Latent Factor Analysis and Regression of Multivariate Poisson Lognormal Counts using Amortized Variational Inference

Ananyapam De

MS Thesis Presentation

Under supervision of Dr. Johannes Soeding

Outline

- Quick Recap and Methodology for PLNmodels
- Our Methodology
- Extension to other GLMM's
- Problems with initialization and optimization and fixes
- Speed Optimization and Time Complexity
- Results
  - A sample demonstration
  - Simulation Study
  - Real datasets
  - Runtime analysis
- Conclusion and Future directions

Quick Recap

- Model Parameters:

$$\mu \in \mathbb{R}^{\mathbb{D}}$$
$$\mathbf{\Sigma} \in \mathbb{R}^{\mathbb{D}\times\mathbb{D}}$$
$$\mathbf{B} \in \mathbb{R}^{\mathbb{D}\times\mathbb{Q}}$$

- Latent variables:

$$z_i \in \mathbb{R}^{\mathbb{D}}$$

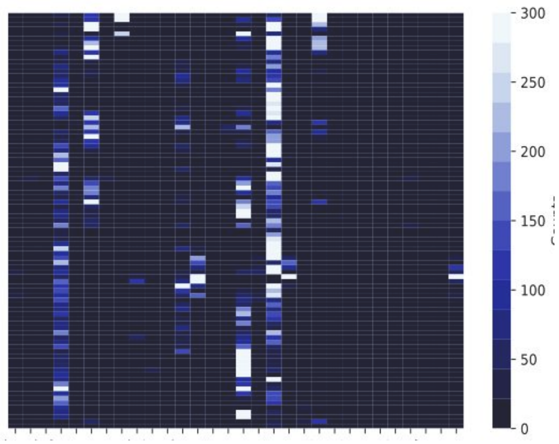
- Covariates:

$$\mathbf{x_i} \in \mathbb{R}^{\mathbb{Q}}$$

- Response variable:

$$\mathbf{y} \in \mathbb{R}^{\mathbb{D}}$$

$$z_i \sim \mathcal{N}(\mu,\, \mathbf{\Sigma} + \mathbf{B}\mathbf{x_i})$$
$$\mathbf{y} \sim \mathrm{Pois}(e^{z_i})$$



Methodology: PLNmodels

- Two step procedure - E step and M step.
- E step requires *p(Z|Y),* which is intractable for the Poisson Lognormal model.
- Resort to *variational approximation.*
- Used a Gaussian with a diagonal covariance as the variational distribution learnt separately for each sample.
- We use *amortization* for this to learn a set of pan-sample parameters, drastically reducing the number of parameters to learn.

Quick Recap: Our Methodology

$$\mathrm{ELBO}(\boldsymbol{\theta}, \phi) = \sum_{n=1}^{N} \mathbb{E}_{q(\mathbf{z}_n|\mathbf{y}_n, \mathbf{x}_n, \phi, \psi_i)} \left[\ln \frac{p(\mathbf{y}_n, \mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\theta})}{q(\mathbf{z}_n | \mathbf{y}_n, \phi)} \right]$$

$$\text{Variational distribution: } q(\mathbf{z}_n|\mathbf{y}_n, \mathbf{x}_n, \phi, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{z}_n | \boldsymbol{\mu}_n, \mathbf{S}_n)$$

$$\mathbf{S}_n = (\boldsymbol{\Lambda}_n + \boldsymbol{\Sigma}^{-1})^{-1}$$

$$\boldsymbol{\mu}_n = \mathbf{S}_n \left(\boldsymbol{\Lambda}_n \mathbf{m}_n + \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu} + \mathbf{B}\mathbf{x}_n) \right)$$

Amortization $\Rightarrow$

$$\boldsymbol{\Lambda}_n = \mathrm{diag}\left(\lambda_{ni}(y_{ni}, \phi) \right) \in \mathbb{R}^{D \times D}$$

$$\mathbf{m}_n = \left(\eta_{ni}(y_{ni}, \phi) \right) \in \mathbb{R}^{D}$$

Amortization

- Laplace Approximation

$$\eta_i(y_i, \phi) = \phi_{i,0} \ln\left(e^{\phi_{i,1}} + y_i\right)$$

$$\ln \lambda_i(y_i, \phi) = \phi_{i,2} \ln\left(e^{\phi_{i,3}} + y_i\right)$$

- Neural Network

$$\begin{pmatrix} \eta_i - \ln(0.5 + y_i) \\ \ln \lambda_i - \ln(0.5 + y_i) \end{pmatrix} \leftarrow \mathrm{dense}(\mathrm{lin}, 2, H) \circ \left(\mathrm{dense}(\mathrm{atan}, H, H)\right)^L \circ \mathrm{dense}(\mathrm{atan}, H, 2)$$

We use a Shallow neural network with H = 6 and L = 2.

No more stochasticity!

$$\text{ELBO}(\boldsymbol{\theta},\boldsymbol{\phi}) = \sum_{n=1}^{N}\left[\underbrace{\mathbb{E}_{q_n}\left(\sum_{i=1}^{D}\ln p\left(y_{ni}|\mu_i^y = g^{-1}(z_{ni}),\boldsymbol{\theta}\right)\right)}_{\text{Expected Conditional Log Probability: } \rho(\boldsymbol{\theta},\boldsymbol{\phi})} - \underbrace{\mathcal{H}\left(\boldsymbol{q}_n,\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})\right)}_{\text{Cross Entropy}} + \underbrace{\mathcal{H}(\boldsymbol{q}_n)}_{\text{Entropy}}\right]$$

When p is the Poisson distribution

$$\rho(\boldsymbol{\theta},\boldsymbol{\phi},\boldsymbol{\psi}) = \sum_{i=1}^{D}\sum_{n=1}^{N}\left[y_{ni}m_{ni} - e^{m_{ni}+S_{nii}/2}\right]$$

Properties of the Gaussian

- Entropy

$$\mathcal{H}\Big(\mathcal{N}(\boldsymbol{\mu}_n, \mathbf{S}_n)\Big) = \frac{1}{2}\ln|2\pi e \boldsymbol{\Sigma}_n| = \frac{1}{2}\ln|\boldsymbol{\Sigma}_n| + \text{const}$$

- Cross Entropy

$$\mathcal{H}\Big(\mathcal{N}(\boldsymbol{\mu}_n, \mathbf{S}_n), \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})\Big) = \frac{1}{2}\Big[\ln|\boldsymbol{\Sigma}| + (\boldsymbol{\mu}_n - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu}_n - \boldsymbol{\mu}) + \text{Tr}\left\{\boldsymbol{\Sigma}^{-1}\mathbf{S}_n\right\}\Big] + \text{const}$$

Extension to other GLMM's

- If $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ then $z_i \sim \mathcal{N}(\mu_i, \Sigma_{ii})$
- Gauss-Hermite Quadrature

$$\int_{-\infty}^{+\infty} e^{-x^2} f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

n is the number of sample points used

w_i are the weights computed using Hermite Polynomials

Speed Optimization and Time Complexity

- Woodbury identity

$$(\mathbf{A} + \mathbf{UCV})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}\left(\mathbf{C}^{-1} + \mathbf{VA}^{-1}\mathbf{U}\right)^{-1}\mathbf{VA}^{-1}$$

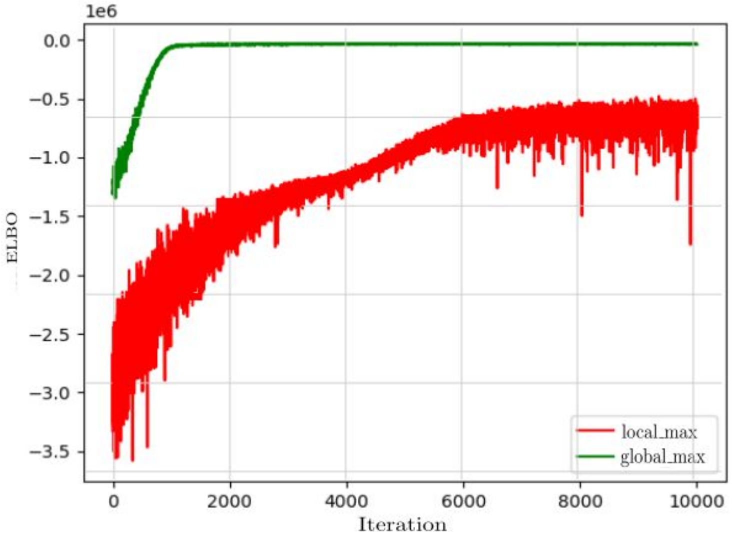
- Weinstein–Aronszajn identity

$$|\mathbf{I}_N + \mathbf{AB}^\top| = |\mathbf{I}_K + \mathbf{A}^\top\mathbf{B}|$$

Final Time Complexity: $O(NDK^2 + NDQ)$

Problems with initialization and optimization

- Bad initial values can lead to local maximas of the ELBO and return ridiculous parameter estimates!
- These estimates can correspond to parameters running off to infinity or leading to singular covariance matrices
- L-BFGS is quite fast but can often cause convergence issues



Problems with initialization and optimization: Fixes

- Make heuristic estimates of the parameters before starting the optimization. Use method of moments estimators.
- Optimize quantities by considering their constraints. (For example instead of optimizing D , we optimize $eps + ln(D)$ which constrains D to only have positive values).
- Neural Network should have low weights and the phi's should be initialized with reasonable values
- Include a strong and decaying prior in the ELBO which would heavily penalize large values of the parameters.
- Use ADAM for optimization instead of L-BFGS.

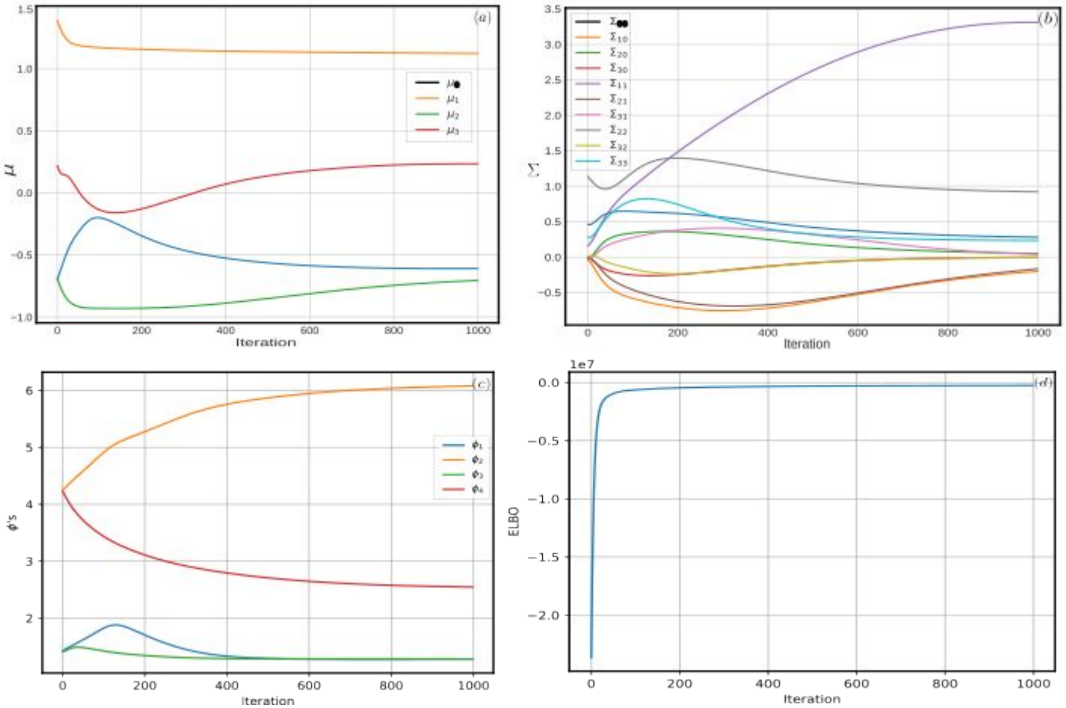
Results: Sample Demonstration

$$N = 50000$$
$$\mathbf{B} = \mathbf{0}$$
$$D = 4$$

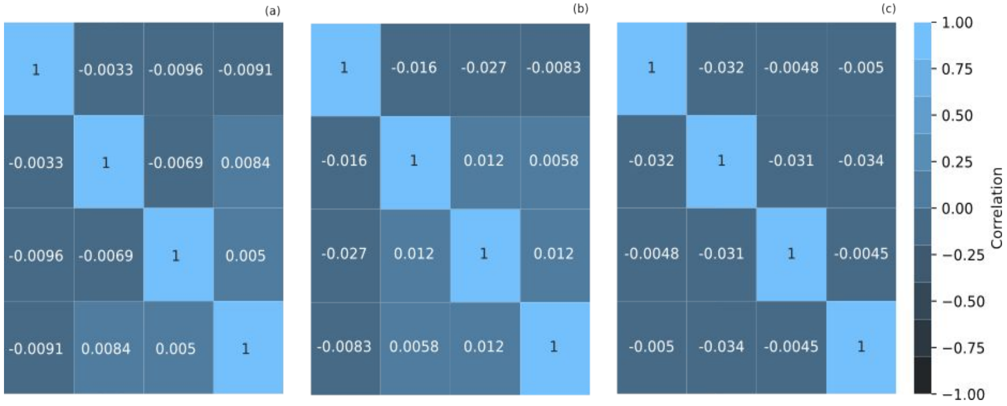
μ and Σ are known

Initialization:

$$\mu \sim \mathcal{N}(0,\, \sigma^2)$$
$$\mathbf{L} \sim \mathcal{N}(0,\, \sigma^2)$$
$$\mathbf{D} \sim \exp(\mathcal{N}(0,\, \sigma^2))$$
$$\sigma = 0.1$$
$$t = 1000$$



Results: Sample Demonstration



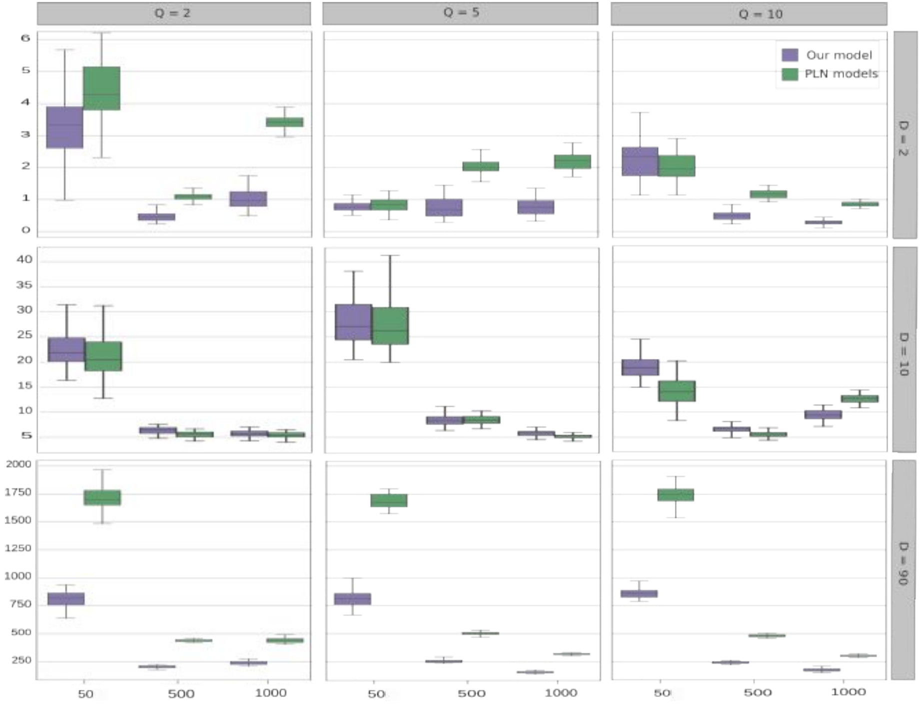
Correlation of Counts

Our Model

PLNmodels

Metric	Our Model	PLNmodels
μ_{MSE}	0.073	3.158
μ_{bias}	0.152	0.641
Σ_{MSE}	11.31	175.368
Σ_{bias}	0.320	2.13

Results: Simulation Study



$$N \in \{50, 500, 1000\}$$
$$D \in \{2, 10, 90\}$$
$$Q \in \{2, 5, 10\}$$
$$\mathbf{X} \sim \mathcal{N}(0,\ 1)$$
$$R\ =\ 50$$
$$\mathbf{Y}^{(\mathbf{1})},\ \mathbf{Y}^{(\mathbf{2})},\ \ldots,\ \mathbf{Y}^{(\mathbf{R})}$$

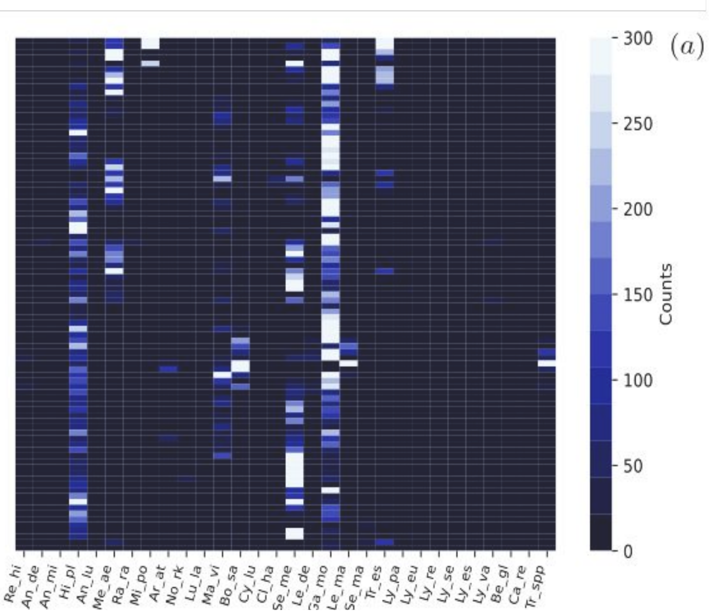
Our model brings down MAE drastically by 50% !

Real Datasets

- Barent's Data - describes the assemblages and distributions of 30 fish species in the southwestern and central part of the Barents Sea with the covariates latitude, longitude, temperature and depth
- Oaks Data - includes information on the abundance of 114 taxa, comprising of 66 bacterial OTUs (Operational Taxonomic Unit) and 48 fungal OTUs, across a total of 116 samples.
- Problems:
  - Determining performance is difficult due to absence of true estimates.
  - Choosing K can be a challenge. For now, we choose it by inspecting the dataset and coming up with a reasonable K.

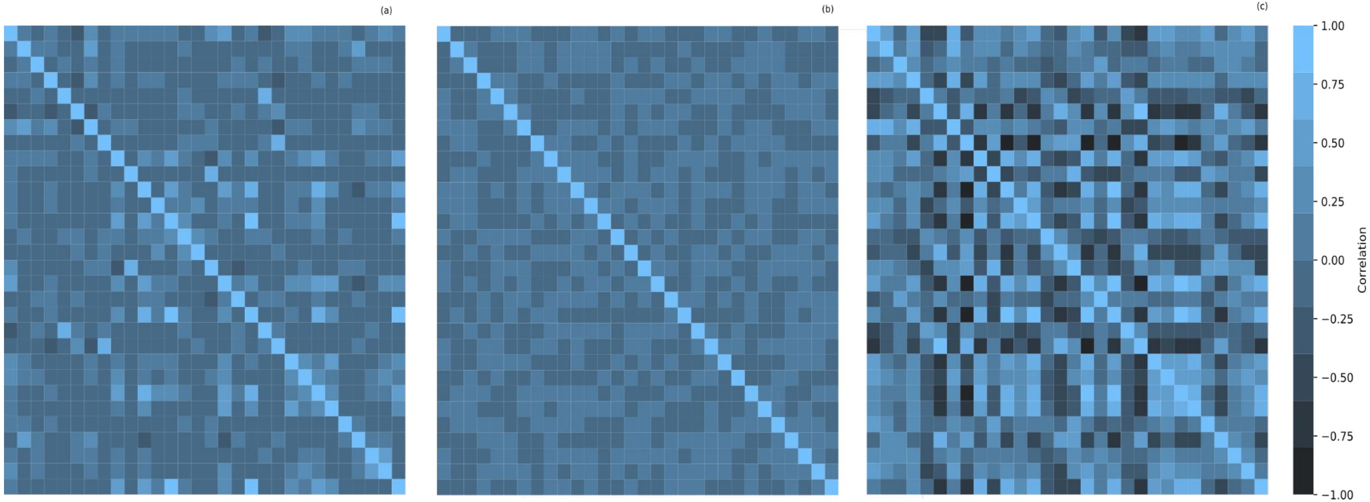
Barents Dataset

- Hyperparameters/ data set configuration:
  - N = 89
  - D = 30
  - Q = 4
  - K = 5



Results: Barents

Correlation Heatmap

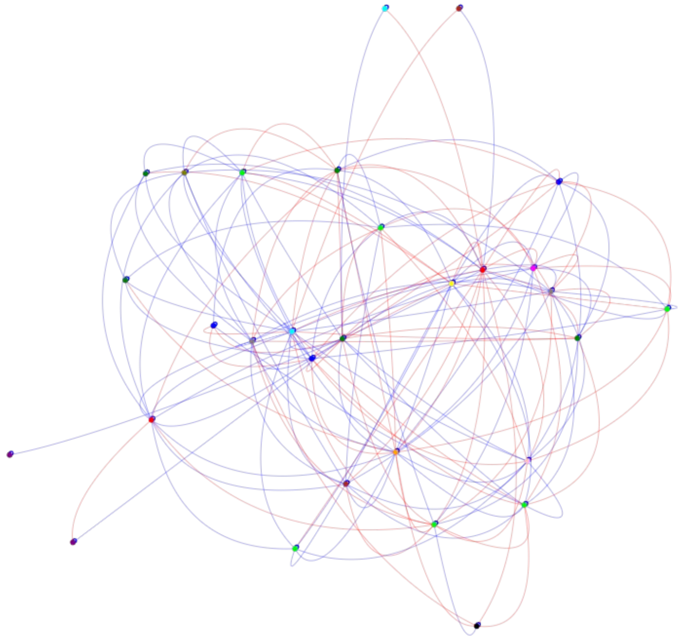


Correlation of Counts

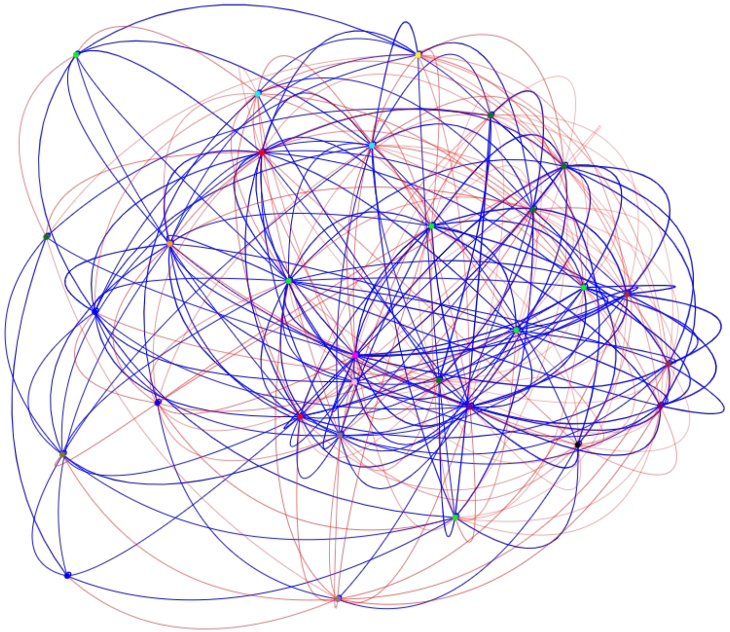
Our Model

PLNmodels

Barents: Network Structure



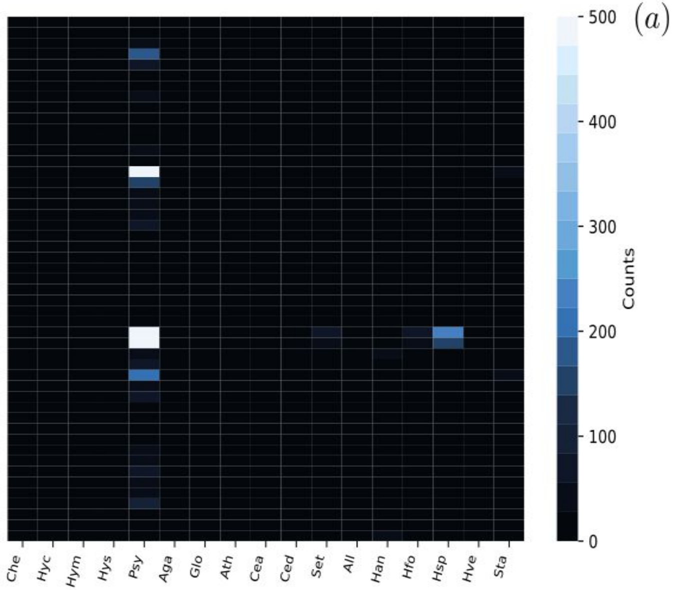
Our Model



PLNmodels

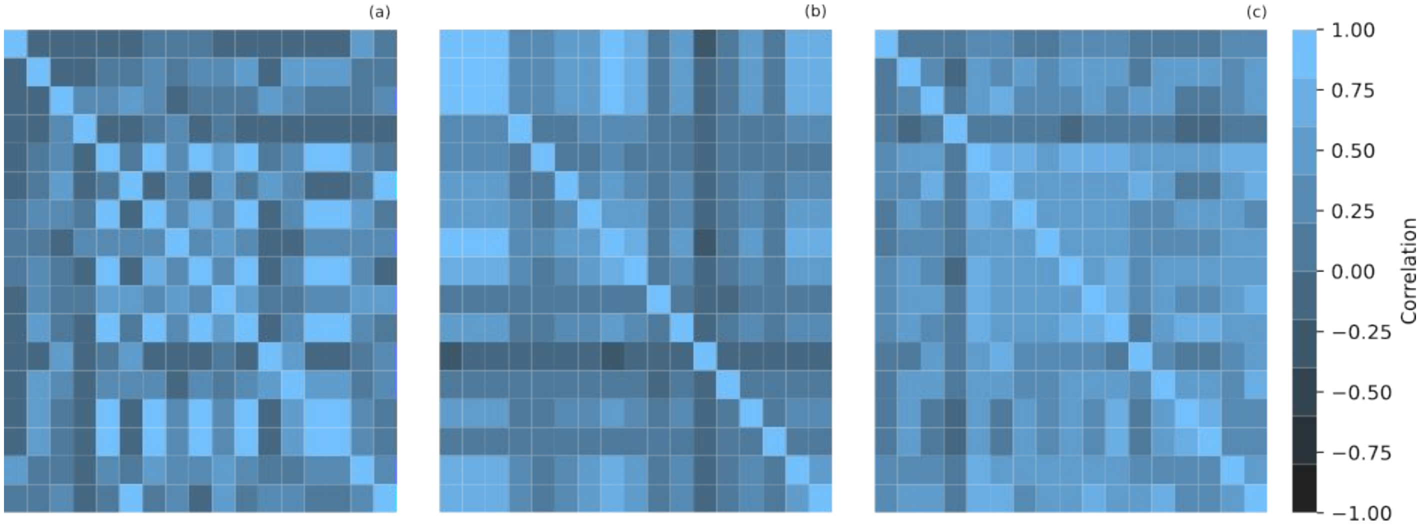
Trichoptera Dataset

- Hyperparameters/ data set configuration:
  - N = 49
  - D = 17
  - Q = 7
  - 1 factor



Results: Trichoptera

Correlation Heatmap



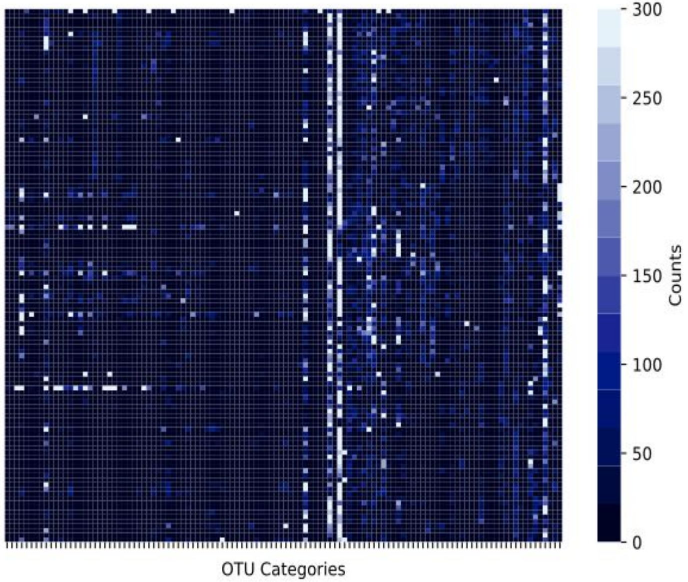
Correlation of Counts

Our Model

PLNmodels

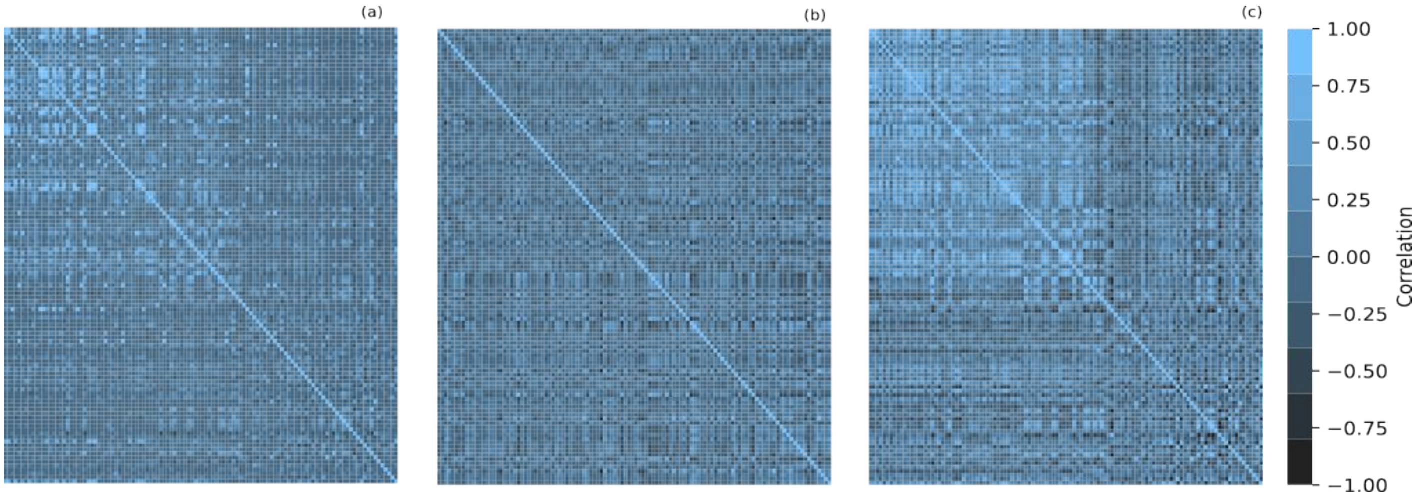
Results: Oaks

- Hyperparameters/ data set configuration:
  - N = 116
  - D = 114
  - Q = 11
  - 3 factors



Results: Oaks

Correlation Heatmap



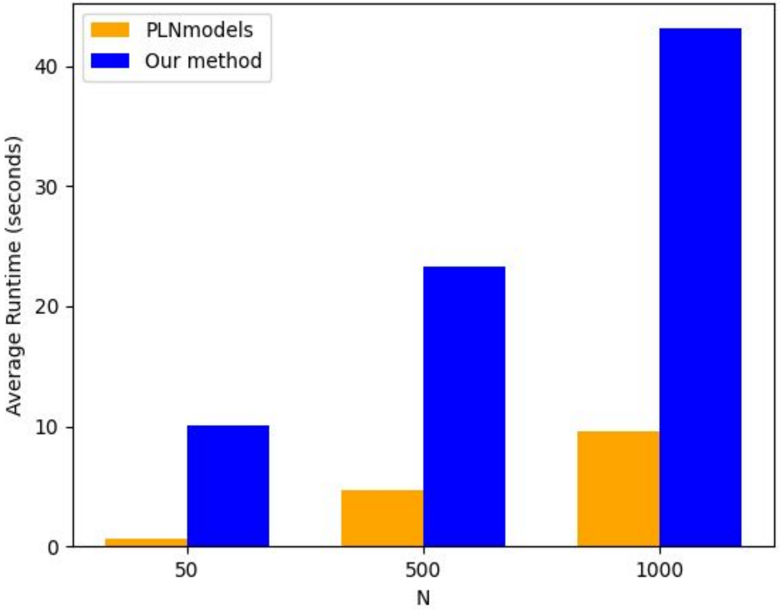
Correlation of Counts

Our Model

PLNmodels

Runtime Analysis

- Currently our software takes an average of 40 seconds (30 seconds more than PLNmodels) for inference with N=1000 samples.
- Profiling shows us this is due to repeated function calls to `torch.einsum` which are not optimized for *some* computations in PyTorch.
- PLNmodels is written in R/C++, our software is in Python and PyTorch



Conclusion

- Surpasses PLNmodels with unparalleled accuracy when tested on simulated datasets at various parameter settings ranging from easy to hard.
- Helps us uncover subtle correlation structures not modelled accurately by PLNmodels.
- We don't yet have a methodology for modelling covariates as factors which are present in real datasets.
- Computational time is more than PLNmodels by the order of seconds/few minutes which is optimizable according to our time complexity.

IISER KOLKATA

Future Directions

- Investigate on more robust initializations and extending them for other GLMM's
- Improve speed of the software
- Develop a method for modelling factors
- Developing a Poisson PCA for automatically choosing K and inferring sparse networks

Thank you!

Problems to be addressed

- Speed:
  - Used caching to store intermediate results for faster computations
  - Replaced einsum calls with matrix operations in torch
  - NOT significant speedup
  - Time complexity of our model: O(NDK^2)
  - Time complexity of Chiquet's model: O(NSDK), but their model is about 5 to 7 times faster in practice.
- Convergence:
  - Three main methods to overcome convergence issues:
    - Better initialization
    - Slower learning rate
    - Better suited algorithms
  - Even PLNmodels suffers from convergence issues. Check this issue here.
  - PLNmodels use two backends: torch and nlopt
  - The default behaviour of nlopt is CCSA while torch is RPROP, apparently CCSA is much more robust. Read more here.
  - For the Poisson Log normal model, to the best of my testing, the current initialization WORKS (finally! :)
- Extensions to other models like Bernoulli, Binomial, Gamma, Gumbel have the outline ready, but need to be carefully implemented, tested and improved.

Software Design: Extensions to other mGLMM's

- Functions that need to be implemented for each class:
  - `init_model_params():` Initialization of model parameters
  - `init_var_params():` Initialization of variational parameters (in case of neural networks, no need to implement this function)
  - `expCondLogProb():` This function computes this quantity below:
  -
  -

$$\underbrace{\mathbb{E}_{q_n}\left(\sum_{i=1}^{D} \ln p\left(y_{ni} \mid \mu_i^y = g^{-1}(z_{ni}), \boldsymbol{\theta}\right)\right)}_{\text{Expected Conditional Log Probability: } \rho(\boldsymbol{\theta},\boldsymbol{\phi})}$$

  - Gauss-Hermite Quadratur
  - `computeEtaLambda():` Using NN or the variational params, this function computes these quantities:
- Next steps:
  - Math stuff: Study possible robust initializ $\eta_i(y_i, \phi)$ r $\lambda_i(y_i, \phi)$ hese models.
  - Coding stuff:
    - Discuss and possibly modify the design and the code to be more efficient and roll out a package.
    - Trying out other network architectures (layer normalization)
  - Stat stuff: Derive and Implement
    - Linear Discriminant Analysis
    - Model Based Clustering
    - Network Inference