by calling a package that contained pre-compiled C++ code (Case D). The latter approach was included to measure the benefit of precompiling the C++ code and eliminating the compile overhead at runtime. Both the original and alternative code are available in Appendix B.

2.1.3 Optimizer Bottleneck

Schwendinger et al. (2021) explored several approaches to constraint optimization for the log-binomial regression model including: Augmented Lagrangian, conic optimization, and others. These methods manage the range of probability constraint to guarantee that the sum of the coefficients does not exceed 1. In this manuscript we focus on `constrOptim` and `auglag` which are two easily implemented libraries that can be used for optimization problems with constraints, but they have different approaches and functionalities. Function `constrOptim` applies a logarithmic barrier to enforce the constraints and then `optim` is called Lange (1994). Function `auglag`, in the `alabama` package Varadhan (2023), applies the Augmented Lagrangian method, which adds additional terms to the unconstrained objective function, designed to emulate a Lagrange multiplier. Both optimizers handle linear inequality constraints, while `auglag` can additionally handle nonlinear constraints. Both functions minimize a provided objective function. The original `lcpm` package code incorporated `constrOptim` using Nelder-Mead optimization method.

Since our problem contains linear constraints, each algorithm solves the same optimization problem. In essence, our testing measured each algorithm's runtime performance. To measure the differences in runtime for the same data, we added a call to `auglag` after the original `constrOptim` call. Performance and runtime data were captured for both optimizers.

2.2 Test Approach

Table 1 summarizes all test cases and which code changes were applied to each.

Table 1: Test cases and applied changes.

Test Case	Changes
A	Original Code
B	Marshaling Changes
C	Marshaling Changes and C++ Objective Function Compiled at Runtime
D	Marshaling Changes and C++ Objective Function Pre-Compiled

Each of the four test cases (A (original `ppm`), B, C, and D) represent current code or an alternative approach to the marshaling and objective function code. It was important to determine how these code changes interacted with the two internal optimizers. For that reason, both the `constrOptim` and the `auglag` optimizer were executed for all test cases.

It was also critical to determine how these changes performed under different conditions, including the number of predictors, sample size, and number of times the test was run (i.e. iterations). Test cases were executed using between 2 and 16 predictors, a sample size between 5,000 and 1.5 million, with the tests repeated between 3 and 1000 times.

2.3 Simulation

To assess the impact of bottlenecks and the suggested improvements in code, datasets were simulated from methods discussed in Schwendinger et al. (2021). The following two scenarios for $p = 2$ and $p > 2$ predictors were used to generate simulated data extending these results for three ordinal outcomes:

$$\log[P(y \leqslant j|x_k)] = \alpha_j + \sum_{k=1}^{p} x_k\beta_k.$$

- For p = 2, we used:
  Parameters $\alpha_1 = -0.20$, $\alpha_2 = -0.10$, $\beta_1 = -1$, $\beta_2 = -0.5$.

  $x_1 \sim bin[0.5], x_2 \sim Unif[1, 5].$

- For p > 2, we used:
  first half $\beta_k = -0.1$, and the remaining half are $\beta_k = 0.1$, $\alpha_1 = -1 - p/2$, $\alpha_2 = -p/2$ and independent $x_k \sim bin[0.5]$.

Each test case was executed with multiple numbers of predictors (p for 2 to 16) and different sample sizes (n = 5,000; 50,000; 500,000; 1 million; 1.5 million). Timings were captured after each call to marshal code and for each optimization call. Each iteration of a test case generated new data. The greater the iterations requested, the greater the variety of sample datasets the test case was tested against. The results that follow are based on running the simulation scenario to generate N datasets which are assessed for bias, RMSE, absolute log-likelihood difference, convergence, and boundary issues. The following measurements and formulas were used for that assessment.

- Simulation average runtime (RT).
- Non-convergence (NCR): relative number of times the solver signaled non-convergence or other issue. This typically occurs when the underlying optimizer exceeds the maximum number of iterations of 100,000 without attaining the convergence tolerance specified (10^{-12}).

- Absolute log-likelihood difference (ALLD): average absolute difference between the log-likelihood obtained by the solver and the highest log-likelihood obtained by all solvers $l(\hat{\beta}*)$. Values closer to zero imply that a solver is better at attaining a higher maximum likelihood and a better estimate.

$$ALLD = \frac{1}{N}\sum_{k=1}^{N}|l(\hat{\beta}_k) - l(\hat{\beta}^*)|.$$

- Bias (BIAS) relative bias in percent determined by the following where a BIAS close to zero is an indicator of a better estimation method:

$$BIAS = \frac{100}{N}\sum_{k=1}^{N}\frac{(\hat{\beta}_k - \beta_k)}{(\beta_k)}.$$

- Root mean square error (RMSE): is an assessment of variation in estimation determined by the following, where small RMSE is desired:

$$RMSE = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(\hat{\beta}_k - \beta_k)^2}.$$

- Percent of N with MLE on boundary: either MLE in

$$\Omega_0 = \{(\hat{\alpha}, \hat{\beta})|exp(\hat{\alpha}_j + x_i'\hat{\beta}) \leqslant 0.00001\} \qquad or \qquad \Omega_1 = \{(\hat{\alpha}, \hat{\beta})|exp(\hat{\alpha}_j + x_i'\hat{\beta}) \geqslant 0.99999\}.$$

The boundaries for the LCPM and PPM models are a special consideration and are considered when the MLEs produce fitted probability estimates of 0 or 1. In a numerical optimization scenario, these are assessed using the above Ω_0 and Ω_1 criteria. This metric will also be used to assess consistency across optimizers in finding boundary MLE. At present, no clear data structures are identified that induce MLE on a boundary. However, Singh and Fick (2020a) note several examples in which separability is not necessary.

2.4 Computational Resources

We found the initial profiling using an HP laptop was insufficient for larger simulated scenarios. As such, we moved the majority of the testing to AWS Amazon (2025). Additional details of the hardware and simulation scenarios can be found in Tables 2 and 3, respectively.

R (R Core Team, 2021) version 4.3.1 was used both in the HP environment and AWS. The AWS platform was an AWS ParallelCluster through Amazon's Cloud Formation Service. No test iteration were run in parallel, but rather individual tests were sent to different nodes to speed up a test cycle. The results of these simulations are presented in the following section.

Table 2: Hardware configurations.

Environment	Make/Model	CPUs	Nodes	Specs	Memory	OS
HP Laptop	HP Envy	1	1	12th Gen i7-1260P	16GB	Win 11
AWS	Amazon	16	1 master	m6i.4xlarge	64GB	Centos 6.2
	Amazon	8	4 slaves	m6i.2xlarge	32GB	Centos 6.2

Table 3: Simulation scenario environment.

Scenario	Fig	Test Cases	Sample Size	p	N	Optimizers	Env
Profiling Current State	2	A	5k, 50k, 500k, 1M, 1.5M	16	3	constrOptim	HP AWS
Test	3	A, B, C, D	5k, 50k, 500k, 1M, 1.5M	16	3	constrOptim auglag	AWS
Marshaling	6	A, B, C, D	5k, 50k, 500k, 1M, 1.5M	16	1000	constrOptim auglag	AWS
Iterations Test	4	A, B, C, D	100, 500, 5k	16	1000	constrOptim	AWS
Multiple predictors	5	A, B, C, D	100, 500, 5k	6, 10, 16	3	constrOptim	AWS
Optimizer Comparison	7	A, B	5k, 50k, 500k. 1M. 1.5M	2, 16, 50	1000, 154, 96	constrOptim auglag	AWS

3 Results

In this section, we present results that encompass several scenarios discussed in the Methods section. The first scenario was a profile test looking for a general trend in runtimes across the various test cases. It used 16 predictors, many sample sizes, and 3 iterations of each test. The scenarios that follow the first scenario adjust predictors and iterations to determine the changes on the runtime results of the test cases. The impact of individual code changes is then examined in more detail.

Finally, we compare the performance metrics for both optimizers to guarantee that the `auglag` optimizer did not impact the quality of the results.

3.1 High-Level Findings

Figure 3, below, encapsulates the overall findings using $p = 16$ predictors across multiple sample sizes (n). Due to the broad differences in response across the different components, the y-axis scale was adjusted on each chart to allow them to appear together. The chart shows the timing in each section of the `lcpm` problem across all test cases. Each test was executed (i.e., iterations) 3 times. The reported runtimes in the plots are the average of those 3 iterations.

Looking at the shape of each diagram, we see significant trends. The left panel displays marshaling changes (in cases B, C, and D). It shows that marshaling time decreased across all cases it was applied to. This demonstrates that the alternate coding improved the inefficiencies of the original method and that the increases in sample size did not increase the marshaling runtime greatly.

The middle and right panels of Figure 3 compare objective functions, but we also note

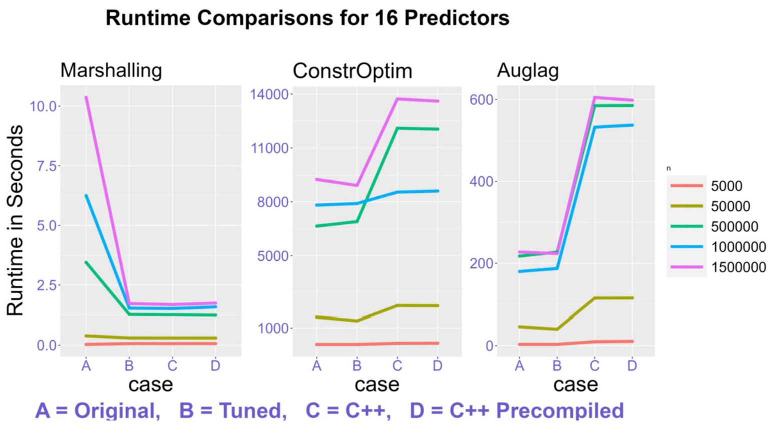

Figure 3: General findings for 16 predictors and 3 iterations.

that the changes (test cases C and D) did not improve runtime performance for either. Both use cases increased timings for all sample sizes, adding significant time for larger sample sizes. We suspect that this is because `Rcpp` must convert the objective function's passed parameters to a structure C++ can understand, which adds overhead not present in the original solution. Also, the optimizer calls the objective function many times to estimate the probabilities, so any overhead would be multiplied by the number of calls. However, more research is needed to confirm these suspicions. The optimizer plots show that the performance impact starts at a sample size of 50k with 16 predictors.

Although the patterns observed were consistent across all sample sizes for all components, there was an unexplainable anomaly. In the testing of `constrOptim`, optimizer runtime jumped up considerably for use cases C and D when the sample size increased from 50,000 to 500,000 and existed for the sample size of 1.5M. However, this same pattern did not exist for 1M sample size. There was an increase, making the code for use cases C and D not beneficial, but the increase was not as large. It is important to remember that this test only completed 3 iterations per use case. It is possible that one of the iterations had especially quick runtime for some yet undetermined reason, which will have to be explored in future work.

3.2 Measuring Consistent Results Based on Iterations

To assess the impact of the number of iterations on the consistency of the evaluation of the objective function, simulation scenarios were run for the 4 test cases using 16 predictors requesting 1000 iterations. For these tests, we used the runtime numbers for the `constrOptim` optimizer. The results are shown in Figure 4. Since a large number of iterations take a long time to run on large sample sizes, the sample size was limited to 5k. We then compared the results of the new test with the results of the earlier test of three iterations (shown in the middle panel of Figure 3).

Both lines represent optimizer runtimes based on a sample size of 5000. This allows us to compare the runtimes for that sample size. The charts show a similar shape. However, the range of runtimes shifted by about 80 seconds for the greater iterations. There also appeared to be a slight decrease in runtime between use cases C and D. However, the jump in runtime between B and the C++ use cases (C and D) still existed, i.e., the pattern across the test cases remained the same.

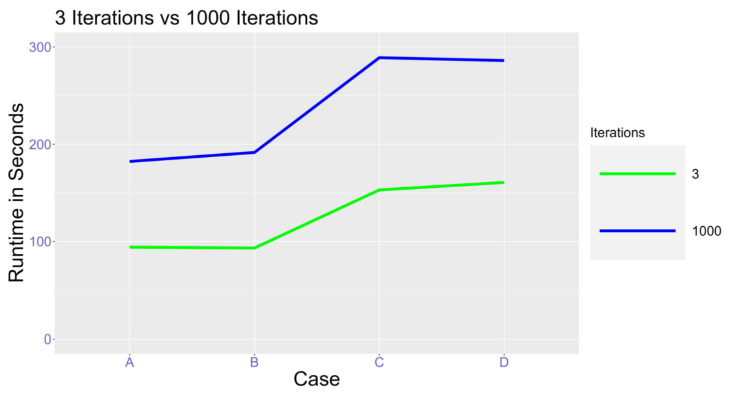

Figure 4: Comparing optimizer runtime results with varying iterations.

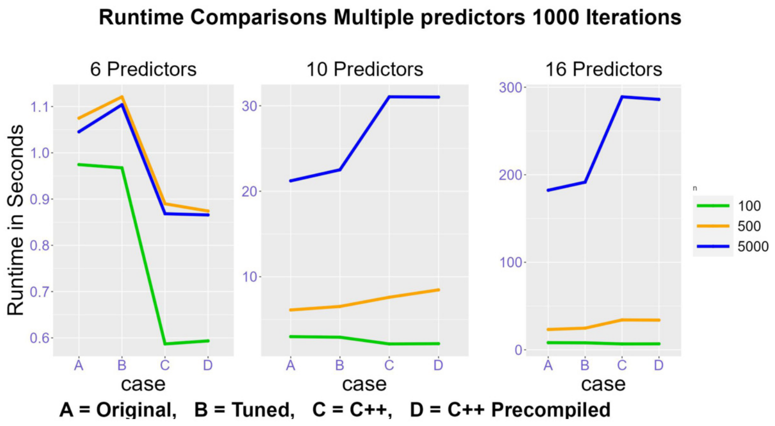

Figure 5: Impact of number of predictors on runtime of test cases.

3.3 Impact of Number of Predictors on Results

To determine the impact of the number of predictors and various sample sizes on the evaluation of the objective function, simulations were run for sample sizes (n) 100, 500, and 5000 using predictors (p) of 6, 10, and 16. Figure 5 shows the average runtimes for the `constrOptim` optimizer in 1000 simulations. As before, the scale is adjusted so that all plots appear together.

This chart shows that the objective function changes (test cases C and D) provided decreasing runtimes for fewer predictors and smaller sample sizes, but the impact was the opposite as predictors and sample size increased. This makes the code less efficient as the problem grows larger. This is evidence that the size of the passed matrix is the determining factor of the runtime of the objective function. Fewer predictors result in a smaller passed matrix. This chart shows that the impact on performance shows up sooner than shown on Figure 3. Here, we see a performance impact as soon as a sample size of 500 and 10 predictors.

3.4 Marshaling Results

By percentage, the marshaling changes produced large improvements. Figure 6 compares marshaling time between cases A and B for 16 predictors across various sample sizes.

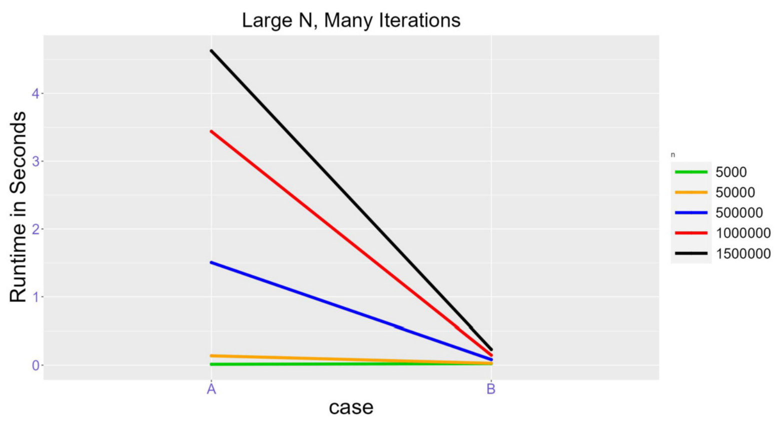

Figure 6: Marshaling runtimes for various sample sizes.

The lines show a significant drop between cases A and B. However, the change in seconds was small. For 6 predictors and a sample size of 1.5 million, the savings amounted to 4 seconds. Figure 3 shown earlier demonstrated that for 16 predictors and 1.5 million samples, the savings were only $\approx$ 8 seconds. These savings are not significant for tasks that run from 2 to 4 hours.

3.5 Objective Function Results

As mentioned, conversion of the objective function to C++ did not result in consistent savings (see Figures 3, 4, and 5). These results reinforced our earlier findings about the changes in the objective function. As a result, test cases C and D were removed from our continued analysis.

3.6 Optimizer Results

The code changes were applied so that for all other use case scenarios (A, B, C, D) both optimizers were called. This provided optimizer performance data under all use cases, since performance data was captured at each call for each optimizer. The reported optimizer performance data provided in the results was the mean under all conditions. As expected, as predictors and sample size increased so did the runtimes. This is true because these factors influence the size of passed matrix and the optimizers' workload. The `auglag` optimizer change provided the largest improvement. Figure 7 compares runtimes for 16 predictors across multiple sample sizes.

For 16 predictors, the average runtime across all sample sizes improved from 2.3 hours to 5.6 minutes. The most significant point is the minimal increases that resulted from increasing the sample size. Function `auglag` increased from .057 to 3.759 minutes when the sample size increased from 5000 to 1.5 million, an increase of 3.7 minutes (6400%), while `constrOptim` increased from 1.57 to 154 minutes (4900%) for the same sample sizes. Although the percentages are large for both, the actual increase in runtime is much larger for `constrOptim` than `auglag` (148 minutes vs 3 minutes). The difference in runtime comparing the optimizers is from the use of the numerical gradient and Hessian by `auglag`, which allows for faster convergence.

3.7 Confirm Quality of Results

The last step was to confirm that our changes did not impact the quality of the returned model. Tables 4 and 5 capture the performance information. Table 4 shows the quality measurements

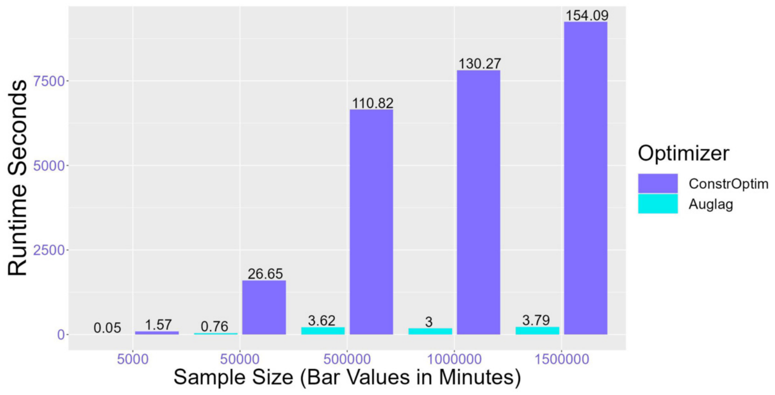

Figure 7: Comparing optimizer runtimes.

Table 4: Performance data for p = 2 and multiple sample sizes.

Case	N	Iterations	Method	Bias %	RMSE	ALLD	Ω_1
A	5k	1000	constrOptim	−3.03	0.26698	0	0
		1000	Auglag	−2.97	0.26701	.001091	0
A	50k	1000	constrOptim	0.74	0.15182	0	0
		1000	Auglag	0.81	0.15182	0.00124	0
A	500k	1000	constrOptim	−0.32	0.08486	0	0
		1000	Auglag	−0.25	0.08487	0.13169	0
A	1 M	154	constrOptim	0.28	0.71364	0	0
		154	Auglag	0.30	0.71382	0.23573	0
A	1.5 M	96	constrOptim	0.61	0.06435	0	0
		96	Auglag	0.69	0.06435	0.17773	0

for p = 2 when examining the optimizer executions for case A with various sample sizes across both optimizers. Table 5 shows runtime, convergence, and boundary performance for the two optimizers across various predictors. The statistics are means across all iterations. In these tables, Ω_0 represents the number of times that the fitted probability values approached 0 and Ω_1 represents the number of times that the fitted probability values approached 1.

In Table 4 the BIAS and RMSE are very close across the two optimizers regardless of the sample size. This implies that the optimizers are providing comparable MLE and RMSE. The ALLD was very small with a slight edge to `constrOptim` as it provided the higher maximum likelihood consistently. There was no non-convergence for either optimizer or boundary MLE. These results are encouraging that on most metrics, the results are similar for the optimizers.

Table 5 shows the performance data from $p > 2$. A test with p = 50 was added to determine its impact on the `constrOptim` and `auglag` comparison. ALLD measurements were very close with a slight advantage (smaller ALLD) for `auglag` as number of predictors (p) increases. The function `constrOptim` had 44 occurrences of non-convergence, and `auglag` did not have a non-convergence. Both optimizers encountered boundary cases (Ω_1) where the MLE provided the

Table 5: Performance data for p > 2 and multiple sample sizes.

p	Method	Avg Time	Savings	ALLD	Non-conv	Ω_0	Ω_1
6	constrOptim	1.0987		0.0721	0	0	5920 / 7200
	Auglag	0.1679	−85.17 %	0.0637	0	0	5912 / 7200
10	constroptim	11.8208		0.0044	16	0	3062 / 6200
	Auglag	0.6011	−94.74 %	0.0000	0	0	3054 / 6200
16	constrOptim	70.0113		0.3753	20	0	1209 / 3627
	Auglag	1.6763	−97.47 %	0.0000	0	0	1201 / 3627
50	constrOptim	871.0984		0.9687	8	0	39 / 43
	Auglag	6.1620	−99.29 %	0.0000	0	0	39 / 43

fitted probability of 1. For simulations, `auglag` encountered fewer of these boundaries. Although these points are marginally favoring `auglag`, the major improvement is the average runtime. This clearly favored `auglag` as the number of predictors increased.

In general, the results show that both optimizers provided similar accuracy and precision in estimating β coefficients, given the same data. Therefore, `auglag`'s ability to perform the task faster did not negatively impact its results.

4 Conclusions

Profiling the current `lcpm` package showed that it took between ≈ 2 hours (500k sample size) and ≈ 2.5 hours (1.5M) to fit a model for sample sizes greater than $500k$. Three runtime bottlenecks were discovered during profile testing of the current `lcpm` code: marshaling, objective function, and optimizer. Four alternative code approaches were developed to improve runtime (one for marshaling, two for objective function, and one for optimizer). The marshaling alternative approach showed a considerable percentage saving, but a small real savings in time, i.e., 8 seconds. The objective function alternatives provided savings only on small sample sizes and actually negatively impacted runtime for sample sizes > 5k. The `auglag` optimizer change provided the largest improvement. For $p > 2$, we obtained significant improvements in runtime (85% to 99%). For 16 predictors, the average runtime across all sample sizes improved from 2.3 hours to 5.6 minutes.

Some potential explanations for the ineffectiveness of the objective function alternatives to improve runtime centered around the need for `Rcpp`'s need to translate the large matrix parameters due to the large sample sizes. Validation of this supposition remains for further study.

5 Discussion

In this work we have explored various bottlenecks and improvements to find that the change in optimizer provided the greatest improvement. The use of the Nelder-Mead search method subject to constraints in `constrOptim` and the package `lcpm`, led to considerable slow down in setting with many predictors. In the calls to each optimizer, we did not provide a gradient function due to our inability to develop an algorithm that could develop a gradient for all use cases. The function `constrOptim` using a Nelder-Mead search can determine the minimum without using a gradient function, while `auglag` evaluates the gradient numerically. We wonder if that resulted in the need for more iterations to get convergence in `constrOptim` resulting in the longer runtimes and non-convergence in settings with large sample sizes with many predictors. Schwendinger et al. find similar results comparing `constrOptim` and `auglag` but are able to address additional optimization methods that have not or cannot be translated to the LCPM or PPM settings.

In exploring the objective function bottleneck, we approached it using package `RcppArmadillo` in the hopes that the conversion to C++ would improve computational efficiency. The matrix conversion is thought to play the biggest role in slowing this approach down. We would like to explore why the objective function changes did not improve runtimes as expected. We wish to further investigate why the small sample size simulation saw initial improvement but did not carry forward to larger sample sizes.

The final exploration is to understand the causes and nature of boundary issues. While Albert & Anderson (1984) found necessary and sufficient conditions for the existence of finite MLE for the logit link, additional exploration still remains for the conditions of both non-finite MLE and boundary MLE for the log link in the LCPM and PPM. It is essential to identify settings where inference may be inappropriate.

Supplementary Material

The supplementary zip file contains all source code (both R and CPP), an R package used to run test case D, a sample windows batch script to run the code and 3 sample csv files that are input into the batch file. There is also a readme file included with more explanation on how to use these files.

Appendix

A Marshalling Code Differences

The following function executed the marshaling code for each test case. Test case A is the original code. Test cases B, C, and D used the new marshaling code.

```
generateSumdat <- function(incase = 'A', fmla, data) {

        if (incase == 'A') {
                Y<-model.frame(fmla, data)[, 1]
                X<- model.matrix(fmla, data)
                mydata<-na.omit(as.data.frame(cbind(Y,X[,-1])))
                colnames(mydata)<-c("y",colnames(X)[-1])
                sumdat<-plyr::count(mydata)

        } else if (incase %in% list('B', 'C', 'D')) {
                library(dplyr)
                mydata <- model.frame(fmla, data,na.action = na.omit)
                colnames(mydata)[1] = 'y'
                x.col.count <- length(colnames(mydata)
                                    [colnames(mydata) !='y'])
                mydata$y <- as.numeric(as.factor(mydata$y))
                out <- names(Filter(is.factor,mydata))[]
                        if (length(out) > 0) {
                    colnames <- out[!(out %in% c('y'))]
                    mydata <- fastDummies::dummy_cols(mydata,
                    select_columns=colnames,
                                        remove_selected_columns=TRUE,
                                        remove_first_dummy=TRUE)
                }
                group <- colnames(mydata)
                sumdat = data.frame(mydata %>% group_by_at(group) %>%
                                        dplyr::summarise(freq = n()))

            }
            return (sumdat)
}
```

B Objective Function Differences

B.1 Original Objective Function Code (Test Case A and B)

```
minusloglikppm<-function(betapar,Xa1,XaJ, Xaj1, Xaj2,Xaf,...){
        #first ordinal value
        loglik.1<-crossprod(Xa1[,ncol(Xa1)], Xa1[,-ncol(Xa1)]%*%
                              betapar)
        #last ordinal value
        loglik.J<-crossprod(XaJ[,ncol(XaJ)],log(1-exp(XaJ[,-
                                    ncol(XaJ)]%*%betapar)))
        # all other ordinal values
        loglik.j<-crossprod(Xaj1[,ncol(Xaj1)],log(exp(Xaj1[,-
                              ncol(Xaj1)]%*%betapar)-exp(Xaj2[,-
                              ncol(Xaj2)]%*%betapar)))
                loglik.i<- as.numeric(loglik.1)+as.numeric(loglik.J)+
                    as.numeric(loglik.j)
        minusloglik<--loglik.i
        minusloglik
}

h<-function(betapar,Xa1,XaJ,Xaj1,Xaj2,Xaf,...){

        out <- Xaf %*% betapar
        return(out)
}
```

B.2 New C++ Objective Function Code (Test Case C and D

The following code was used for both test cases C and D. For test case C, the code was compiled at runtime. For test case D, the code was pre-compiled and executed from a package.

```
#include <RcppArmadillo.h>
//[[Rcpp::depends(RcppArmadillo)]]

using namespace Rcpp;

//[[Rcpp::export()]]

double minusloglikppm100( arma::vec & betapar, arma::imat & Xa1,
                          arma::imat & XaJ, arma::imat
                          & Xaj1, arma::imat & Xaj2) {
                arma::uword Xa1_cols = Xa1.n_cols;
                arma::uword Xa1_rows = Xa1.n_rows;
                double loglik_1 = as_scalar(Xa1.col(Xa1_cols -1).t() *
            (Xa1.submat(0,0,Xa1_rows-1, Xa1_cols -2) * betapar));
                arma::uword XaJ_cols = XaJ.n_cols;
                arma::uword XaJ_rows = XaJ.n_rows;
                double loglik_J = as_scalar(XaJ.col(XaJ_cols -1).t()
                * log(1 - exp(XaJ.submat(0,0,XaJ_rows-1,
                Xa1_cols -2) * betapar)));
                arma::uword Xaj1_cols = Xaj1.n_cols;
                arma::uword Xaj1_rows = Xaj1.n_rows;
                arma::uword Xaj2_cols = Xaj2.n_cols;
                arma::uword Xaj2_rows = Xaj2.n_rows;
            double loglik_j = as_scalar(Xaj1.col(Xaj1_cols -1).t()
                * log(exp(Xaj1.submat(0,0,Xaj1_rows-1, Xaj1_cols -2)
                * betapar) - exp(Xaj2.submat(0,0,Xaj2_rows-1,
                Xaj2_cols -2) * betapar)));
            return arma::as_scalar(-(loglik_J + loglik_1
                                    + loglik_j));
}

//[[Rcpp::export()]]
arma::vec h100( arma::vec & betapar, arma::imat & Xaf) {
                return arma::vec(Xaf * betapar);
}

//[[Rcpp::export()]]
arma::vec apm_deriv100( arma::vec & betapar) {
                return betapar;
}
```

Acknowledgement

We would like to thank Tom King from CCSU Information Technology for his design and implementation of the AWS cluster. His advice was critical for the completion of this project.

References

Albert A, Anderson JA (1984). On the existence of maximum likelihood estimates in logistic regression models. *Biometrika*, 71(1): 1–10. https://doi.org/10.1093/biomet/71.1.1

Amazon (2025). AWS ParallelCluster. https://docs.aws.amazon.com/parallelcluster/latest/ug/cloudformation-v3.html. Accessed 02-13-2025.

Andrade B (2019). lbreg: Log-binomial regression with constrained optimization. https://CRAN.R-project.org/package=lbreg. R package version 1.3.

Blizzard CL, Quinn SJ, Canary JD, Hosmer DW (2013). Log-link regression models for ordinal responses. *Open Journal of Statistics*, 3: 16–25. https://doi.org/10.4236/ojs.2013.34A003

Clore J, Cios K, DeShazo J, Strack B (2014). Diabetes 130-US hospitals for years 1999–2008. UCI Machine Learning Repository. https://doi.org/10.24432/C5230J.

Halevy A, Norvig P, Pereira F (2009). The unreasonable effectiveness of data. Google Research. https://static.googleusercontent.com/media/research.google.com/en//archive/people/peter/papers/UnreasonableEffectivenessOfData.pdf.

Lange K (1994). An adaptive barrier method for convex programming. *Methods and Applications of Analysis*, 1(4): 392–402. https://doi.org/10.4310/MAA.1994.v1.n4.a1

Luo J, Zhang J, Sun H (2014). Estimation of relative risk using a log-binomial model with constraints. *Computational Statistics*, 29: 981–1003. https://doi.org/10.1007/s00180-013-0476-8

McCullagh P (1980). Regression models for ordinal data. *Journal of the Royal Statistical Society, Series B (Methodological)*, 42(2): 109–142. https://doi.org/10.1111/j.2517-6161.1980.tb01109.x

R Core Team (2021). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/.

Rajaraman A (2008). More data than usual. https://anand.typepad.com/datawocky/2008/03/more-data-usual.html. Accessed: 2025-02-14.

Schnoebelen T (2016). More data beats better algorithms. https://www.datasciencecentral.com/more-data-beats-better-algorithms-by-tyler-schnoebelen/?utm_source=chatgpt.com. Accessed: 2025-02-08.

Schwendinger F, Grun B, Hornik K (2021). A comparison of optimization solvers for log binomial regression including conic programming. *Computational Statistics*, 36: 1721–1754. https://doi.org/10.1007/s00180-021-01084-5

Singh G, Fick GH (2020a). Ordinal outcomes: A cumulative probability model with the log link and an assumption of proportionality. *Statistics in Medicine*, 39: 1343–1361. https://doi.org/10.1002/sim.8479

Singh G, Fick GH (2020b). LCPM: Ordinal outcomes: Generalized linear models with the log link. https://CRAN.R-project.org/package=lcpm. R package version 0.1.1.

Varadhan R (2023). alabama: Constrained nonlinear optimization. https://CRAN.R-project.org/package=alabama. R package version 2023.1.0.

Williams R (2010). Fitting heterogeneous choice models with oglm. *Stata Journal*, 10(4): 540–567. http://www.stata-journal.com/article.html?article=st0208.4. https://doi.org/10.1177/

1536867X1101000402

Yee TW (2024). VGAM: Vector generalized linear and additive models. https://CRAN.R-project.org/package=vgam. R package version 1.1-11.