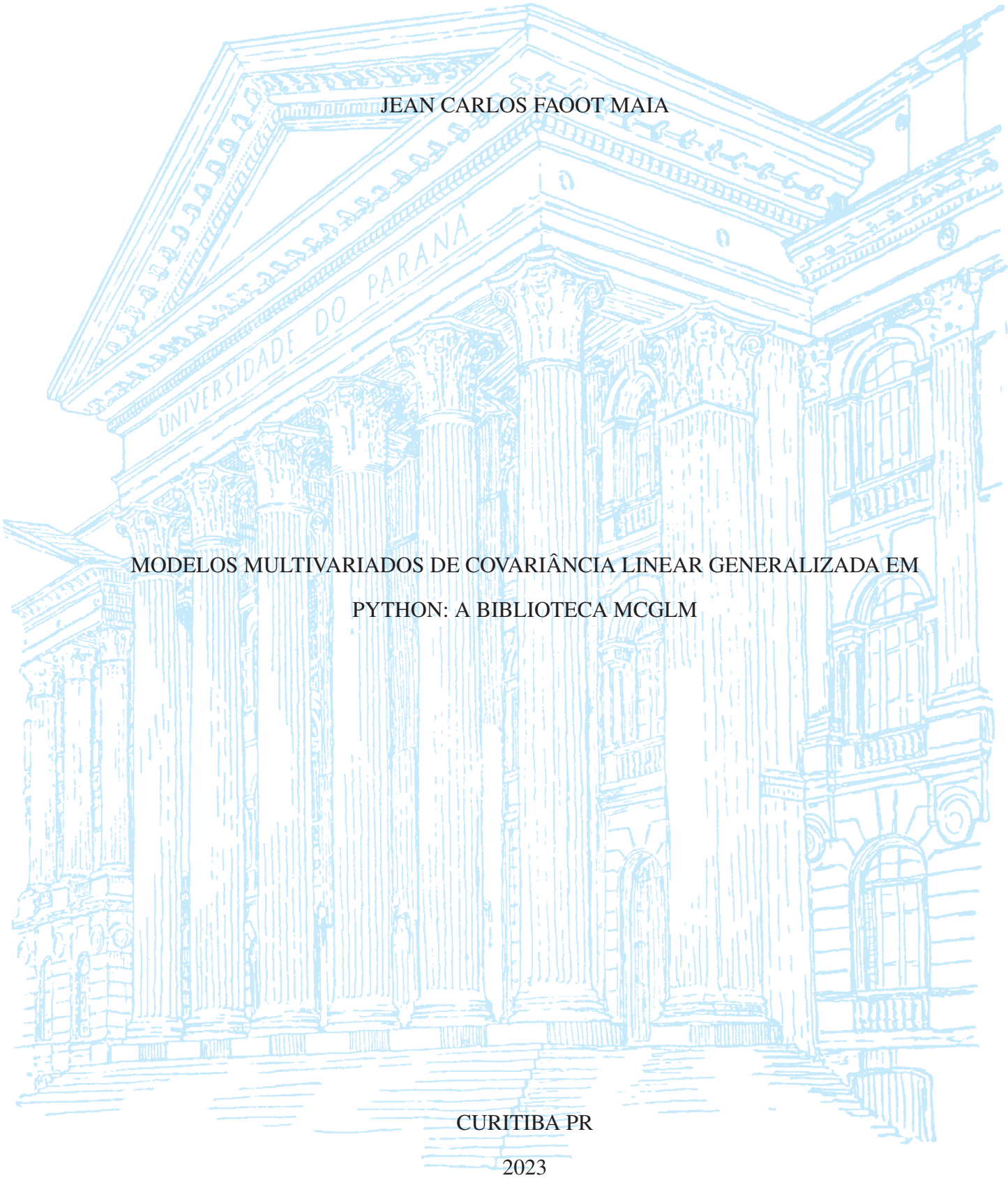
UNIVERSIDADE FEDERAL DO PARANÁ

JEAN CARLOS FAOOT MAIA



MODELOS MULTIVARIADOS DE COVARIÂNCIA LINEAR GENERALIZADA EM PYTHON: A BIBLIOTECA MCGLM

CURITIBA PR

2023

JEAN CARLOS FAOOT MAIA

MODELOS MULTIVARIADOS DE COVARIÂNCIA LINEAR GENERALIZADA EM PYTHON: A BIBLIOTECA MCGLM

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Wagner Hugo Bonat.

CURITIBA PR

2023

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA CIÊNCIA E TECNOLOGIA

Maia, Jean Carlos Faoot.
Modelos multivariados de covariância linear generalizada em Python: a biblioteca MCGLM. / Jean Carlos Faoot Maia. – Curitiba, 2023.
1 recurso on-line : PDF.

Dissertação (Mestrado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientador: Prof. Dr. Wagner Hugo Bonat.

1. Informática. 2. Programação (Computadores). 3. Python (Linguagem de programação de computadores). 4. Modelos lineares (Estatística). I. Bonat, Wagner Hugo. II. Universidade Federal do Paraná. Programa de Pós-Graduação em Informática. III. Título.

Bibliotecário: Nilson Carlos Vieira Júnior CRB-9/1797

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **JEAN CARLOS FAOOT MAIA** intitulada: **MODELOS MULTIVARIADOS DE COVARIÂNCIA LINEAR GENERALIZADA EM PYTHON: A BIBLIOTECA MCGLM**, sob orientação do Prof. Dr. WAGNER HUGO BONAT, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 02 de Fevereiro de 2023.

Assinatura Eletrônica
06/02/2023 11:40:36.0
WAGNER HUGO BONAT
Presidente da Banca Examinadora

Assinatura Eletrônica
06/02/2023 18:45:28.0
ANDERSON LUIZ ARA SOUZA
Avaliador Externo (UNIVERSIDADE FEDERAL DO PARANA / ESTATÍSTICA)

Assinatura Eletrônica
08/02/2023 16:49:30.0
ANDRÉ RICARDO ABED GRÉGIO
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

AGRADECIMENTOS

Redigir esta página foi uma tarefa gratificante. Recordar-me das etapas e ciclos que galgamos para a completude desta dissertação colocam-me em um estado de pleno contentamento.

Sinto-me privilegiado com a oportunidade deste projeto. Por tanto tempo buscara aprender sobre modelagem estatística. Fico encantado ao recapitular todo o conteúdo que estudei para a produção desta dissertação; além do aprofundamento específico no MCGLM, a longa revisão bibliográfica dos grandes clássicos da modelagem estatística. É complexo mensurar o impacto dessa etapa no meu particular; percebi-me como um amante da ciência, da estatística, do método científico e da vida.

"O método científico é inerentemente colaborativo". Eu não poderia ter sido assistido por um mestre melhor. Como disse em inúmeras oportunidades, considero Wagner Bonat um grande intelectual da área. Terei eterna gratidão pela sua atenção, paciência e parceria ao longo desta dissertação. Várias barreiras conceituais, que pareciam intransponíveis, foram superadas devido à sua incrivel capacidade de elucidação. Sinto-me privilegiado por ter ganhado um mestre e, sobretudo, um amigo.

Agradeço à minha banca, composta por Anderson Ara e André Grégio. Mestres que compuseram a produção desta dissertação com prestatividade e fraternidade. Agradeço aos meus amigos e gestores do Datalab Serasa Experian, que me ofereceram opoio e fomento incondicionais durante o processo. Outros camaradas de profissão que foram fundamentais durante essa etapa: Quenio, Larissa e Ubirajara. Obrigado!

À minha familia; meu pai, meu irmão, tios e avós. O meu espaço de interação nuclear. Assistir às suas felicidades diante das minhas conquistas é uma das formas mais legítimas de amor que experiencio. À minha esposa e companheira, Bia. Fonte de inspiração, amor e sentido. Tu fostes o amparo fundamental para que esta dissertação fosse produzida, amo-te.

Por fim, dedico esta dissertação à minha mãe, Matilde. Maior exemplo de luta, força e amor que presenciei. Imaginar a felicidade dela mediante o meu novo título acadêmico é facil, pois ela nunca se sentiu compelida ao revelar os seus honestos sentimentos. Tamanha força e individualidade deixaram marcas intensas, que mantém a nossa conexão viva, apesar da impossibilidade física. Onde estiver, sei que você está feliz, mãe.

À ciência brasileira, à universidade brasileira. Meus mais sinceros agradecimentos.

Jean Carlos Faoot Maia.

RESUMO

O advento da computação é fundamental ao desenvolvimento da Estatística; a Computação viabilizou modelos estatísticos, impulsionou a pesquisa e permitiu a distribuição de bibliotecas em larga-escala. Não obstante, a ciência Estatística transmutou a Computação, transformando-na em um elemento intransponível da sociedade moderna. Esta dissertação é mais um evento dessa amálgama próspera. Modelos Multivariados de Covariância Linear Generalizada é uma família de modelos estatísticos, postulada pelos pesquisadores Wagner Bonat e Bent Jørgensen, que universaliza o Modelo Linear Generalizado (GLM) para a análise multivariada de dados dependentes. Esta dissertação se propôs a implementar e publicar esse modelo estatístico para a linguagem `Python` em código aberto, como uma biblioteca. Como resultado, o código fonte está disponivel em: `https://github.com/jeancmaia/mcglm` e a biblioteca pode ser visualizada no PyPI, o principal repositório de software da linguagem `Python`, através do link: `https://pypi.org/project/mcglm/`. A biblioteca `Python` `mcglm` foi implementada nas normas da programação orientada a objetos, amparada nos princípios do acronimo *SOLID*, de Robert C. Martin. O desenvolvimento do código foi conduzido por testes unitários, perfazendo em uma cobertura de mais de noventa e um por cento. Visando uma fácil adesão à bibilioteca, as interfaces de treinamento e análise replicam o padrão da biblioteca `statsmodels`, a principal ferramenta para análises estatísticas na linguagem `Python`. Como objeto desta dissertação, os usuários da linguagem `Python` tornam-se aptos a acessar a um modelo estatístico inovador, oportunamente testado e plenamente extensível. A biblioteca `mcglm` amplia a lista de algoritmos estatísticos existentes na linguagem `Python` e ajuda a promover o aprendizado estatístico para os usuários desta linguagem, uma das mais expressivas da década.

Palavras-chave: Modelo Estatístico. MCGLM. Python. Regressão.

ABSTRACT

The advent of computing is pivotal in the history of Statistics; Computing enabled statistical models, boosted research and enabled large-scale distribution of libraries. Nevertheless, Statistical science has transmuted Computing, transforming it into an insurmountable element of modern society. This dissertation is one more event in this thriving amalgamation. Multivariate Covariance Generalized Linear Models is a family of statistical models postulated in 2015 by researchers Wagner Bonat and Bent Jørgensen. This family of models proposes an extension to the Generalized Linear Model (GLM) for multiple dependent data. This dissertation proposed to implement and publish this statistical model for the Python language in an open way, as a library. As a result, the source code is available at: https://github.com/jeancmaia/mcglm and the library can be viewed on PyPI, the main Python language software repository, via the link: https://pypi.org /project/mcglm/. The Python mcglm library was implemented in the norms of Object Oriented programming, supported by the principles of the acronym SOLID, by Robert C. Martin. Code development was driven by unit testing, yielding over ninety-two percent coverage. Aiming at an easy accession to the library, the training and analysis interfaces replicate the standard of the statsmodels library, the main tool for statistical analysis in the Python language. As the object of this dissertation, Python language users become able to access an innovative statistical model, timely tested and fully extensible. The mcglm library expands the list of statistical algorithms existing in the Python language and helps to promote statistical learning for users of this language, one of the most expressive of the decade.

Keywords: Statistical Model. MCGLM. Python. Regression.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

GLM	Modelos Lineares Generalizados
GAM	Modelos Aditivos Generalizados
GEE	Equações de Estimação Generalizadas
MCGLM	Modelos Multivariados de Covariância Linear Generalizada

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Datado do início do século XIX e controverso quanto à real autoria, o método de estimação dos mínimos quadrados estabeleceu uma proposta de otimização (Stigler, 1981). De acordo com o teorema de Gauss-Markov (Hallin, 2014), a estimação resultante dos mínimos quadrados é ótima e não-enviesada sob condições lineares de valores contínuos. Esse método de otimização é aplicado pelo modelo estatístico regressão linear (Galton, 1886), que viabiliza a análise explicativa entre variáveis independentes e uma variável dependente. Os pressupostos desse modelo são a inerente linearidade; a indepêndencia dos componentes da variável resposta; erro gaussiano com média 0 e variância constante.

As três premissas da regressão linear foram abordadas por inúmeras propostas estatísticas ao longo dos anos, visando à sua expansão; encontra-se, por fim, dados com peculiaridades que suplantam essas restrições, como dados de contagem ou limitados. O pressuposto gaussiano foi pautado no século XX, para a análise em dados limitados, reais positivos assimétricos e contagem (Postelnicu, 2011; Kaufmann e Schering, 2014; Kateri, 2014). Os modelos lineares generalizados (GLM) propunham uma solução unificada de expansão à regressão linear, permitindo a análise de variáveis resposta pertencentes à família exponencial (Nelder e Wedderburn, 1972; Müller, 2004). O GLM é uma solução robusta e amplamente utilizada na análise estatística e em soluções preditivas.

O pressuposto de dependência foi abordado por outras postulações estatísticas. Casos emblemáticos da aplicação desses modelos, como ajuste em séries temporais e dados espaciais, são desafios populares da estatística. Expansões notórias são as Equações de Estimação Generalizadas (GEE) (Liang et al., 1992), Modelos Generalizados Aditivos (GAM) (Hastie e Tibshirani, 1986), Modelos Mistos (Verbeke et al., 2014), Modelos Mistos Generalizados e Copulas (Krupskii e Joe, 2013; Masarotto e Varin, 2012). Do meio das mais recentes postulações, reside o modelo Multivariado de Covariância Linear Generalizada (MCGLM), objeto desta dissertação.

A família de modelos estatísticos Multivariados de Covariância Linear Generalizada (MCGLM) (Bonat e Jørgensen, 2016a) se caracteriza pela sua notável versatilidade. O MCGLM pode ajustar dados gaussianos ou não-gaussianos, univariados ou multivariados, independentes ou não-independentes. Esse traço é oriundo da estimação baseada em pressupostos de dois momentos e da descomplicada especificação dos cinco componentes fundamentais do modelo: preditor linear via matriz de delineamento, função de ligação, função de variância, função ligação de covariância e o preditor linear matricial.

Como uma proposta de ferramenta para análise estatística unificada, o MCGLM permite a aferição dos coeficientes de regressão e testes de hipótese, medidas de qualidade de ajuste, métricas para comparação de modelos, análise dos parâmetros de dispersão com intervalos de confiança e coeficientes de correlação entre respostas para a análise multivariada. Estabelecendo-se portanto, como uma proposta de solução unificada para análise de dados.

É fundamental pontuar a computação e o seu papel fundamental na viabilidade e disseminação dos modelos estatísticos postulados. As análises são suportadas integralmente por operações computacionais; inicia-se com o manuseio dos dados, geralmente registrados em arquivos de dados, e a análise estatística é conduzida com um pacote ou biblioteca de códigos, que faz a ingestão de software pré-desenvolvido na plataforma do usuário. Dessa forma, os pesquisadores e profissionais das áreas Estatística e Ciência da Dados acessam prontamente

aos modelos estatísticos disponíveis. Linguagens distintas possuem termos únicos para essas bases de código adicionáveis; A linguagem `R` chamamos de pacotes, e bibliotecas à `Python`. Parte significativa dessas bases de código foi desenvolvida de modo aberto, permitindo o desenvolvimento colaborativo, livre e contínuo.

A linguagem `R` é consenso entre os estatísticos e pesquisadores, sendo a principal escolha para análise estatística. Não obstante, a linguagem `Python`, consolidada na aplicação de modelos de aprendizado de máquina com foco na mineração de dados (Agha e Haider, 2014), tem ganhado notoriedade nessa tarefa; a biblioteca `statsmodels` (Seabold e Perktold, 2010) é um dos expoentes no fomento da aplicação de modelos estatísticos em `Python`. Nesta dissertação, buscamos cooperar para o avanço do aprendizado estatístico na linguagem `Python` com a primeira biblioteca de código aberto da família de modelos estatísticos MCGLM.

1.2 OBJETIVOS

Esta dissertação implementa a primeira biblioteca de código aberto do MCGLM para a linguagem `Python`, chamada `mcglm`. Ela aprovisiona um ambiente completo para a análise estatística com o suporte do modelo MCGLM, permitindo a aferiação dos parametros de regressão, de dispersão, intervalos de confiança, teste de hipótese e a análise resídual. Por fim, a biblioteca está hospedada no repositório `PyPI` e pode ser instalada com o suporte de um gerenciador de bibliotecas, como `pip` ou `conda`.

Como um dos pilares do projeto, aplicamos a programação orientada a objetos no desenvolvimento da biblioteca, o que trouxe legibilidade e escalabilidade ao código base. Ademais, como um objetivo ancilar, a biblioteca oferece uma interface de treinamento acessível e familiar, herdando o padrão da `statsmodels` e implementando métodos auxiliares para a especificação do modelo.

O MCGLM está disponível aos usuários da linguagem `R` pelo pacote `mcglm` (Bonat, 2016), também de código aberto. Essa versão `R` do modelo guiou o desenvolvimento deste projeto, especialmente para a validação dos cálculos executados pela biblioteca.

Os traços fundamentais do MCGLM o colocam em uma posição promissora para a análise estatística; um modelo versátil, unificado e hábil. O uso apropriado da biblioteca é demonstrado pelos exemplos nesta dissertação, juntamente à sua polivalência, através da unicidade dos casos.

1.3 CONTRIBUIÇÕES

Essa nova biblioteca da linguagem `Python` aprovisiona a primeira interface para análise estatística com o suporte do modelo MCGLM, no padrão do pacote `statsmodels`. Nossa contribuição viabiliza um novo caminho de análise na linguagem `Python`, criando um ambiente alternativo aos profissionais da estatística, familiarizados com a linguagem `R`.

Esta dissertação visa promover à linguagem `Python` o aprendizado estatístico através da implementação de uma família de modelos vanguardista, além da aplicação adequada do paradigma da orientação a objetos no desenvolvimento de modelos estatísticos. Dada a versatilidade do modelo, esta dissertação pode servir como um guia disseminador e unificado da análise estatística em `Python`.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

Nos capítulos seguintes, esta dissertação discorre pelos modelos estatísticos existentes na linguagem `Python`, o modelo MCGLM e seu desenvolvimento e a aplicação do modelo em dois exemplos. No segundo capítulo, uma revisão dos modelos estatísticos em `Python`. No terceiro capítulo, a apresentação completa do modelo MCGLM. No quarto capítulo, a apresentação da interface da nova biblioteca `mcglm` da linguagem Python. No quinto capítulo, a arquitetura de software aplicada ao desenvolvimento. No sexto capítulo, a apresentação de uma simulação para a avaliação das estimativas. No sétimo capítulo, a apresentação de duas análises estatísticas realizadas com a biblioteca `mcglm`. O último capítulo é dedicado a um panorama da dissertação.

2 MODELOS ESTATÍSTICOS E SOFTWARE

O advento computacional é fundamental ao desenvolvimento estatístico. Dentre muitos motivos, elenca-se dois: acesso rápido e confiável a modelos estatísticos publicados via código aberto; a viabilidade de modelos, resultantes da alta capacidade matemática concebida pelo processamento computacional.

O impacto do software de código aberto é indubitável na economia e na cultura. Estima-se uma contribuição atual ao Produto Interno Bruto europeu de 60 a 95 bilhões (Wachs et al., 2022). Devido ao inerente ambiente colaborativo e de fácil validação, os projetos têm ganhado notoriedade pela confiabilidade.

A principal plataforma global de publicação de código aberto é o `GitHub`, que no ano de 2020 reportou mais de 60 milhões de novos repositórios (Trujillo et al., 2022). Devido às suas funcionalidades, o `GitHub` é um ambiente de fácil adesão, colaboração e compartilhamento de conhecimento. Outras ferramentas para hospedagem de código são `GitLab` e `Bitbucket`.

Muitos dos modelos estatísticos estão acessíveis via código aberto. Cabe ao usuário a escolha da linguagem de programação e avaliação das biliotecas, ou pacotes, existentes. Atualmente, a linguagem `R` é consenso entre os pesquisadores e acadêmicos da estatística, entretanto a linguagem `Python` é uma opção viável.

2.1 A COMPUTAÇÃO VIABILIZA O APRENDIZADO ESTATÍSTICO

Denomina-se estimador pontual o método aferidor das estimativas otimizadas de um modelo estatístico. Esse cálculo pode ser exaustivo, inclusive para os modelos mais incipientes. O consolidado método dos mínimos quadrados (Stigler, 1981), que possui uma solução fechada, pode se tornar inviável sem o suporte de operações computacionais. Segue uma notação matemática com a solução analítica desse algoritmo.

Seja X uma matriz de covariáveis; seja y o vetor com a variável resposta; β o vetor de parâmetros de regressão. A solução fechada do método dos mínimos quadrados pode ser descrita:

$$\hat{\beta} = (X^T X)^{-1} X^T y.$$

Em sua formulação inicial, os modelos lineares generalizados aplicam o algoritmo mínimos quadrados generalizados, ou minimos quadrados reponderados iterativo. Esse algoritmo necessita das derivadas de primeira e segunda ordem, em um processo iterativo, para o ajuste completo (De Luca et al., 2017). Sem o auxílio computacional, é relativamente inviável.

Em 1919, R.A.Fisher publicou um de seus trabalhos mais icônicos. Com o suporte dos Modelos Mistos, ele analisou a correlação entre relativos na suposição de herança de Mendelian (Fisher, 1919). Apesar desse caso emblemático, os modelos mistos tornaram-se plenamente viáveis apenas com o advento da computação e incluindo estimação sob as perspectivas frequentistas (Breslow e Clayton, 1993) e bayesianas (Fahrmeir e Lang, 2001).

Fica evidente que os softwares estatísticos são fundamentais na consolidação e disseminação do aprendizado estatístico.

2.2 MODELOS ESTATÍSTICOS E LINGUAGENS DE PROGRAMAÇÃO

A linguagem de programação R é consenso no estudo estatístico acadêmico (Ihaka e Gentleman, 1996) e proporciona um ambiente em expansão, caracterizado pelo inerente ciclo de criação de pacotes de código aberto e disponibilização no CRAN (Comprehensive R Archive Network) (Bommarito e Bommarito, 2021). A linguagem propicia aos usuários um ambiente vivaz, cooperante e expansível.

Pacotes de código aberto, compilados e publicados no CRAN, podem ser rapidamente instalados em ambientes locais ou remotos para a utilização. A linguagem R disponibiliza o comando `install.packages` para a adesão de bibliotecas hospedadas no repósitorio. A política de hospedagem de pacotes está bem descrita na documentação do CRAN `https://cran.r-project.org/`.

A história dos modelos estatísticos na linguagem Python é recente e se confunde com o lançamento do projeto `statsmodels` (Seabold e Perktold, 2010). Python é a escolha mais popular para o desenvolvimento de modelos escaláveis de aprendizado de máquina e mineração de dados, tanto clássico quanto profundo. As bibliotecas de código aberto permitiram a rápida difusão da linguagem no desenvolvimento de Ciência de Dados; destacam-se: `scikit-learn` (Pedregosa et al., 2011), `tensorflow` (Abadi et al., 2015), `pytorch` (Paszke et al., 2019). Essas bibliotecas diferem da `statsmodels` em muitos aspectos, sobretudo na relação com as estimativas. Se as bibliotecas de Ciência de Dados possuem a atribuição irrestrita da busca dos parâmetros otimizados, a biblioteca estatística pormenoriza essas estimativas via intervalos de confiança e testes de hipótese.

PyPI é o principal repositório de software para a linguagem Python, contando com mais de 350 mil projetos e mais de 3 milhões de publicações. Pacotes de código aberto compilados e publicados no PyPI podem ser rapidamente instalados em ambientes locais ou remotos para a utilização. Python possui algumas opções para gestão de bibliotecas, como: `pip` e `conda`. No site da plataforma, é possível encontrar a documentação definitiva `https://pypi.org/`.

2.3 BIBLIOTECAS ESTATÍSTICAS EM PYTHON

A biblioteca `statsmodels` oferece ferramentas para análise estatística em uma interface acessível e consolidada. A lista de modelos estatísticos implementados é extensiva e notória pelo ajuste eficiente sob postulações frequentistas. As bibliotecas `numpy` (Harris, 2020) e `scipy` (Virtanen, 2020) são fundamentais para o desenvolvimento da `statsmodels`. A primeira oferece métodos para manipulação e operação em vetores e matrizes. A última implementa distribuições de probabilidade, testes estatísticos, métodos de otimização, funções de correlação e outras operações. Ambas bibliotecas distribuem suas operações em chamadas à linguagem C (Behnel et al., 2011).

Programação Probabilística é um tema em ascensão na linguagem Python. É possível a especificação de modelos como problemas de inferência bayesianos (Meent et al., 2018). Destaca-se as bibliotecas: PyMC (Salvatier et al., 2016), `pyro` (Bingham et al., 2018), PyStan (Carpenter et al., 2017), and `yaps` (Baudart et al., 2018). Muitos modelos estatísticos prescritos sob preceitos clássicos podem ser rescritos por postulações bayesianas, mas diferem em muitas propriedades estatísticas.

2.3.1 Biblioteca Statsmodels

A biblioteca, lançada em 2010, implementa métodos para análises estatísticas e econometria em Python, de modo aberto. À época, o projeto almejava a adequação dos usuários

das linguagens R, Stata, SAS, SPSS, NLOGIT, GAUSS ou MATLAB e a viabilidade de um novo ambiente para pesquisa e desenvolvimento na área de econometria (Seabold e Perktold, 2010). Desde a criação foram mais de dezenas de versões da biblioteca, apontando à estabilidade e consolidação da ferramenta.

A página da biblioteca possui uma documentação compreensiva para incutir na interface da biblioteca e na usabilidade geral dos *endpoints* - as classes que implementam os modelos da biblioteca, tais como OLS para regressão linear e GLM para os modelo linear generalizado. Cada modelo estatístico possui o seu próprio *endpoint* e o ajuste envolve três passos: Instanciação de um objeto, a partir da classe que implementa o modelo desejado; a chamada do método `fit()`; aferição das estimativas através do método `summary()` ou dos atributos do objeto criado. O site da bilioteca é `https://www.statsmodels.org/stable/index.html`

A `statsmodels` aprovisiona interfaces para modelos estatísticos, analise de séries temporais, analise de sobrevivência, entre outros. No repertório de modelos estatísticos, encontra-se: Regressão Linear, ANOVA, Modelo Linear Generalizado, Equações de Estimação Generalizadas (Liang et al., 1992), Modelos Generalizados Aditivos (Hastie e Tibshirani, 1986), Modelos Mistos (Verbeke et al., 2014), Modelos Mistos Generalizados e Copulas (Krupskii e Joe, 2013; Masarotto e Varin, 2012). Esses modelos implementam ajustes baseados no método de máxima verossimilhança, respaldado no paradigma frequentista.

2.3.2 Bibliotecas PyMC, Pyro, PyStan e YAPS

A biblioteca `PyMC` oferece um ambiente completo para a implementação de modelos estatísticos bayesianos através de Programação Probabilística (Meent et al., 2018). Sob a perspectiva bayesiana, muitos modelos estatísticos são viáveis, como Regressão Linear, Modelo Linear Generalizado, Modelos Mistos e Copulas.

A biblioteca `Pyro` é uma plataforma de desenvolvimento de Programação Probabilística escalável (Bingham et al., 2018). `Pyro` oferece os melhores algoritmos para ajuste de modelos, inclusive a inferência variacional.

As bibliotecas `PyStan` e `YAPS` permitem a implementação de Programação Probabilística em `Python` via interface ao `Stan` (Carpenter et al., 2017), paradigma popular em outras linguagens como em R. Para além do preceito bayesiano, no stan também é possível especificar o modelo estatístico nos modelos clássicos.

2.4 ESCOPO DE COMPARAÇÃO

Há várias ótimas opções para o desenvolvimento probabilístico na linguagem `Python`. A estimação bayesiana, oferecida pelas bibliotecas `PyMC`, `Pyro`, `PyStan` divergem do paradigma frequentista adotado pela biblioteca `statsmodels` e agora, a `mcglm`. Estimação bayesiana e a programação probabilistica não residem no escopo comparativo desta dissertação.

3 MODELOS MULTIVARIADOS DE COVARIÂNCIA LINEAR GENERALIZADA

O Modelo Multivariado de Covariância Linear Generalizada (MCGLM) é uma proposta extensiva ao GLM que permite o ajuste multivariado em dados não independentes. A seguir, desenvolvemos o enredo histórico fundacional do MCGLM.

3.1 INTRODUÇÃO

A regressão linear é um dos primeiros modelos estatísticos postulados (Galton, 1886; Stigler, 1981). O modelo também é abordado na literatura clássica de aprendizado de máquina devido à sua resiliência para predição (Ardeshir et al., 2021; Alkemade et al., 2022). Esse modelo busca analisar uma variável resposta a partir de uma ou mais variáveis explicativas, cuja notação matemática é referenciada como uma matriz de covariáveis X e um vetor resposta y, respectivamente. O vetor de parâmetros β resultante associa multiplicadores às covariáveis na operação linear. É possível medir a relevância estatística das covariáveis via teste de hipótese, onde o mais popular é o teste de *Wald* (Wald, 1943; de Freitas e Bonat, 2022).

Para o diagnóstico de ajuste do modelo, sua indexação e comparação entre modelos, utiliza-se as métricas R^2, R^2 ajustado e a análise residual (Li et al., 2020). A regressão linear possui três pressupostos: a linearidade; a indepêndencia dos componentes de y; erro gaussiano com média 0 e homocedasticidade.

Nelder e Wedderburn (1972) publicaram uma proposta extensiva unificada à regressão linear, os modelos lineares generalizados (GLM). Os modelos lineares generalizados suplantam uma das restrições no modelo linear, a gaussianidade. Como muitos problemas na natureza não podem ser representados pela distribuição gaussiana, os GLMs ganharam ampla adesão pela versatilidade. Nesse modelo estatístico, a variável resposta é modelada como uma distribuição de probabilidade pertencente à família exponencial. A lista de distribuição de probabilidade possíveis são: *Poisson*, para dados de contagem; *Gamma*, para dados reais positivos e assimétricos; *Binomial*, para dados limitados, entre outros.

Além das covariáveis e da variável resposta, os usuários do modelo linear generalizado devem especificar as funções de ligação e variância, ou um modelo de probabilidade pertencente à familia exponencial. Elas são responsáveis pela transformação ao suporte desejado e a especificação do estimador de máxima verossimilhança (Miura, 2011). Na literatura clássica de aprendizado de máquinas, um dos GLMs mais populares aplica a função de ligação *logit*, alcunhado como regressão logística.

O diagnóstico do ajuste é viável pela análise residual ou envelope de modelos (Moral et al., 2017). A comparação entre modelos é possível pelas métricas *akaike information criterion* e *bayesian information criterion*, entre outras (Bhattacharya e Burman, 2016). Os pressupostos dos modelos lineares generalizados: independência dos componentes da resposta; linearidade sob a função de ligação; resposta modelada por uma distribuição de probabilidade pertencente à família exponencial.

Como mencionado, o MCGLM universaliza o GLM. Em uma postulação heterodoxa, a de dois momentos, este modelo é especificado em um fórmula inicial ao MCGLM. Seja Y um

vetor resposta N x 1, X a matriz de delineamento N x k e β um vetor de regressão k x 1. A especificação de dois momentos do GLM:

$$\begin{aligned} \mathrm{E}(\mathbf{Y}) &= \boldsymbol{\mu} = g^{-1}(\boldsymbol{X}\boldsymbol{\beta}), \\ \mathrm{Var}(\mathbf{Y}) &= \boldsymbol{\Sigma} = \mathrm{V}(\boldsymbol{\mu}; p)^{\frac{1}{2}}(\tau_0 \boldsymbol{I})\mathrm{V}(\boldsymbol{\mu}; p)^{\frac{1}{2}}, \end{aligned}$$

onde $g(\cdot)$ é a função de ligação, $V(\mu; p) = diag(\vartheta(\mu; p))$, é uma matrix diagonal cujas as entradas são resultantes da aplicação da função de variância $\vartheta(; p)$ aplicada aos valores do vetor μ. A matrix I denota uma metriz identidade N x N, τ_0 e p são os parâmetros de dispersão e o *power*.

O modelo de equações de estimação generalizadas (GEE) (Liang et al., 1992) expande a família dos modelos lineares generalizados, permitindo o ajuste em dados com componentes longitudinais, com foco nos parâmetros de regressão. O GEE devota-se ao ajuste da distribuição marginal da resposta. A primeira expansão das GEEs em relação às GLMs é o uso da proposta de equação de estimação `Quasi-likelihood` (Wedderburn, 1974), que suplanta a necessidade de especificação de uma distribuição de probabilidade. A dependência dos componentes da variável resposta é especificada por uma matriz de correlação. Portanto, além da especificação das covariáveis, função ligação e função de variância, o GEE exige uma matriz de correlação.

O GEE possibilita o ajuste de algumas estruturas de correlação bem estabelecidas: *Independence*, *M-dependence*, *Exchangeable* e *Auto Regressivo*. Essa opção define a matriz de correlação e a função de variância, simultaneamente. Fica a critério do usuário a especificação da estrutura mais adequada ao problema. O GEE implementa pressupostos de dois momentos para a inferência, combinando o algoritmo *Fisher Scoring* (Jennrich, 1969; Widyaningsih et al., 2017) para os parâmetros de regressão e estimação de momentos para os parâmetros de dispersão.

A postulação do MCGLM propõe uma nova família de modelos estatísticos, combinando e expandindo soluções propostas pelos modelos GLM e GEE.

3.2 O MODELO

O MCGLM (Bonat e Jørgensen, 2016a) universaliza o modelo linear generalizado, permitindo ajuste multivariado e respostas não independentes, com foco nos parâmetros de regressão e dispersão. O modelo pode ser ajustado em vários tipos de dependência, como: longitudinal, espacial e espaço-temporal. O modelo estatístico é baseado em suposições de dois momentos e define mecanismos distintos para média e variância. O processo de estimação aplica algoritmos de otimização e equações de estimação para cada momento.

O processo integrado de otimização do algoritmo combina os ajustes da média e da variância; primeiramente, para o ajuste da média, o MCGLM implementa a função `Quasi-score` (Wedderburn, 1974) e o método de otimização *Fisher Scoring* (Jennrich, 1969; Widyaningsih et al., 2017); para a variância, o MCGLM implementa a função de estimação de *Pearson* e o algoritmo de otimização *Chaser* (Jørgensen e Knudsen, 2004). Os dois ajustes compõe o ciclo unitário da otimização, que é completado em seu ponto otimizado.

O MCGLM é ajustado via uma especificação de cinco componentes: uma matriz de delineamento para o preditor linear, função de ligação, função de variância, preditor linear matricial e uma função de ligação de covariância. Cada resposta deve possuir seu grupo de parâmetros.

3.3 ESPECIFICAÇÃO DO MODELO

Seja $\mathbf{Y}_{N\times R} = \{\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_R\}$ a matriz com as variáveis resposta e $\mathbf{M}_{N\times R} = \{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_R\}$ a correspondente matriz de valores esperados. Seja $\boldsymbol{\Sigma}_r$ a matriz de variância e covariância da resposta r, com r=[1,..,R]. Similarmente, seja $\boldsymbol{\Sigma}_b$ a matriz de correlação entre respostas. A matriz $\boldsymbol{X}_r$ denota as covariáveis da variável resposta r e $\boldsymbol{\beta}_r$ o vetor com os parâmetros de regressão. A especificação de dois momentos do MCGLM (Bonat e Jørgensen, 2016b) é definida por:

$$\begin{aligned} \mathrm{E}(\mathbf{Y}) &= \mathbf{M} = \{g_1^{-1}(\boldsymbol{X}_1\boldsymbol{\beta}_1), \ldots, g_R^{-1}(\boldsymbol{X}_R\boldsymbol{\beta}_R)\}, \\ \mathrm{Var}(\mathbf{Y}) &= \boldsymbol{C} = \boldsymbol{\Sigma}_R \overset{G}{\otimes} \boldsymbol{\Sigma}_b, \end{aligned}$$

o componente C é calculado por vias do produto `Kronecker` generalizado $\boldsymbol{\Sigma}_R \overset{G}{\otimes} \boldsymbol{\Sigma}_b = \mathrm{Bdiag}(\tilde{\boldsymbol{\Sigma}}_1, \ldots, \tilde{\boldsymbol{\Sigma}}_R)(\boldsymbol{\Sigma}_b \otimes \boldsymbol{I})\mathrm{Bdiag}(\tilde{\boldsymbol{\Sigma}}_1^T, \ldots, \tilde{\boldsymbol{\Sigma}}_R^T)$ (Martinez-Beneito, 2013). A matriz $\tilde{\boldsymbol{\Sigma}}_r$ denota uma matriz triangular inferior resultante da decomposição Cholesky de $\boldsymbol{\Sigma}_r$. O operador Bdiag pormenoriza uma matriz bloco diagonal e $\boldsymbol{I}$ denota uma a matrix identidade $N \times N$. Para o vetor de valores esperados, a operação $g_r(\cdot)$ é uma função de ligação usual dos modelos lineares generalizados. A matrix $\boldsymbol{\Sigma}_r$ é definido por:

$$\boldsymbol{\Sigma}_r = \mathrm{V}(\boldsymbol{\mu}_r; p_r)^{\frac{1}{2}}(\boldsymbol{\Omega}(\boldsymbol{\tau}_r))\mathrm{V}(\boldsymbol{\mu}_r; p_r)^{\frac{1}{2}},$$

onde $\mathrm{V}(\boldsymbol{\mu}_r; p_r) = \mathrm{diag}(\vartheta(\boldsymbol{\mu}_r; p_r))$ é uma matrix diagonal, cujas entradas são definidas por meio da função de variância ϑ aplicada ao vetor $\boldsymbol{\mu}$. p_r é o parâmetro de potência e $\boldsymbol{\tau}_r$ o parâmetro de dispersão.

Finalmente, a matriz $\boldsymbol{\Omega}(\boldsymbol{\tau}_r)$, chamado de preditor linear matricial (Anderson, 1973; Pourahmadi, 2000; Bonat e Jørgensen, 2016a), descreve a covariância.

$$h(\boldsymbol{\Omega}(\boldsymbol{\tau}_r)) = \tau_{r0}Z_{r0} + \cdots + \tau_{rD}Z_{rD}, \tag{3.1}$$

onde $\mathrm{h_r}(\cdot)$ é a função de ligação de covariância. As matrizes Z especificam a dependência; o caso independente é descrito por uma matriz diagonal. Como descrito em Bonat e Jørgensen (2016a), vários modelos para dados dependentes podem ser especificados pelas matrizes Z, como a regressão linear; modelo linear generalizado; modelos mistos; modelos médias móveis; modelos autoregressivos condicionados; entre outros.

3.4 ESTIMAÇÃO E INFERÊNCIA

O modelo MCGLM é ajustado via equações de estimação, usando suposições de primeiro e segundo momentos (Jørgensen e Knudsen, 2004). Esses pressupostos derivam dois grupos de parâmetros $\boldsymbol{\theta} = (\boldsymbol{\beta}^\top, \boldsymbol{\lambda}^\top)^\top$. Nessa notação, $\boldsymbol{\beta} = (\boldsymbol{\beta}_1^\top, \ldots, \boldsymbol{\beta}_R^\top)^\top$ e $\boldsymbol{\lambda} = (\rho_1, \ldots, \rho_{R(R-1)/2}, p_1, \ldots, p_R, \boldsymbol{\tau}_1^\top, \ldots, \boldsymbol{\tau}_R^\top)^\top$ denotam $K \times 1$ e $Q \times 1$ os vetores com os parametros de regressão e dispersão, respectivamente.

Seja $\mathcal{Y} = (\boldsymbol{Y}_1^\top, \ldots, \boldsymbol{Y}_R^\top)^\top$ e $\mathcal{M} = (\boldsymbol{\mu}_1^\top, \ldots, \boldsymbol{\mu}_R^\top)^\top$ os vetores empilhados da matriz resposta $\mathbf{Y}_{N\times R}$ e a matriz de valores esperados $\mathbf{M}_{N\times R}$, respectivamente.

A função `Quasi-score` (Liang e Zeger, 1986) adaptada para os parâmetros de regressão

$$\psi_{\boldsymbol{\beta}}(\boldsymbol{\beta}, \boldsymbol{\lambda}) = \boldsymbol{D}^\top\boldsymbol{C}^{-1}(\mathcal{Y} - \mathcal{M}),$$

onde $\boldsymbol{D} = \nabla_{\boldsymbol{\beta}}\boldsymbol{\mathcal{M}}$ é uma matriz $NR \times K$, e $\nabla_{\boldsymbol{\beta}}$ denota o operador gradiente. As matrizes *sensitivity* e *variability* $K \times K$ de $\psi_{\boldsymbol{\beta}}$ são respectivamente descritas:

$$\mathrm{S}_{\boldsymbol{\beta}} = \mathrm{E}(\nabla_{\boldsymbol{\beta}}\psi_{\boldsymbol{\beta}}) = -\boldsymbol{D}^{\top}\boldsymbol{C}^{-1}\boldsymbol{D} \quad \text{e} \quad \mathrm{V}_{\boldsymbol{\beta}} = \mathrm{Var}(\psi_{\boldsymbol{\beta}}) = \boldsymbol{D}^{\top}\boldsymbol{C}^{-1}\boldsymbol{D}.$$

Ao segundo momento, a função de estimação de *Pearson*, definida pelos componentes:

$$\psi_{\lambda_i}(\boldsymbol{\beta}, \boldsymbol{\lambda}) = \mathrm{tr}(W_{\lambda_i}(\boldsymbol{r}^{\top}\boldsymbol{r} - \boldsymbol{C})) \quad \text{, para} \quad i = 1, \ldots, Q,$$

onde $W_{\lambda_i} = -\partial \boldsymbol{C}^{-1}/\partial \boldsymbol{\lambda}_i$ e $\boldsymbol{r} = \boldsymbol{\mathcal{Y}} - \boldsymbol{\mathcal{M}}$ foi adaptado aos parâmetros de dispersão.

A entrada (i, j) da $Q \times Q$ matriz sensitividade da $\psi_{\boldsymbol{\lambda}}$ é dado por:

$$\mathrm{S}_{\lambda_{ij}} = \mathrm{E}\left(\frac{\partial}{\partial \boldsymbol{\lambda}_i}\psi_{\lambda_j}\right) = -\mathrm{tr}\left(W_{\lambda_i}\boldsymbol{C}W_{\lambda_j}\boldsymbol{C}\right).$$

A entrada (i, j) da $Q \times Q$ matriz variabilidade de $\psi_{\boldsymbol{\lambda}}$ é dado por:

$$\mathrm{V}_{\lambda_{ij}} = \mathrm{Cov}(\psi_{\lambda_i}, \psi_{\lambda_j}) = 2\mathrm{tr}(W_{\lambda_i}\boldsymbol{C}W_{\lambda_j}\boldsymbol{C}) + \sum_{l=1}^{NR} k_l^{(4)}(W_{\lambda_i})_{ll}(W_{\lambda_j})_{ll},$$

onde $k_l^{(4)}$ denota a quarta cumulante de $\boldsymbol{\mathcal{Y}}_l$.

Bonat e Jørgensen (2016b) propuseram o cálculo das matrizes de sensitividade e variabilidade cruzadas, que combinam as covariâncias de $\hat{\boldsymbol{\beta}}$ e $\hat{\boldsymbol{\lambda}}$. As matrizes conjuntas de sensitividade e variabilidade de $\psi_{\boldsymbol{\beta}}$ e $\psi_{\boldsymbol{\lambda}}$ são calculadas por:

$$\mathrm{S}_{\boldsymbol{\theta}} = \begin{pmatrix} \mathrm{S}_{\boldsymbol{\beta}} & \mathrm{S}_{\boldsymbol{\beta\lambda}} \\ \mathrm{S}_{\boldsymbol{\lambda\beta}} & \mathrm{S}_{\boldsymbol{\lambda}.} \end{pmatrix} \quad \text{e} \quad \mathrm{V}_{\boldsymbol{\theta}} = \begin{pmatrix} \mathrm{V}_{\boldsymbol{\beta}} & \mathrm{V}_{\boldsymbol{\lambda\beta}}^{\top} \\ \mathrm{V}_{\boldsymbol{\lambda\beta}} & \mathrm{V}_{\boldsymbol{\lambda}.} \end{pmatrix}$$

O modelo implementa o algoritmo *Chaser* modificado proposto para resolver o sistema de equações $\psi_{\boldsymbol{\beta}} = \mathbf{0}$ e $\psi_{\boldsymbol{\lambda}} = \mathbf{0}$ (Jørgensen e Knudsen, 2004), definido por:

$$\begin{aligned} \boldsymbol{\beta}^{(i+1)} &= \boldsymbol{\beta}^{(i)} - \mathrm{S}_{\boldsymbol{\beta}}^{-1}\psi_{\boldsymbol{\beta}}(\boldsymbol{\beta}^{(i)}, \boldsymbol{\lambda}^{(i)}) \\ \boldsymbol{\lambda}^{(i+1)} &= \boldsymbol{\lambda}^{(i)} - \alpha\mathrm{S}_{\boldsymbol{\lambda}}^{-1}\psi_{\boldsymbol{\lambda}}(\boldsymbol{\beta}^{(i+1)}, \boldsymbol{\lambda}^{(i)}). \end{aligned} \tag{3.2}$$

Um novo termo à equação do segundo momento, o *tuning* α, para auxiliar no ajuste do segundo momento. Esse termo é uma constante controladora para a atualização dos parâmetros $\boldsymbol{\lambda}$.

Seja $\hat{\boldsymbol{\theta}} = (\hat{\boldsymbol{\beta}}^{\top}, \hat{\boldsymbol{\lambda}}^{\top})^{\top}$ o estimador da equação de estimação de $\boldsymbol{\theta}$, a distribuição assintótica de $\hat{\boldsymbol{\theta}}$ é:

$$\hat{\boldsymbol{\theta}} \sim \mathrm{N}(\boldsymbol{\theta}, \mathrm{J}_{\boldsymbol{\theta}}^{-1}),$$

onde $\mathrm{J}_{\boldsymbol{\theta}}^{-1}$ é o inverso da matriz de Godambe (Hwang e Basawa, 2011),

$$\mathrm{J}_{\boldsymbol{\theta}}^{-1} = \mathrm{S}_{\boldsymbol{\theta}}^{-1}\mathrm{V}_{\boldsymbol{\theta}}\mathrm{S}_{\boldsymbol{\theta}}^{-T},$$

onde $\mathrm{S}_{\boldsymbol{\theta}}^{-T} = (\mathrm{S}_{\boldsymbol{\theta}}^{-1})^{T}$. Para mais detalhes, ver Bonat e Jørgensen (2016a).

3.5 MEDIDAS DE QUALIDADE DE AJUSTE

Em análise de dados, métricas de comparação de modelos são fundamentais para a escolha e definição dos componentes do modelo. No caso GLM, as métricas de critério de informação como *Akaike* e *Bayesian information criterion* são comumente utilizadas para escolha desses componentes; no MCGLM, adaptamos-no para produzir métricas semelhantes. Carey VJ (2011) propôs a pseudo log-verosimilhança gaussiana, dada por:

$$\text{plogLik}(\boldsymbol{\theta}) = -\frac{NR}{2}\log(2\pi) - \frac{1}{2}\log|\hat{\boldsymbol{C}}| - (\boldsymbol{\mathcal{Y}} - \hat{\boldsymbol{\mathcal{M}}})^{\top}\hat{\boldsymbol{C}}^{-1}(\boldsymbol{\mathcal{Y}} - \hat{\boldsymbol{\mathcal{M}}}),$$

onde NR é o numero de observações, $\hat{M}$ e $\hat{C}$ denotam o vetor de valores esperados e a matriz de covariâncias estimadas.

Usamos a pseudo log verosimilhança gaussiana para calcular duas estatísticas similares às métricas *Akaike* e *Bayesian information criterion*. A pseudo versão de *Akaike information criterion* é dada por:

$$\text{pAIC}(\boldsymbol{\theta}) = 2(P + Q) - 2\text{plogLik}(\boldsymbol{\theta}).$$

Enquanto que a pseudo versão *Bayesian information criterion* é dada por:

$$\text{pBIC}(\boldsymbol{\theta}) = logNR(P + Q) - 2\text{plogLik}(\boldsymbol{\theta}).$$

3.6 COMPONENTES DO MODELO

A função de ligação de covariância $h(\cdot)$ é descrita por Chiu TYM (1996). Para citar alguns exemplos populares: `identity`, `inverse` and `exponential-matrix`.

A função de variância é fundamental ao MCGLM, pois define a distribuição marginal de uma variável resposta. Para destacar algumas escolhas usuais, a função de variância *power* é especialista no trato de dados contínuos e fundamenta a família de modelos de distribuição `Tweedie`. De acordo com (Jørgensen, 1987, 1997), essa família possui seus casos emblemáticos: *Gaussiano* (*power* = 0), *Gamma* (*power* = 2) e *Gaussiano Inverso* (*power* = 3). Para a análise de dados limitados, a função de variância *extended binomial* é uma escolha usual. Para o ajuste de dados de contagem, a função de dispersão apresentada por (Jørgensen, 2015), denominada *Poisson-Tweedie* é flexivel para capturar modelos notáveis, como: *Hermite* (*power* = 0), *Neyman Type A* (*power* = 1), *Binomial Negativa* (*power* = 2) and *Gaussiana Poisson-inverse* (*power* = 3). A tabela a seguir sumariza as funções de variância citadas:

Nome da função	Fórmula
`power/Tweedie`	$\text{V}(.; p_r) = \mu_{p_r}$
`binomial`	$\text{V}(.; p_r) = \mu^{p_r}(1 - \mu_r)^{p_r}$
`Poisson-Tweedie`	$\text{V}(.; p) = \mu + \mu_p$

Tabela 3.1: Tabela com funções de variância

No MCGLM, o usuário especifica a dependência através das matrizes de dependência Z, fundamentando o perfil flexível do modelo. Muitos dos clássicos modelos estatísticos são

replicáveis através da especificação das matrizes Z. Para incutir na especificação destas matrizes, esta seção demonstra três casos emblemáticos, para *datasets* com três observações.

3.6.1 Simetria composta ou permutável

A estrutura permutável de efeitos adiciona um efeito randômico para cada indivíduo, mantendo o efeito fixo (Verbeke et al., 2010). O preditor linear matricial é especificado abaixo.

$$\boldsymbol{\Omega}(\boldsymbol{\tau}) = \tau_0 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \tau_1 \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

3.6.2 Não-estruturado

O modelo não-estruturado adiciona efeitos randômicos aos pares de indivíduos, mantendo o efeito fixo. A formula abaixo demonstra o preditor linear matricial.

$$\boldsymbol{\Omega}(\boldsymbol{\tau}) = \tau_0 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \tau_1 \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \tau_2 \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} + \tau_3 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{3.3}$$

3.6.3 Médias Móveis

O modelo médias móveis permite o ajuste em séries temporais. Fica a critério do usuário a escolha do número de ordens de avaliação da série (Feller, 1957). O preditor linear matricial a seguir demonstra janela de tamanho 2.

$$\boldsymbol{\Omega}(\boldsymbol{\tau}) = \tau_0 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \tau_1 \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} + \tau_2 \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}. \tag{3.4}$$

Mais exemplos de dependências foram pormenorizados em Bonat e Jørgensen (2016a).

4 BIBLIOTECA PYTHON

A biblioteca `mcglm`, objeto desta dissertação, visa aprovisionar uma interface para a análise estatística suportada pelo modelo estatístico MCGLM, no padrão da biblioteca `statsmodels` (Seabold e Perktold, 2010). Devido à interface consolidada entre os praticantes de análise estatística em `Python`, espera-se que os usuários instilem celeremente no uso da biblioteca.

Essa implementação do `Python` visa replicar os cálculos do pacote `R mcglm` (Bonat, 2018) e os testes unitários validam as respostas esperadas entre as bibliotecas das funcionalidades principais; o código tem noventa e um por cento de cobertura de testes unitários. A Figura 4.1 demonstra a execução do mapeamento da cobertura de testes com a biblioteca *coverage*. O código fonte da biblioteca `Python` e o site para o `PyPI` estão disponíveis em: `https://github.com/jeancmaia/mcglm` e `https://pypi.org/project/mcglm/`.

```
========================================================================== 26 passed in 4.78s ====
(master_thesis) jean@pop-os:~/dev/Github/mcglm$ poetry run coverage report
Name                          Stmts   Miss  Cover
-------------------------------------------------
mcglm/__init__.py                 7      0   100%
mcglm/dependencies.py            51      6    88%
mcglm/mcglm.py                  484     78    84%
mcglm/mcglmcattr.py             252     21    92%
mcglm/mcglmmean.py               55      0   100%
mcglm/mcglmvariance.py           36      0   100%
mcglm/utils.py                   20      0   100%
tests/__init__.py                 0      0   100%
tests/test_dependencies.py       31      0   100%
tests/test_mcglm.py             286      0   100%
-------------------------------------------------
TOTAL                          1222    105    91%
```

Figura 4.1: Cobertura de testes do código

4.1 BIBLIOTECAS ESTRUTURAIS

Essa biblioteca é viável devido às notáveis bibliotecas de programação científica da linguagem `Python`. Além da mencionada `statsmodels`, as bibliotecas `numpy` (Harris, 2020) e `scipy` (Virtanen, 2020) são utilizadas pela `mcglm`. Esta seção descreve a contribuição das bibliotecas estruturais no desenvolvimento deste projeto.

A biblioteca `numpy` é consenso à programação científica em `Python`; abrangendo desde o desenvolvimento de projetos pessoais até populares bibliotecas de aprendizado de máquinas em Python. Para a biblioteca `mcglm`, o `numpy` contribui com a definição e manipulação de matrizes. As operações matriciais fundamentais do `mcglm`, como soma, multiplicação, produto *Hadamard*, inversão e decomposição *Cholesky*, são realizadas com operações implementadas pelo `numpy`. Página da biblioteca: `https://numpy.org/`.

A biblioteca `scipy` é fundamental ao desenvolvimento científico em `Python`. Ela provê operações eficientes em linguagem `Cython`; além de integração nativas com os pacotes `BLAS` e `LAPACK`, pacotes clássicos que implementam operações de algebra linear, decomposição espectral e mínimos quadrados (Anderson et al., 1990).

O `scipy` é fundamental ao `mcglm` em dois momentos. Pela API completa para operações de algebra linear e pela interface para manipulação e operações em matrizes esparsas. Um

exemplo é o método `block_diag` é utilizado no `mcglm` para a geração de matrizes bloco diagonal. As operações em matrizes esparsas trouxeram eficiência ao cálculo de derivadas da matriz C, que será detalhado no próximo capítulo. Durante o desenvolvimento do projeto, foi possível observar ganhos computacionais relevantes após a migração para `scipy.sparse`. Sites dos métodos mencionados `https://docs.scipy.org/doc/scipy/reference/linalg.html` e `https://docs.scipy.org/doc/scipy/reference/sparse.html`.

A biblioteca `mcglm` utiliza de dois importantes componentes da biblioteca `statsmodels`: as funções de ligação e a classe `GLMResults`. O primeiro caso, as funções de ligação são parametrizadas em muitos algoritmos estatísticos do `statsmodels`, como GLM e GEE. Ao `mcglm`, as funções de ligação desempenham papel crucial no cálculo do primeiro momento; as opções disponíveis na biblioteca refletem as atuais disponíveis no `statsmodels`. A classe `GLMResults` implementa o objeto de saída do treinamento, identificável pelo icônico relatório sumário. A biblioteca `mcglm` implementa a classe `MCGLMResults` para ajustar o `GLMResults` à apresentação necessária ao ajuste do modelo estatístico MCGLM. Página com as funções de ligação `https://github.com/statsmodels/statsmodels/blob/main/statsmodels/genmod/families/links.py`.

4.2 PROGRAMAÇÃO ORIENTADA A OBJETOS

Orientação a objetos é um paradigma de programação que permite a especificação de sistemas complexos em módulos abstraídos e extensíveis. Os objetos são definidos por classes, operações e atributos. Entre as práticas mais comuns para comunicação entre objetos, destacam-se a herança, polimorfismo, abstração e encapsulamento (Yilmaz et al., 2019).

Os princípios ao desenvolvimento de aplicações orientadas a objeto são apontadas pelo acrônimo `SOLID`. O primeiro princípio é de abertura e fechamento; os componentes devem ser implementados permitindo expansão, mas não permitindo modificações dos componentes existentes. O segundo princípio é o da substituição; os componentes podem ser substituidos, sem danos, se as interfaces forem mantidas. O terceiro princípio é a dependência inversa; os componentes são dependentes das funções e abstrações, ao invés das implementações concretas. Finalmente, o princípio da segregação por interface, apenas os componentes relevantes devem ser carregados (Martin, 2017).

A biblioteca `mcglm` foi implementada sob o paradigma da orientação a objetos e os preceitos dos princípios `SOLID`. Apesar da complexidade inerente do modelo estatístico, o código oferece razoável legibilidade aos desenvolvedores e rápida extensibilidade ao código base.

4.3 BIBLIOTECA MCGLM

A biblioteca `mcglm` aprovisiona a primeira interface para análise estatística com suporte do modelo MCGLM em `Python`, condicionado pela especificação de cinco componentes: função de ligação, função de variância, preditor linear, função de covariância de ligação e o preditor matricial linear.

4.3.1 Distribuição das classes

A Figura 4.2 apresenta a separação dos arquivos do `mcglm`. O arquivo `_init_.py` estabelece a indexação dos arquivos; `mcglm.py` hospeda as classes MCGLM, MCGLMParameters e MCGLMResults; o arquivo `dependencies.py` implementa os métodos para

geração de matrizes de dependencias; `mcglmcattr.py` implementa a classe MCGLMCAttributes; `mcglmmean.py` e `mcglmvariance.py` implementam as classes MCGLMMean e MCGLMVariance; `utils.py` implementa métodos auxiliares as classes.

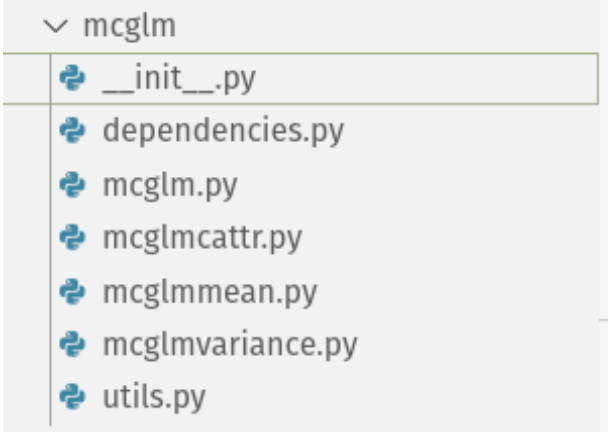


Figura 4.2: Apresentação da biblioteca mcglm

4.3.2 Interface da biblioteca

A documentação da classe, escrita via *docstrings*, descreve os parâmetros fundamentais para construção de objetos.

Listagem 4.1: Código Python para exibição dos parâmetros da biblioteca MCGLM.

```
from mcglm import MCGLM
print(MCGLM.__doc__)
```

Listagem 4.2: Documentação da biblioteca MCGLM.

```
   MCGLM class that implements MCGLM algorithm. (Bonat, Jorgensen 2015)

   It extends GLM for multi-responses and dependent components by fitting
   second-moment adjustment.

   Args:
      endog : array_like
         endog is a matrix with outcome variables. In case of multiple
         responses, the user must concatenate responses as one single
         numpy array. You might concantenate with method concat of Numpy
         library.
      exog : array_like
         A dataset with the endogenous matrix in a Numpy fashion.
         Since the library doesn't set an intercept by default, the
         user must add it. In the case of multiple responses, the user
         must pass the design matrices as a python list.
      z : array_like
         List with matrices components of the linear covariance matrix.
      link : array_like, string or None
         Specification for the link function. The MCGLM library
         implements the following options: identity, logit, power,
         log, probit, cauchy, cloglog, loglog, negativebinomial. In the
         case of None, the library chooses the identity link. In
         multiple responses, user must pass values as list.
```

```
25        variance : array_like, string or None
26           Specification for the variance function. The MCGLM library
27           implements the following options: constant, tweedie, binomialP,
28           binomialPQ, power, geom_tweedie, poisson_tweedie. In the case
29           of None, the library chooses the constant link. In multiple
30           responses, user must pass values as list.
31        offset : array_like or None
32           Offset for continuous or count. In multiple responses, user
33           must pass values as list.
34        ntrial : array_like or None
35           ntrial for binomial responses. In multiple responses, user
36           must pass values as list.
37        power_fixed : array_like or None
38           Parameter that allows estimation of power when variance
39           functions is either tweedie, geom_tweedie or poisson_tweedie.
40           In multiple responses, user must pass values as list.
41        maxiter : float or None
42           Number max of iterations. Defaults to 200.
43        tol : float or None
44           Threshold of minimum absolute change on parameters. Defaults
45           to 0.0001.
46        tuning : float or None
47           Step size parameter. Defaults to 0.5.
48        weights : array_like or None
49           Weight matrix. Defaults to None.
```

O argumento `endog` é um vetor, ou uma matriz, com as realizações da variável resposta e o argumento `exog` define as covariáveis relativas às respostas, através das matrizes de delineamento. Os dois argumentos replicam o padrão de nomes da biblioteca `statsmodels`. No caso de múltiplas respostas, `endog` e `exog` devem ser definidas via listas `Python`. O argumento `z` estabelece as matrizes de dependência, definidas em estruturas de `numpy` *array*.

Os argumentos `link` e `variance` definem as funções de ligação e variância. As ligações disponíveis são `identity`, `logit`, `power`, `log`, `probit`, `cauchy`, `cloglog`, `loglog`, `negativebinomial` e `inverse_power`; as opções disponíveis para variância são: `constant`, `BinomialP`, `BinomialPQ`, `tweedie`, `geom_tweedie` e `poisson_tweedie`; à exceção dos três últimos, as escolhas são usuais nos modelos lineares generalizados. *Tweedie* é modelo de distribuição de probabilidade que pode emular alguns modelos pertencentes à família de dispersão exponencial, condicionada à escolha do parâmetro *power*. *Poisson Tweedie* é um modelo que adiciona à uma variavel aleatória Poisson, efeitos aleatórios parametrizados por uma distribuição *Tweedie*; é um modelo adequado ao ajuste de dados de contagem. *Tweedie Geometrica* é um modelo de probabilidade que é gerado a partir da soma geométrica de realizações *Tweedie*.

Portanto, a escolha da tupla para as funções de ligação e variância é uma tarefa conceituada aos usuários do modelo linear generalizado. Casos estandardizados são: `identity` e `constant` para dados continuos; `log` e `power` definido em 2 para dados reais positivos de cauda longa; `logit`, `loglog`, `cauchy` ou `cloglog` para o caso de dados limitados; `log` e `tweedie` definido em 1 para o caso de valores inteiros/contagem. Os valores padrão para as funções de ligação e variância são `identity` e `constant`, escolhas adequadas a um ajuste gaussiano e variância constante.

O argumento `offset` é dedicado a variáveis resposta dos tipos contínuas e contagem. O parâmetro `ntrial` se refere ao número de ensaios de uma variável resposta do tipo binomial. Finalmente, `power_fixed` indica ao algoritmo a otimização do parâmetro *power*, apenas

válido para os casos de variância `tweedie`, `geom_tweedie` e `poisson_tweedie`. O parâmetros são definidos via listas, em caso de múltiplas respostas.

4.3.3 Um objeto ajustado

Com um objeto devidamente instanciado, é possível iniciar o processo de ajuste do modelo através do método `fit()`, que retorna um objeto da classe `MCGLMResults`. Esse objeto possui dois métodos: `summary()`, que é o relatório com as estimativas e métricas de qualidade de ajuste, e o `anova()`, o teste `ANOVA` para covariáveis categóricas.

Métodos	Descrição
`summary()`	Relatório completo com as informações do ajuste.
`anova()`	Teste ANOVA para variáveis categóricas.

Tabela 4.1: Tabela com os métodos de `MCGLMResults`

Outros atributos de um objeto ajustado são úteis na rotina de utilização da biblioteca. O atributo *mu* registra o vetor de valores esperados; *pearson_residuals* registra o vetor de resíduos normalizados de *Pearson*; *aic*, *bic* e *loglikelihood* são as conhecidas métricas comparativas de modelos, para citar algumas.

4.3.4 Métodos auxiliares

Além disso, os usuários podem usar métodos auxiliares para a definição do preditor linear matricial, ou matrizes de dependência, com a biblioteca `mcglm`. Os métodos que estão disponíveis sob os chamadores: `mc_id()`, `mc_mixed()`, `mc_ma()`. Os métodos implementam o caso independente, modelos mistos e médias móveis.

4.3.5 Exemplo de aplicação da biblioteca

```
1 import pandas as pd
2 df = pd.DataFrame([])
3
4 df['id'] = [1, 1, 1, 1, 2, 2, 2, 2]
5 df['time'] = [1, 2, 3, 4, 1, 2, 3, 4]
```

Os componentes do preditor linear matricial, por exemplo, para médias móveis de primeira ordem MA(1) é dado por:

Listagem 4.3: Criação das matrizes de depêndencia do MCGLM.

```
1 Z0 = mc_id(data)
2 Z1 = mc_ma(id='id', time='time', data=df, order=1)
```

4.3.6 Instalação da Biblioteca

A biblioteca está disponível no repositório `Pypi` e pode ser facilmente instalado pelo gestor de pacotes `pip` usando o comando:

Listagem 4.4: Instalação da biblioteca MCGLM.

```
1 pip install mcglm
```

Aditivamente, o código fonte pode ser acessado no `GitHub` pela url: `https://github.com/jeancmaia/mcglm`.

5 ARQUITETURA DE SOFTWARE E IMPLEMENTAÇÃO

O diagrama UML da Figura 5.1 apresenta a arquitetura de software da biblioteca `mcglm`, desenvolvida sob o paradigma da orientação a objetos. A classe principal `MCGLM` possui o método `fit()` para acionar o ajuste do modelo, onde os seus parâmetros devem ser passados na instanciação do objeto. A implementação do modelo estatístico na linguagem `Python` reside em seis classes: `MCGLM`, `MCGLMMean`, `MCGLMVariance`, `MCGLMCAttributes`, `MCGLMParameters` e `MCGLMResults`.

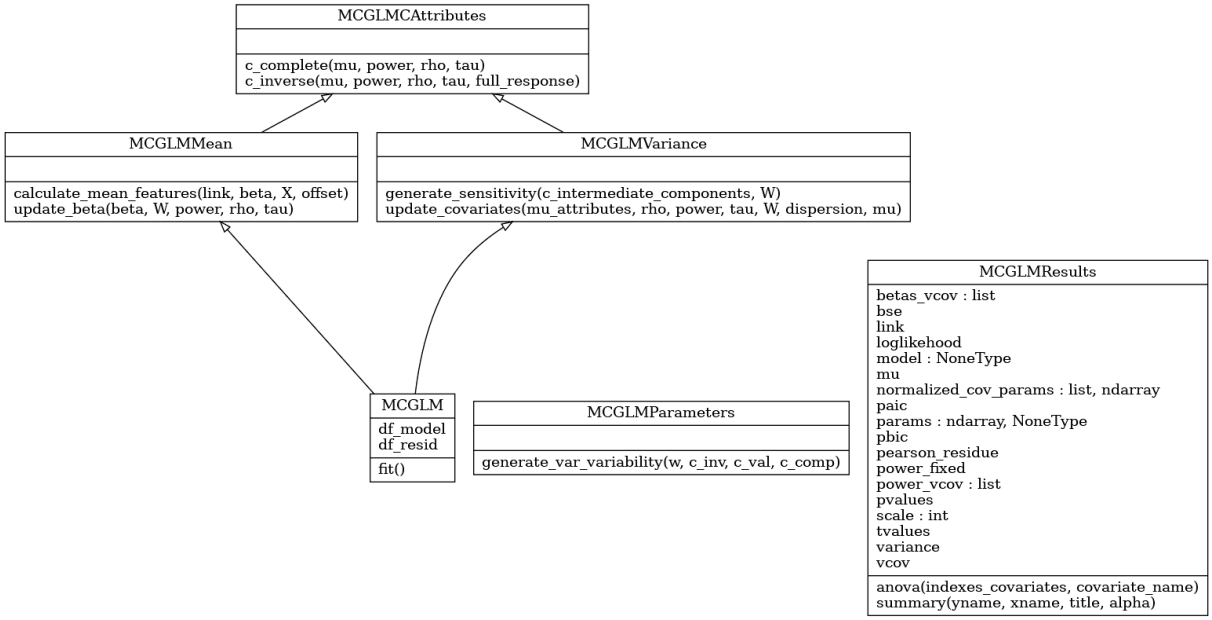


Figura 5.1: UML - arquitetura de software

As classes da biblioteca possuem atribuições delimitadas e refletem os principais componentes do framework `MCGLM`, descrito no capítulo anterior. A Figura 5.2 detalha a relação operacional inter-classes.

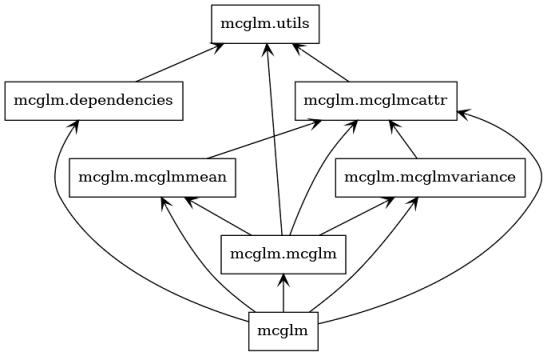


Figura 5.2: Python arquivos e classes

5.1 CLASSE MCGLM

A classe fundamental da biblioteca é a `MCGLM`, que provê a interface de treinamento e acesso às estimativas resultantes do modelo. Essa classe deve ser instanciada pelo usuário para a execução do treinamento. A seguir, um exemplo demonstrando o uso da biblioteca `mcglm`.

Listagem 5.1: Demonstração do uso da biblioteca mcglm.

```
from mcglm import MCGLM, mc_id

mcglm = MCGLM(
   endog=y,
   exog=X,
   z=[mc_id(X)],
   link="log",
   variance="power"
   power=2
)

mcglmresults = mcglm.fit()
mcglmresults.summary()
```

A classe `MCGLM` herda as classes `MCGLMMean` e `MCGLMVariance`, que serão propriamente descritas neste capítulo. O método principal `__init__()`, chamado no momento de instanciação do objeto, possui duas atribuições: Definir os métodos passados via parâmetro como atributos da classe e calcular os valores iniciais otimizados com o método `_calculate_static_attributes()`. Esse método inicia com a definição dos valores padrão que não foram especificados pelo usuário como `identity` e `constant`, caso as funções de ligação e variância não sejam especificadas. Finalmente, o método gera os vetores iniciais de regressão, `tau` e `rho`. O primeiro possui a dimensão do número de covariáveis. O vetor `tau` possui dimensão igual ao número de matrizes `Z` passadas e representa os parâmetros de dispersão. `rho` é o vetor para o retorno dos coeficientes de correlação entre as respostas, sua dimensão é calculada com o suporte do script Python abaixo, implementado na biblioteca principal.

Listagem 5.2: Cálculo do tamanho do vetor de dispersão.

```
rho = list()
for _ in range(int(n_targets * (n_targets - 1) / 2)):
   rho.append(0)
```

O método `fit()` executa o ajuste do modelo caso a instanciação do modelo seja completada com sucesso. O método executa quatro tarefas: execução do método `_fit()`, que implementa o ajuste do framework MCGLM; o parseamento do histórico das matrizes de regressão e dispersão; cálculo das métricas: *pseudo-loglikelihood*, *akaike information criterion* e *bayesian information criterion*; fim da execução com o retorno de objeto da class `MCGLMResults`, que será detalhado neste capítulo.

O método `fit()` implementa o ajuste de dois momentos do framework estatístico `MCGLM`. Inicialmente, o método `_update_optimal_dispersion()` é chamado para o cálculo dos valores iniciais otimizados para os vetores de regressão e dispersão, calculados a partir de estimativas de modelos lineares generalizados que compartilham da mesma função de ligação. O vetor com as estimativas resultantes é usado como o vetor inicial para o modelo `mcglm` e o parâmetro de dispersão, acessado pelo attributo `scale`, é definido como o primeiro valor no vetor `tau` gerado. A seguir, o método `_update_optimal_dispersion()`, que busca o vetor otimizado.

Listagem 5.3: Método responsável pelo vetor otimizado.

```
    def __update_optimal_dispersion(self):
        """Update optimal dispersion method calculates optimal values for
        regression parameters and the dispersion. It harnesses the GLM API of
        statsmodels for calculating those values.
        """
        def logit_est(endog, exog, offset):
            """For the Logit link function, the method adjusts a GLM with
            Binomial family. It retrieves the parameters specified.

            Args:
                endog (array-like): outcome variable.
                exog (array-like): independent variables.
                offset (int): shift on response variable.

            Returns:
                tuple: regression parameters, dispersion parameter.
            """
            ...
            return mdl_results.params.values, mdl_results.scale

        def loglog_est(endog, exog, offset):
            """For the LogLog link function, the method adjusts a GLM with
            Binomial family. It retrieves the parameters specified.

            Args:
                endog (array-like): outcome variable.
                exog (array-like): independent variables.
                offset (int): shift on response variable.

            Returns:
                tuple: regression parameters, dispersion parameter.
            """
            ...
            return mdl_results.params.values, mdl_results.scale

        def cloglog_est(endog, exog, offset):
            """For the CLogLog link function, the method adjusts a GLM with
            Binomial family. It retrieves the parameters specified.

            Args:
                endog (array-like): outcome variable.
                exog (array-like): independent variables.
                offset (int): shift on response variable.

            Returns:
                tuple: regression parameters, dispersion parameter.
            """
            ...
            return mdl_results.params.values, mdl_results.scale

        def cauchy_est(endog, exog, offset):
            """For the Cauchy link function, the method adjusts a GLM with
            Binomial family. It retrieves the parameters specified.

            Args:
                endog (array-like): outcome variable.
                exog (array-like): independent variables.
```

```
58              offset (int): shift on response variable.
59
60          Returns:
61              tuple: regression parameters, dispersion parameter.
62          """
63          ...
64          return mdl_results.params.values, mdl_results.scale
65
66      def probit_est(endog, exog, offset):
67          """For the Cauchy link function, the method adjusts a GLM with
68          Binomial family. It retrieves the parameters specified.
69
70          Args:
71              endog (array-like): outcome variable.
72              exog (array-like): independent variables.
73              offset (int): shift on response variable.
74
75          Returns:
76              tuple: regression parameters, dispersion parameter.
77          """
78          ...
79          return mdl_results.params.values, mdl_results.scale
80
81      def identity_est(endog, exog, offset=None):
82          """For the Identity link function, the method adjusts a OLS. It
83          retrieves the parameters specified.
84
85          Args:
86              endog (array-like): outcome variable.
87              exog (array-like): independent variables.
88              offset (int): shift on response variable.
89
90          Returns:
91              tuple: regression parameters, dispersion parameter.
92          """
93          ...
94          return mdl_results.params, mdl_results.scale
95
96      def log_est(endog, exog, offset):
97          """For the Log link function, the method adjusts a GLM with Tweedie
98          (power=1) family. It retrieves the parameters specified.
99
100         Args:
101             endog (array-like): outcome variable.
102             exog (array-like): independent variables.
103             offset (int): shift on response variable.
104
105         Returns:
106             tuple: regression parameters, dispersion parameter.
107         """
108         ...
109         return mdl_results.params, mdl_results.scale
110
111     def power_est(endog, exog, offset):
112         """For either the Power or Reciprocal link function, the method
113         adjusts a GLM with Tweedie(power=2) family. It retrieves the
114         parameters specified.
115
```

```
116             Args:
117                 endog (array-like): outcome variable.
118                 exog (array-like): independent variables.
119                 offset (int): shift on response variable.
120 
121             Returns:
122                 tuple: regression parameters, dispersion parameter.
123             """
124             ...
125             return mdl_results.params.values, mdl_results.scale
126 
127         def negative_binomial_est(endog, exog, offset):
128             """For the Negative Binomial link function, the method adjusts a
129             GLM with Tweedie(power=2) family. It retrieves the parameters
130             specified.
131 
132             Args:
133                 endog (array-like): outcome variable.
134                 exog (array-like): independent variables.
135                 offset (int): shift on response variable.
136 
137             Returns:
138                 tuple: regression parameters, dispersion parameter.
139             """
140             ...
141             return mdl_results.params.values, mdl_results.scale
142 
143         first_estimation = {
144             "logit": logit_est,
145             "identity": identity_est,
146             "power": power_est,
147             "log": log_est,
148             "probit": probit_est,
149             "cauchy": cauchy_est,
150             "cloglog": cloglog_est,
151             "loglog": loglog_est,
152             "negativebinomial": negative_binomial_est,
153             "inverse_power": log_est,
154             "reciprocal": power_est
155         }
```

Após o cálculo dos valores iniciais otimizados, o método `fit()` atualiza o vetor de dispersão, com as variáveis `rho`, `power` e `tau` com o método `_create_dispersion_vector()`. O vetor de dispersão pode não receber os valores de *power*, caso o usuário não demande a busca do *power* otimizado, e pode não receber o `rho`, para o ajuste em uma resposta. O vetor de dispersão registra os valores de acordo com a especificação do capítulo anterior.

Finalmente, o método está pronto para iniciar o ajuste de dois momentos, o atributo `_max_iter` define o total máximo de iterações para a busca das estimativas. No processo iterativo, primeiro e segundo momentos são ajustados ordenadamente. O primeiro momento é ajustado com o auxílio da função `update_beta()`, especificado pela classe `MCGLMMean`, enquanto que o segundo momento é ajustado pelo método `update_covariances()`, implementado na classe `MCGLMVariance`. Ao fim, o método `_check_stop_criterion()` avalia a parada na iteração corrente, retornando positivo caso o ajuste dos parâmetros seja inferior

ao atributo `_tol`. A seguir, a implementação do método de otimização do `mcglm`, com as chamadas aos dois métodos de otimização.

Listagem 5.4: Seção com o método de otimização do modelo MCGLM.

```
for iter in range(self._max_iter):
# First moment
(
   new_regression,
   quasi_score,
   mean_sensitivity,
   mean_variability,
) = self.update_beta(regression, W, power, rho, tau)
regression_historical.append(new_regression)

# Second moment
mu_attributes, mu, mu_derivatives = self.calculate_mean_features(
   self._link, new_regression, self._X, self._offset
)
(
   new_dispersion,
   c_inverse,
   c_values,
   c_derivatives_componentes,
   var_sensitivity,
) = self.update_covariances(
   mu_attributes, rho, power, tau, W, dispersion, mu
)
dispersion_historical.append(new_dispersion)

regression, dispersion, rho, power, tau = self.__iteration_update(
   new_regression, new_dispersion, rho, power, tau
)

if self.__check_stop_criterion(
   regression_historical, dispersion_historical
):
   break
```

A execução é finalizada com os cálculos das matrizes de variância e covariância, sensitividade e variabilidade, com o auxílio da classe `MCGLMParameters` e o parseamento do vetor de dispersão para o retorno adequado à instanciação da classe `MCGLMResults`.

5.2 CLASSE MCGLMMEAN

A classe `MCGLMMean` é responsável pelo tratamento do primeiro momento e dos parâmetros de regressão. Os dois principais métodos da classe são: `update_beta()` e `calculate_mean_features()`, na qual o fluxo para atualização dos paramêtros de regressão é plenamente desenvolvido. As funções de ligação foram importadas do pacote statsmodels `https://github.com/statsmodels/statsmodels/blob/main/statsmodels/genmod/families/links.py`.

As opções de função de ligação disponíveis são: `logit`, `identity`, `power`, `log`, `probit`, `cauchy`, `cloglog`, `loglog`, `negativebinomial` e `inverse_power`. Por fim, a classe possui uma relação de herança com a classe `MCGLMAttributes`.

O método `calculate_mean_features()` calcula o vetor resposta esperado e a sua derivada de primeira ordem. O primeiro componente é calculado pelo inverso da função

ligação, aplicado à operação linear entre o vetor de estimativas de regressão corrente e as covariáveis. O segundo é a multiplicação da matriz de covariáveis e a inversa da função de ligação. A seguir, os principais métodos da classe `MCGLMMean`, com a operação linear e o método de otimização.

Listagem 5.5: Métodos fundamentais da classe MCGLM com as operações de predição linear e otimização.

```
1
2   class MCGLMMean(MCGLMCAttributes):
3   """
4   MCGLMMean is the class for the first moment adjustment within MCGLM
5   inference. It handles lifecycle completely, ranging from mu and
6   derivatives to quasi-likelihood calculations.
7   This class has two interfaces: 'calculate_mean_features' which calculates
8   mu attributes, and 'update_beta' that applies quasi-likelihood estimation
9   and retrieves a new beta.
10  This class implements the Estimating Equation Quasi-score (Wedderburn,
11  1974) and the second-order optimization algorithm (Jennrich, 1969) and
12  (Widyaningsih et al., 2017).
13
14  References
15  ----------
16  Wedderburn, R. W. M. (1974). Quasi-likelihood functions, generalized
17  linear models, and the Gauss-Newton method. Biometrika, 61(3) 439-447.
18
19  Jennrich, R. I. (1969). A Newton-Raphson algorithm for maximum likelihood
20  factor analysis. Psychometrika, 34.
21
22  Widyaningsih, P., Saputro, D. e Putri, A. (2017). Fisher scoring method
23  for parameter estimation of geographically weighted ordinal logistic
24  regression (gwolr) model. Journal of Physics: Conference Series,
25  855:012060.
26  """
27
28  def __init__(self):
29     super(MCGLMCAttributes, self).__init__()
30
31  def __get_link_function(self, link: str):
32     """Check whether the link function is available
33
34     Parameters
35     ----------
36        link : str
37           A link function
38     Returns
39     -------
40        statsmodels.genmod.families.links : a corresponding object for the
41        link function.
42     """
43     assert link in AVAILABLE_LINK_FUNCTIONS, (
44        f"The link function " + str(link) + " isn't available"
45     )
46     ...
47
48  def __linear_predictor(self, X, beta):
49     """Method linear predictor applies linear operation between
50     covariances
51     and the regression parameters.
```

```
52
53          Parameters
54          ----------
55              X : array_like
56                  Design matrix with covariances
57              beta : array-like
58                  Regression parameters
59          Returns
60          -------
61              array_like : The calculated output vector.
62          """
63          ...
64
65      def _link_function_attributes(
66          self, link: str, beta: np.array, X: np.array, offset: int = 0
67      ):
68          """A protected method for calling the __link_function_attributes
69
70          Parameters
71          ----------
72              link : str
73                  Link function.
74              beta : array_like
75                  Regression Parameters.
76              X : array_like
77                  Matrix with covariances.
78              offset : (int, optional)
79                  Offset add value. Defaults to 0.
80          Returns
81          -------
82              dict : Dict value with the mean and its derivatives.
83          """
84          ...
85
86      def __link_function_attributes(
87          self, link: str, beta: np.array, X: np.array, offset: int = 0
88      ):
89          """The method __link_function_attributes calculates the vector of
90          expected values and its derivatives. It returns data as dictionary
91
92          Parameters
93          ----------
94              link : str
95                  Link function.
96              beta : array-like
97                  Regression Parameters.
98              X : array-like
99                  Matrix with covariances.
100             offset : int, optional
101                 Offset add value. Defaults to 0.
102         Returns
103         -------
104             dict : Dict value with the mean and its derivatives.
105         """
106         ...
107
108     def calculate_mean_features(self, link, beta, X, offset):
109         """Base method to calculate every attribute related to the mean.
```

```
110
111         Parameters
112         ----------
113             link : str
114                 Link function.
115             beta : array-like
116                 Regression Parameters.
117             X : array-like
118                 Matrix with covariances.
119             offset : int, optional
120                 Offset add value. Defaults to 0.
121         Returns
122         -------
123             tuple : Mean attributes, the raw mean and its derivatives.
124         """
125         ...
126
127     def update_beta(self, beta, W, power, rho, tau):
128         """The method update_beta takes the current beta, leverages the
129         quasi-likelihood estimator to calculate the next regression parameters.
130
131         Parameters
132         ----------
133             beta : array-like
134                 Regression Parameters.
135             W : array-like
136                 Weight matrix
137             power : float
138                 Power parameter
139             rho : float
140                 Correlation parameters
141             tau : float
142                 Dispersion parâmeters.
143         Returns
144         -------
145             tuple : A tuple with the new regression parameters, the
146             quasi-score parameter, sensitivity and the variability matrix.
147         """
148         ...
```

O método `update_beta()` recebe um vetor de regressão e entrega o novo vetor atualizado. O método executa quatro passos: cálculo do valor esperado e sua derivada com o método `calculate_mean()`; cálculo da matriz inversa de C com o método `c_inverse()` da classe `MCGLMCAttributes`; aplicação da equação de estimação `Quasi-score`, implementado no método `_quasi_score()`; execução do passo de otimização com o método de otimização *Fisher Score*, implementado no método `_update_fisher_score()`.

O método `_quasi_score()` implementa o método `Quasi-score` com operações matriciais `numpy`. O método retorna o valor de `score` e as matrizes sensitividade e variabilidade. A seguir, o código do método de otimização `Quasi-score`.

Listagem 5.6: Implementação do método de estimador de quasi-verossimilhança.

```
1     def __quasi_score(self, mu_derivative, c_inverse, y, mu, W):
2         """Quasi-score functions are estimating equations for the Maximum
3         Likelihood Estimator (Wedderburn, 1974). This method harnesses the
4         Numpy backbone to produce a classic method of statistical models.
5
6         Parameters
```

```
7         ----------
8            mu_derivative : array-like
9               First-order derivatives of means.
10           c_inverse : array-like
11              Inverse of C matrix
12           y : array-like
13              A vector with outcome variable.
14           mu : array-like
15              A vetor with mean parameters.
16           W : array-like
17              A matrix of weights.
18
19        Returns
20        -------
21           tuple : A tuple with score, sensitivity and variability.
22        """
23        ...
```

O método `_update_fisher_score()` usa da matriz de sensitividade e do score para atualizar o vetor de regressão. A implementação do método reflete a especificação matemática do otimizador, moldado em operações `numpy`. O método segue descrito abaixo:

Listagem 5.7: Método para otimização dos parâmetros de regressão.

```
1     def __update_fisher_score(self, sensitivity, score, beta):
2        """A private method that implements the second-order optimization
3        algorithm Fisher-scoring.
4
5        Parameters
6        ----------
7           sensitivity : array-like
8              Sensitivity matrix
9           score : array-like
10             Quasi-score output.
11          beta : array-like
12             Regression parameters.
13
14       Returns
15       -------
16          array-like : New regression vector.
17       """
18       ...
```

5.3 CLASSE MCGLMVARIANCE

A classe `MCGLMVariance` possui a atribuição da atualização dos parâmetros de dispersão. Os componentes de *C* são fundamentais à execução, por isso a classe possui uma relação de herança com a classe `MCGLMCAttributes`.

O método `update_covariances()` é responsável por arquitetar a execução de atualização do vetor de dispersão. A atualização do vetor de dispersão é conquistada em quatro etapas: cálculo dos atributos da matriz *C* através do método `c_complete()`, que será devidamente explanado na seção do `MCGLMCAttributes`; aplicação da equação de estimação de *Pearson*; aplicação do algoritmo de otimização *Chaser*; retorno dos atributos gerados.

O método `_pearson_estimating_equation()` implementa a equação de estimação de `Pearson`. A equação demanda inúmeras operações matriciais entre as derivadas da

matriz C, inverso da matriz C, resíduo e a matriz W. Após a geração dos valores da equação de estimação, o método de otimização chaser está pronto para executar seu ajuste. A seguir, o código com a função de estimação `Pearson`.

Listagem 5.8: A classe `MCGLMVariance`.

```
class MCGLMVariance(MCGLMCAttributes):
"""
The MCGLMVariance class handles the second optimization of the MCGLM
second-moment assumptions, therefore, the step for variance. It implements
every step of Variance within the scope of the MCGLM algorithm, using many
attributes to be specified as attributes. A general class must inherit
this MCGLMVariance and leverages its methods properly. MCGLM is in charge
of setting the fundamental python attributes and modules orchestration for
a complete mcglm adjustment.

The variance step on the optimization sketch boils down to Pearson
estimating equations and the chaser algorithm for optimization. The latter
uses tuning to set the step size of each iteration.

Heavy operations regarding C components, pivotal to the chaser
optimization step, are implemented on the MCGLMAttricutes class, inherited
here. The method _c_complete, the one that crafts all of the three
attributes thoroughly, is comprehensive for the variance calculation in
this class.
"""

def __init__(self):
   super(MCGLMCAttributes, self).__init__()

def update_covariances(self, mu_attributes, rho, power, tau, W,
dispersion, mu):
   """The method update_covariances implements a cycle of iteration for
   the second-moment estimation, the variance.

   Parameters
   ----------
      mu_attributes : dict
         A dict with mean and derivatives.
      rho : array_type
         Parameters of correlation.
      power : float
         A parameter for Power Tweedie distribution.
      tau : float
         Dispersion parameters.
      W : array_type
         A weight matrix.
      dispersion : array-type
         A vector with dispersion parameters.
      mu : array_type
         A vector with mean parameters.
   Returns
   -------
      tuple: A tuple with new vector of dispersion vector, atributes of
      matrix C, sensitivity.
   """
   ...

def __chaser_step(self, score, sensitivity):
```

```
        """The protected method chaser step calculates the step for a
        optimization step.

        Parameters
        ----------
            score : array_type
                A vector with quasi-score values.
            sensitivity : array_type
                The sensitivity matrix.
        Returns
        -------
            array_type: The absolute change to operate on vector.
        """
        ...

    def __pearson_estimating_equation(self, mu, W, c_inverse, c_derivative):
        """The estimating equation for the dispersion parameters.

        Parameters
        ----------
            mu : array_type
                A vetor with mean parameters
            W : array_type
                A weight matrix
            c_inverse : array_type
                The inverse of C Matrix
            c_derivative : array_type
                The derivatives of C Matrix
        Returns
        -------
            tuple : a tuple with score, sensitivity matrix and the matrix C
            normalized by Pearson.
        """
        ...

    def __core_pearson(self, c_component, c_inverse, residue, W):
        """The protected method core_pearson handles the inner-components of
        Pearson estimation equations operations.

        Parameters
        ----------
            c_component : array_type
                A matrix with components of C.
            c_inverse : array_type
                The inverse of matrix C.
            residue : array_type
                The difference between mean and an outcome variable.
            W : array_type
                A weight matrix.
        Returns
        -------
            array_type : A matrix with the core pearson.
        """
        ...

    @staticmethod
    def generate_sensitivity(c_intermediate_components, W):
        """The method to create the sensitivity matrix.
```

```
112
113         Parameters
114         ----------
115             c_intermediate_components : array_type
116                 Intermediate components of C matrix.
117             W : array_type
118                 A weight matrix.
119         Returns
120         -------
121             array_type : A sensitivity matrix
122         """
123         ...
```

O método `_chaser_step()` usa do vetor `score` e da matriz de sensitividade para a atualização dos valores de dispersão. O parâmetro *tuning*, definido na instânciação do objeto `MCGLM` estabelece o tamanho do passo adicionado à cada execução. O código a seguir registra a implementação do método.

Listagem 5.9: Algoritmo de otimização `Chaser`.

```
1 def __chaser_step(self, score, sensitivity):
2     return self._tuning * solve(sensitivity, score)
```

5.4 CLASSE MCGLMCATTRIBUTES

A classe `MCGLMCAttributes` é responsável pelos cálculos atrelados às matrizes de C, apontada como a variância na especificação do `MCGLM`. Inegavelmente, essa classe implementa as operações mais complexas computacionalmente do ajuste do modelo. Os métodos `c_inverse()` e `c_complete()` são utilizados pelas classes `MCGLMMean` e `MCGLMVariance` respectivamente. O último método compõe a resposta do primeiro com a matrix C e suas derivadas.

O método `c_inverse()` cria interface de interação para a geração do inverso de C. O passo-a-passo é: criação do $\Omega(\tau)$; criação do Σ, juntamente com a decomposição *Cholesky*; criação do Σ entre respostas; geração da matriz C inversa.

O método responsável pela criação da matriz $\Omega(\tau)$ é o `_generate_omega()`, que aplica a operação preditor linear matricial especificada no `MCGLM` entre os componentes `tau` e Z. Em posse de $\Omega(\tau)$, é possivel construir o componente Σ, com o suporte do método `_calculate_sigma()`. O método está descrito abaixo.

Listagem 5.10: Algoritmo de otimização `Chaser`.

```
1     def _generate_omega(self, tau):
2         """Base method to calculate derivatives related to correlation
3         parameters.
4
5         Parameters
6         ----------
7             tau : list
8                 List with dispersion parameters
9         Returns
10        -------
11            list : the results of matrix linear sum between tau and dependence
12            matrices.
13        """
14        ...
```

A função de variância é aplicada no cálculo dos componentes da matriz Σ. O método retorna os valores do sigma bruto, do resultado da decomposição *Cholesky* e o inverso dessa decomposição. A biblioteca usa dos métodos `cholesky()` e `inv()` do pacote `numpy` para a execução. No cenário de variância `constant`, o Σ bruto assume o valor do Ω integralmente. Para os cenários da variância pertencente a lista: `tweedie`, `binomialP`, `binomialPQ`, `power`, o cálculo do Σ bruto é resultante da operação `sandwich` entre o Ω e a raiz da função de variância aplicada aos valores esperados. Finalmente, para os casos de `poisson_tweedie` ou `geom_tweedie`, o Σ bruto segue a operação do caso anterior com a adição da matrix diagonal com os valores esperados. Página com as operações de álgebra linear `numpy`: `https://numpy.org/doc/stable/reference/routines.linalg.html`.

Listagem 5.11: Método para operações no σ.

```
def _calculate_sigma(
    self, mu, power, omega, variance, Ntrial, covariance="identity"
):
    """Base method to calculate sigma

    Parameters
    ----------
        mu : array_like
            A vector with expected values.
        power : float
            A power parameter.
        omega : array_like
            The omega resulted matrix.
        variance : str
            The variance function.
        Ntrial : int
            The number of trial. Parameter for Binomial distribution.
    Returns
    -------
        array_like : A matrix with Sigma values.
    """
    ...
```

A matriz de variância-covariância entre respostas é calculado pela criação de uma matriz diagonal com os valores do vetor `rho`.

A matriz inversa de *C* é calculada pelo método `_generate_c_inverse_and_blocks()`. Os parâmetros utilizados pelo método são: A decomposição `cholesky` do variância-covariância; o inverso dela; e o variância-covariância entre respostas. Esse método conta com o auxílio de dois métodos terceiros: `block_diag()`, do `scipy`, para a criação de matrizes bloco diagonal e `kron()` do `numpy`, para o produto *Kronecker*. O método prepara a execução construindo duas matrizes bloco diagonal, a partir das matrizes `sigma_chol` e `sigma_chol_inv`; gera a matrix inversa da Σ entre respostas; obtém o inverso de *C* aplicando a multiplicação encadeada de matrizes, descrito na sequência. A seguir, o código desse método.

Listagem 5.12: Método que gera o inverso da Matriz *C*.

```
def __generate_c_inverse_and_blocks(
    self,
    sigma_chol,
    sigma_chol_inv,
    sigma_between,
    diagonal_matrix,
):
```

```
        """Base method to calculate derivatives related to correlation
        parameters.

        Parameters
        ----------
           sigma_chol : array_like
              The sigma matrix decomposed by Cholesky.
           sigma_chol_inv : array_like
              The inverse of sigma matrix decomposed by Cholesky.
           sigma_between : array_like
              The inner-matrix among sigmas.
           diagonal_matrix : array_like
              A diagonal matrix.
        Returns
        -------
           array_like : A matrix with derivatives related to the C Matrix.
        """
        ...
```

Esse método calcula os três componentes de C: inverso de C, derivadas de C e o C bruto. Foi produzido para suportar o ajuste do segundo momento.

Após gerar o C inverso com o método `_generate_c_inverse()`, descrito na subseção anterior, a matriz derivada de C é obtida pelo cálculo da derivada do sigma e derivida de C. A matriz C bruta é gerada no fim do processo. Segue o método que implementa todos os componentes de C:

Listagem 5.13: Método para processar os métodos da matriz C.

```
    def c_complete(self, mu, power, rho, tau):
        """
        A method to generate the whole list of C components, explicitly made
        for the variance treatment step. This method interacts with sigma and
        omega crafting practices, passing the list of each parameter.

        Parameters
        ----------
           mu : array_like
              A vetor with mean parameters.
           power : float
              Power parameter.
           rho : float
              Correlation parameter.
           tau : float
              Dispersion parameter.
        Returns
        -------
           tuple : A tuple with every component of C.
        """
        ...
```

O método `_generate_sigma_derivatives()` gera as derivadas de σ. O cálculo é semelhante ao calculo do σ, adicionado a um tratamento especial ao `power`, caso o usuário requisite o ajuste do `power`. O cálculo envolve a operação `sanduiche power` com o Ω, raiz da variância nos valores esperados e a sua derivada. O método é apresentado a seguir.

Listagem 5.14: Método com o cálculo das derivadas de σ.

```
    def _calculate_sigma_derivatives(
```

```
        self, mu, power, variance, z, power_fixed, Ntrial, omegas,
        covariance="identity"
    ):
        """
        Base method for computing variance-covariance matrix, based on
        variance function and omega matrix. This method will implement for
        cases where covariance is equal to identity, and variance falls in the
        list:
        ['constant', 'tweedie', 'binomialP', 'binomialPQ', 'power',
        'geom_tweedie', 'poisson_tweedie']

        Parameters
        ----------
            mu : array_like
                A vector with expected values.
            power : float
                A power parameter.
            variance : str
                The variance function.
            z : list
                The list with z matrices for dependencies specification.
            power_fixed : boolean
                The specification of power estimation.
            Ntrial : int
                The number of trial. Parameter for Binomial distribution.
            omegas : list
                a list with omegas.
        """
        ...
```

O método `_generate_derivative_c()` calcula a derivada da matriz *C*. Esta operação faz uso de matrizes esparsas para aprimoramento de performance `https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html`.

No fim, a matriz *C* com suas derivadas é retornada como uma matrix `numpy` padrão. O primeiro passo é calcular as derivadas da Σ decomposta pela operação *Cholesky* através do método `_derivative_cholesky()`. É um cálculo iterativo, operacionalizado por cada matriz da derivada de Σ, gerando matrizes diagonais e finalizando com a multiplicação com a matriz Σ *Cholesky*. O pós-operatório envolve a construção de matrizes bloco diagonal da matriz *Cholesky*, implementado no método `_mc_transform_list_bdiag()`.

Listagem 5.15: Método para cálculo das derivadas Cholesky.

```
    def __derivative_cholesky(self, d_sigma, inv_chol, chol):
        ...
```

As derivadas da matriz *C* são calculadas pela operação sanduiche *power* entre as matrizes: sigma-entre-respostas, a matriz bloco diagonal derivada sigma *cholesky* e a matrix bloco diagonal Σ *cholesky*. Finalmente, para o caso de mais de uma resposta, calcula-se e adiciona-se a derivada dos `rho` à matriz *C* derivada com o método `_derivative_rho()`.

5.5 CLASSE MCGLMPARAMETERS

A abordagem frequentista leva a um comportamento assintótico gaussiano dos parâmetros. A classe `MCGLMParameters` calcula a matriz de variância-covariância, permitindo a

aferição das propriedades assintóticas dos estimadores e, por consequência, teste de hipótese e intervalo de confiança para cada parâmetro.

O principal método da classe é o `_parameters_attributes()`. O método usa parâmetros como: resíduos, derivadas dos valores esperados, matriz de pesos `W`, *score*, sensitividade e variabilidade da função *Quasi-score*, e os atributos da matriz C. Retorna a matriz de variância/covariância, sensitividade e variabilidade. Com esses componentes, é possivel delimitar os parâmetros resultantes da estimação.

Listagem 5.16: A classe `MCGLMParameters` implementa os cálculos das estimativas.

```
class MCGLMParameters:
    """According to MCGLM specification, grounded for frequentist inference
    traits, the estimation of resulting parameters converge asymptotically to
    a gaussian distribution with tuple mean-variance = (actual parameters,
    inverse of matrix Godambe). This property allows the calculation of
    pivotal traits regarding the parameters, such as: hypothesis testing and
    confidence interval.

    This class implements every method related to this trait.
    """

    @staticmethod
    def _parameters_attributes(
        resid,
        mu_deriv,
        W,
        quasi_score,
        mean_sens,
        mean_varia,
        c_inv,
        c,
        c_deriv,
        var_sensi,
    ):
        """The parameters of MCGLM converge assymtoptically to a Normal
        Distribution. This trait allows some statistical inferences as
        hypothesis testing, confidence interval and so on. This method crafts
        three important matrices: Varcov: variance covariance matrix,
        joint_inv_sensitivity: inverse sensitivity, and joint_variability: the
        joint variability distribution."""

        ...

    def _mc_cross_sensitivity(cov_product, columns_size):
        ...

    def generate_var_variability(res, w, c_inv, c_val, c_comp):
        ...

    def _calculate_variability(product, inv_C, C, res, W):
        ...

    def _mc_variability(sensitivity, we, k4, w):
        ...

    def _covprod(a, w, res):
        ...

```

```
49    def _mc_cross_variability(cov_product, inv_cw, res, d):
50        ...
```

5.6 CLASSE MCGLMRESULTS

Um ajuste de modelo bem-sucedido retorna um objeto da classe `MCGLMResults`, responsável por manipular e apresentar os parâmetros para a análise estatística. Visando replicar o consolidado padrão do statsmodels de relatório, a classe herda de `GLMResults` `https://www.statsmodels.org/dev/generated/statsmodels.genmod.generalized_linear_model.GLMResults.html`.

A classe `MCGLMResults` disponibiliza uma lista extensa de métodos: `aic`, `bic`, `loglikelihood`, `vcov`, `tvalues`, `pvalues`, `mu`, `pearson_residual`, `anova` e `summary`.

5.6.1 Métodos para avaliação de modelo

Os atributos `aic`, `bic` e `loglikelihood` possibilitam avaliação do ajuste pelas métricas *pseudo akaike information*, *pseudo bayesian information* e *pseudo loglikelihood*.

Listagem 5.17: Métodos principais da classe `MCGLMResults`.

```
1  def __p_log_likelihood(self, endog, mu, c_values, c_inverse):
2      """
3      Gaussian Pseudo-loglikelihood
4      """
5      ...
6
7  def __p_aic(self, pseudo_loglike, degrees_of_freedom):
8      """
9      pseudo Akaike Information criterion
10     """
11     ...
12
13 def __p_bic(self, pseudo_loglike, degrees_of_freedom, length_endogenous):
14     """
15     pseudo Bayesian Information criterion
16     """
17     ...
```

5.6.2 Atributos para testes de hipótese

O atributo `vcov` registra a matrix de variância/covariância calculada. `tvalues` apresentam a estatística `z` e o `pvalues`, os `p-values` da estatística `z`.

5.6.3 Método para análise da resposta esperada

O atributo `mu` retorna o vetor esperado ajustado pelo modelo. Os resíduos de *Pearson* gera o resíduo normalizado de *Pearson* para análise de resíduos.

Listagem 5.18: Método para cálculo dos resíduos de `Pearson`.

```
1  @property
2  def pearson_residuals(self):
3      """
```

```
    Pearson residuals. The Pearson residuals are defined as
    (`endog` - `mu`)/sqrt(VAR(`mu`)) where VAR is the distribution
    specific variance function. See statsmodels.families.family and
    statsmodels.families.varfuncs for more information.
    """
    ...
```

5.6.4 Relatório Sumário

Os modelos estatísticos disponíveis na biblioteca `statsmodels` geram o emblemático relatório sumário para compilação dos parâmetros estimados. Devido à sua padronização e clareza, a biblioteca `mcglm` implementa o mesmo modelo.

5.6.5 Métodos para matrizes de dependência

A biblioteca possui três métodos para auxiliar nas matrizes de dependência Z: `mc_id()`, `mc_mixed()` e `mc_ma()`. Esses métodos criam as matrizes dependência para os casos de dados independentes, modelos mistos e médias móveis. Todavia, o uso desses métodos não é obrigatório, e o usuário pode especificar as matrizes de forma particular.

6 SIMULAÇÕES COM A BIBLIOTECA MCGLM

Este capítulo apresenta sessões de simulação para aferir a fidedignidade da biblioteca `mcglm`. As funções de variância `BinomialP` e `tweedie` são validadas através de realizações dos modelos *Binomial*, *Gamma*, *Gaussiano Inverso* e *Compound Poisson-Gamma*, onde o parâmetro `power` também é estimado. Os scripts `Python` utilizados neste capítulo são apropriadamente especificados.

Seja um vetor resposta Y resultante de realizações independentes de um modelo pré-especificado, tendo o vetor μ como os parâmetros de média e o parâmetro de dispersão constante ϕ. Para estabelecer uma notação matemática, $Y_i \sim Tw_p(\mu_i, \phi)$ para a modelo *Tweedie*, $Y_i \sim B(10, \mu_i)$ para a modelo *Binomial* e $g(\mu_i) = \eta_i = x_i\beta$. Além disso, definimos três parâmetros de regressão β_0, β_1 e β_2, definidos em (2; 0,8; -1,5) para *Tweedie* e (0,8; 1; -1,5) para *Binomial*; duas covariáveis, a primeira é uma sequência entre -1 a 1, condicionada pelo tamanho amostral, e uma variável categórica de dois níveis, gerada aleatoriamente, com p definido em 0,5. O vetor μ é calculado por meio da operação linear entre covariáveis e parâmetros de regressão. Por fim, o parâmetro *power* é vital para o *Tweedie* emular sua distribuição semelhante a algum `EDM`, a família de dispersão exponencial; validamos quatro valores candidatos para o *power* (1,01; 1,5; 2; 3), que geram distribuições de probabilidade distintos. Para o caso *Binomial*, executamos a simulação para dez ensaios. Esta simulação avalia as estimativas pelo viés, consistência e taxa de cobertura, em diferentes tamanhos de amostra, para mil simulações.

Valida-se quatro tamanhos de amostra: 100, 250, 500 e 1000; três níveis de variância para cada tamanho amostral, gerados usando coeficiente de variação igual a 15%, 50% e 80%, respectivamente. Portanto, para *power* definido em 1,01, os parâmetros de dispersão são (1,5; 15; 40); para *power* definido em 1,5, os parâmetros de dispersão são (0,2; 2; 5,5); para *power* definido em 2, os parâmetros de dispersão são (0,023; 0,25; 0,65); para *power* definido em 3, os parâmetros de dispersão são (0,0003; 0,0034; 0,0083). No caso *Binomial*, analisamos a autenticidade do parâmetro de dispersão ao valor 1.

Realizamos simulações de *Tweedie* para valores de potência 1,01 e 1,5, com auxílio da biblioteca `tweedie`; para os valores de potência 2 e 3, utilizamos a biblioteca `scipy`. Finalmente, para *Binomial*, a biblioteca `scipy`. Para todas as simulações, pretendemos validar se o parâmetro de dispersão ϕ, e os parâmetros de regressão correspondem aos valores finais. As urls são, para o `tweedie`, a `https://pypi.org/project/tweedie/` e ao `scipy`, a `https://scipy.org/`.

O script `Python` a seguir cria as simulações a partir dos parâmetros `n_sim` como número de simulações; `sample_size` como tamanho da amostra; `dispersion` como o parâmetro de dispersão; `power` como o parâmetro de potência. As simulações *Tweedie* e *Binomial* têm seus próprios grupos de parâmetros para definir o vetor μ e a configuração do modelo `mcglm`. O método `get_link` especifica e implementa completamente.

Listagem 6.1: Script `Python` gerador das simulações.

```
1 def generate_model_simulation(
2    n_sim: int,
3    sample_size: int,
4    dispersion: float,
5    power: float,
6    covariate1 : np.array,
7    covariate2 : np.array,
8    beta_0 : float,
```

```
    beta_1 : float,
    beta_2 : float,
    model_type : str,
    power_fixed : bool = True
) -> dict:
    """
    The method generate_model_simulation runs out simulations for a specific
    configuration of model and parameters.
    """
    def model_simulation(model_type, mu, dispersion, sample_size, power=None):
        """
        Base method to trigger a random variable realization of sample size
        as "sample_size"; expected value as "mu"; dispersion as "dispersion"
        and model type as "model_type".
        """
        if model_type == "tweedie":
            return tweedie.tweedie(mu=mu, p=power, phi=dispersion)
                      .rvs(sample_size)
        elif model_type == "gamma":
            shape = 1 / dispersion
            scale_g = mu * dispersion
            return gamma.rvs(shape, scale=scale_g, size=sample_size)
        elif model_type == "invgauss":
            mu_ig = mu * dispersion
            return invgauss.rvs(mu_ig, scale= 1 /dispersion, size=sample_size)
        elif model_type == "binomial":
            return binom.rvs(p=mu, size=sample_size)


    def get_link(model_type : str):
        if model_type == "binomial":
            return {"object_link": Logit(),
                    "link": "logit",
                    "variance": "binomialP"}
        else:
            return {"object_link": Log(),
                    "link": "log",
                    "variance": "tweedie"}


    historical_data = {"beta0": [],
                       "beta1": [],
                       "beta2": [],
                       "dispersion": [],
                       "beta0_confidence_interval": [],
                       "beta1_confidence_interval": [],
                       "beta2_confidence_interval":[],
                       "dispersion_confidence_interval":[],
                       "power_confidence_interval":[],
                       "power": []}
    model_conf = get_link(model_type)


    eta = beta_0 + beta_1 * covariate1 + beta_2 * covariate2
    mu = model_conf["object_link"].inverse(eta)

    for _ in range(n_sim):

```

```
        y = model_simulation(model_type=model_type,
                             mu=mu,
                             dispersion=dispersion,
                             sample_size=sample_size,
                             power=power)

        data = pd.DataFrame({"y": y, "cov1": covariate1, "cov2": covariate2})

        data["cov2"] = data["cov2"].astype("str")
        X = dmatrix("~ cov1 + cov2", data, return_type="dataframe")

        # Z specification
        Z = [mc_id(data)]

        # Model fitting
        mcglm = MCGLM(
            endog=data["y"],
            exog=X,
            z=Z,
            link=model_conf["link"],
            variance=model_conf["variance"],
            power=power if power_fixed else 2.5,
            power_fixed=power_fixed,
            ntrial=1,
        )

        mcglmresults = mcglm.fit()

        beta_0_cinf, beta_2_cinf, beta_1_cinf, dispersion_cinf, power_cinf =
        parser_estimates(mcglmresults.summary().as_csv())

        historical_data["beta0"].append(beta_0_cinf[0])
        historical_data["beta1"].append(beta_1_cinf[0])
        historical_data["beta2"].append(beta_2_cinf[0])
        historical_data["dispersion"].append(dispersion_cinf[0])
        historical_data["power"].append(power_cinf[0])

        historical_data["beta0_confidence_interval"].append(
            (beta_0_cinf[1] <= beta_0) & (beta_0_cinf[2] >= beta_0))
        historical_data["beta1_confidence_interval"].append(
            (beta_1_cinf[1] <= beta_1) & (beta_1_cinf[2] >= beta_1))
        historical_data["beta2_confidence_interval"].append(
            (beta_2_cinf[1] <= beta_2) & (beta_2_cinf[2] >= beta_2))
        historical_data["dispersion_confidence_interval"].append(
            (dispersion_cinf[1] <= dispersion) & (dispersion_cinf[2] >=
            dispersion))
        historical_data["power_confidence_interval"].append(
            (power_cinf[1] <= power) & (power_cinf[2] >= power))

    return historical_data
```

A seguir, o método que orquestra as simulações nas condições estabelecidas.

Listagem 6.2: Script `Python` orquestrador das simulações.

```
betas_tweedie = {"beta_0": 2, "beta_1": 0.8, "beta_2": -1.5}
betas_logit = {"beta_0": 1, "beta_1": 0.7, "beta_2": -1.8}

n_sim = 1000
```

```
5
6  def dump_json(json_name_file : str, log_simulation : dict):
7     loc_file = "log/" + json_name_file + ".json"
8     with open(loc_file, "w") as fh:
9        json.dump(log_simulation, fh)
10
11 for simulation_conf in [
12       {"power": 1.01,
13       "dispersion":[1.5, 15, 40],
14       "model_type":"tweedie",
15       "regression_parameter": betas_tweedie},
16       {"power": 1.5,
17       "dispersion":[0.2, 2, 5.5],
18       "model_type": "tweedie",
19       "regression_parameter": betas_tweedie},
20       {"power": 2,
21       "dispersion":[0.023, 0.25, 0.65],
22       "model_type": "gamma",
23       "regression_parameter": betas_tweedie},
24       {"power": 3,
25       "dispersion":[0.0003, 0.0034, 0.0083],
26       "model_type": "invgauss",
27       "regression_parameter": betas_tweedie},
28       {"power": 1,
29       "dispersion":[1],
30       "model_type": "binomial",
31       "regression_parameter": betas_logit}
32       ]:
33    regression_parameters = simulation_conf["regression_parameter"]
34    for dispersion in simulation_conf["dispersion"]:
35       report = "- Simulation report - "
36       for sample_size in [250, 500, 750, 1000]:
37
38          covariate1 = np.arange(
39             -1.0, 1.0, 1 / (sample_size / 2)
40          )
41          covariate2 = np.random.choice([0, 1], size=sample_size)
42
43
44          simulation = {"n_sim": n_sim,
45                   "sample_size": sample_size,
46                   "dispersion": dispersion,
47                   "power": simulation_conf["power"],
48                   "covariate1": covariate1,
49                   "covariate2": covariate2,
50                   "beta_0":regression_parameters["beta_0"],
51                   "beta_1": regression_parameters["beta_1"],
52                   "beta_2": regression_parameters["beta_2"],
53                   "model_type": simulation_conf["model_type"]}
54
55          log_simulation = generate_model_simulation(**simulation)
56
57          dump_json("power_"+ str(simulation_conf["power"]) +
58                  "__" + "n_sim_" + str(n_sim) +
59                  "__" + "dispersion_" + str(dispersion) +
60                  "__" + "sample_size_" + str(sample_size), log_simulation)
61
62          histogram_log_simulation(log_simulation,
```

```
                                power=simulation_conf["power"],
                                sample_size=sample_size,
                                dispersion=dispersion,
                                beta_0=regression_parameters["beta_0"],
                                beta_1=regression_parameters["beta_1"],
                                beta_2=regression_parameters["beta_2"]
                               )
        report += "\n" + coverage_analysis(log_simulation,
              dispersion, sample_size,
              n_sim, simulation_conf["power"])
```

6.1 BINOMIAL

A Figura 6.1 apresenta as estimativas resultantes de mil simulações para o modelo *Binomial*. Esta seção apresenta a convergência dos parâmetros de regressão e dispersão via os graficos do tipo *boxplot*, e a analise de cobertura dos parâmetros via gráficos de linha.

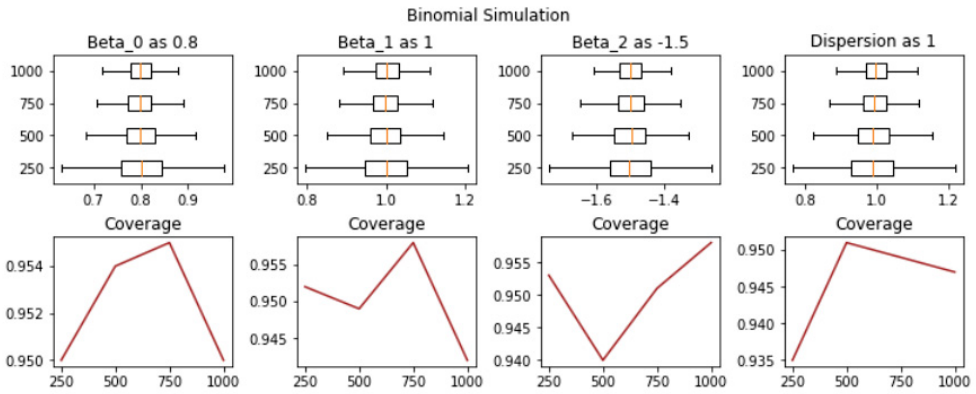


Figura 6.1: Resultado da simulação `Binomial`

É possível verificar a consistente e progressiva convergência aos parâmetros reais, sob o aumento do tamanho amostral, e a cobertura satisfatória dos intervalos de confiança, ao redor 95% para a maioria das estimativas.

6.2 TWEEDIE E POWER DEFINIDO EM 1,01

As Figuras 6.2, 6.3 e 6.4 apresentam as estimativas resultantes das três sessões de mil simulações, com parâmetros de dispersão distintos, e o *power* inicial em 1,01. Esta seção apresenta a convergência dos parâmetros de regressão e dispersão via os graficos do tipo *boxplot*, e a analise de cobertura dos parâmetros via gráficos de linha.

É possível verificar a consistente convergência aos parâmetros reais, inclusive o *power*, e a ótima cobertura dos intervalos de confiança, ao redor 95% para a maioria das estimativas.

6.3 TWEEDIE E POWER DEFINIDO EM 1,5

As Figuras 6.5, 6.6 e 6.7 apresentam as estimativas resultantes das três sessões de mil simulações, com parâmetros de dispersão distintos, e o *power* inicial em 1,5. Persiste-se o padrão de relatório de simulações.

Novamente, é possível verificar a consistente convergência aos parâmetros reais e a alta cobertura dos intervalos de confiança, ao redor 95% para a maioria das estimativas.

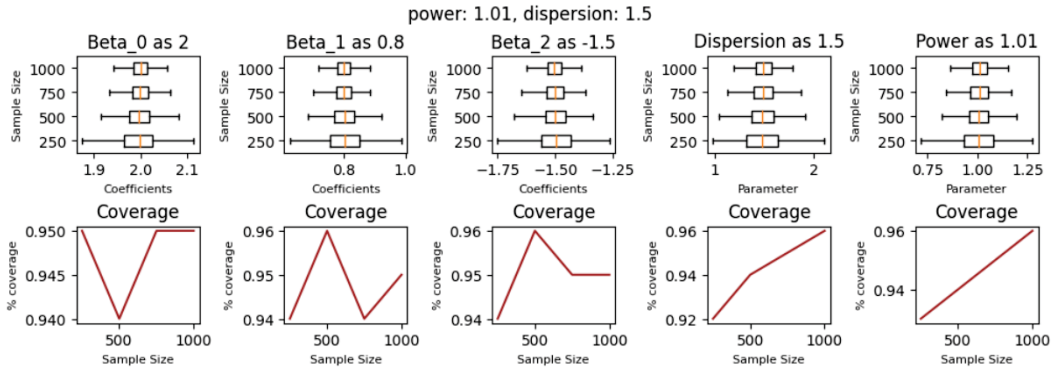


Figura 6.2: Resultado da simulação `Tweedie` com *power* 1,01 e dispersão 1,5.

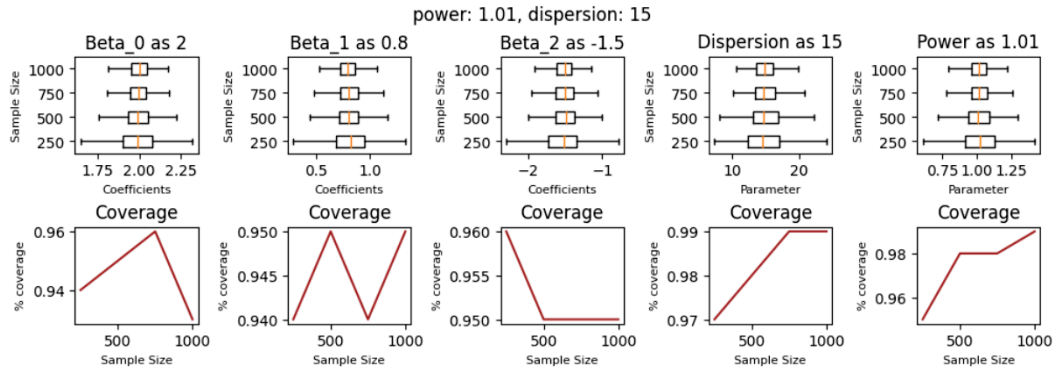


Figura 6.3: Resultado da simulação `Tweedie` com *power* 1,01 e dispersão 15.

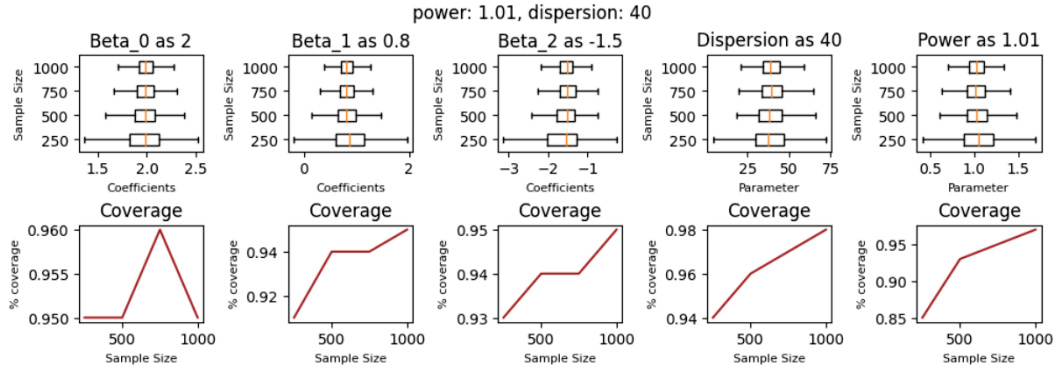


Figura 6.4: Resultado da simulação `Tweedie` com *power* 1,01 e dispersão 40.

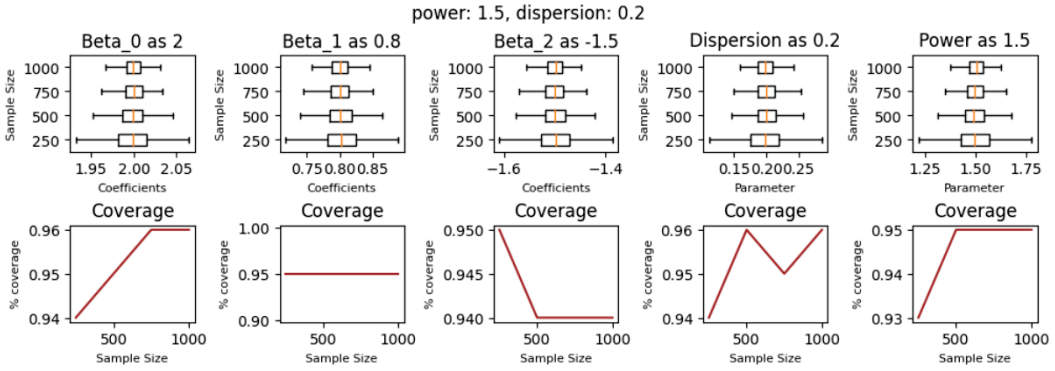


Figura 6.5: Resultado da simulação `Tweedie` com `power` 1,5 e dispersão 0,2.

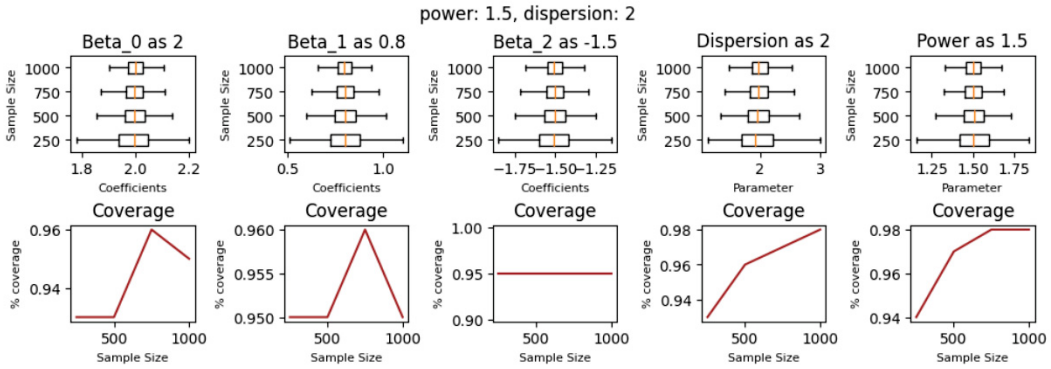


Figura 6.6: Resultado da simulação `Tweedie` com `power` 1,5 e dispersão 1,8.

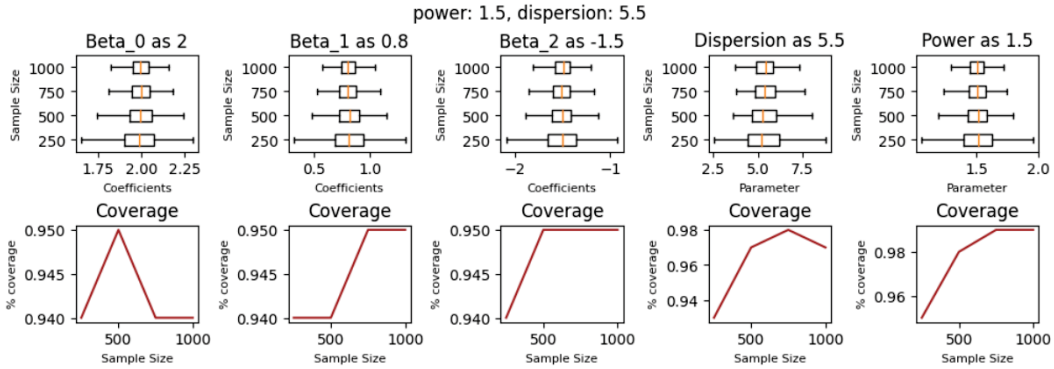


Figura 6.7: Resultado da simulação `Tweedie` com `power` 1,5 e dispersão 5,2.

6.4 TWEEDIE E POWER DEFINIDO EM 2

As Figuras 6.8, 6.9 e 6.10 apresentam as estimativas resultantes das três sessões de mil simulações, com parâmetros de dispersão distintos, e o *power* inicial em 2. Persiste-se o padrão de relatório de simulações.

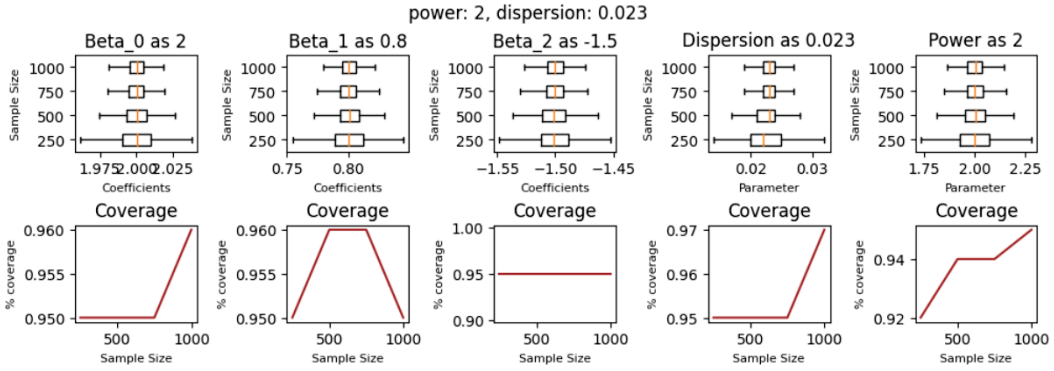


Figura 6.8: Resultado da simulação `Tweedie` com `power` 2 e dispersão 0,023.

É possível verificar a consistente convergência aos paramêtros reais e a ótima cobertura dos intervalos de confiança, ao redor 95% para a maioria das estimativas.

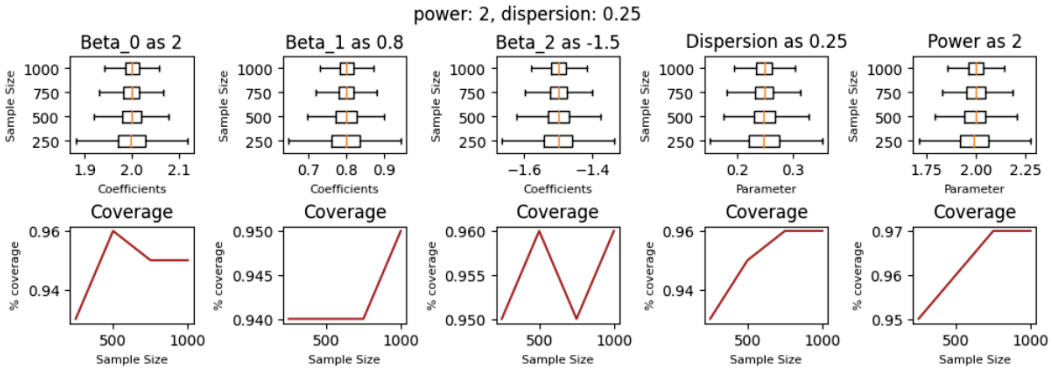


Figura 6.9: Resultado da simulação `Tweedie` com `power` 2 e dispersão 0,25.

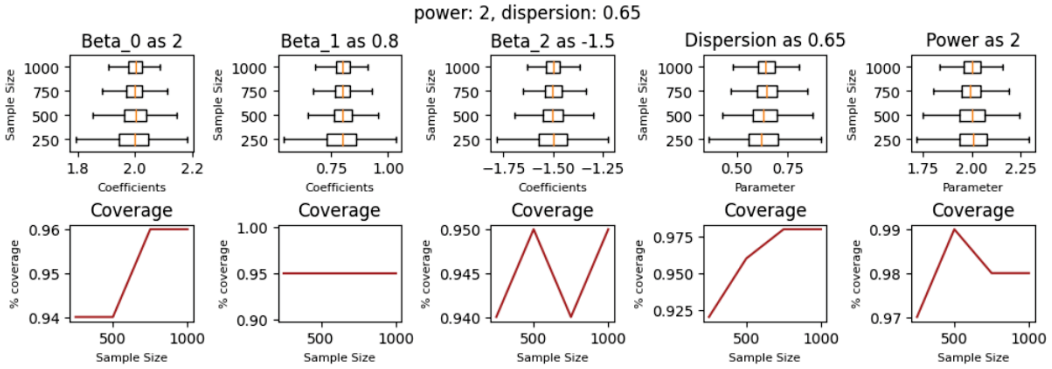


Figura 6.10: Resultado da simulação `Tweedie` com `power` 2 e dispersão 0,65.

6.5 TWEEDIE E POWER DEFINIDO EM 3

As Figuras 6.11, 6.12 e 6.13 apresentam as estimativas resultantes das três sessões de mil simulações, com parâmetros de dispersão distintos, e o *power* inicial em 3. Persiste-se o padrão de relatório de simulações.

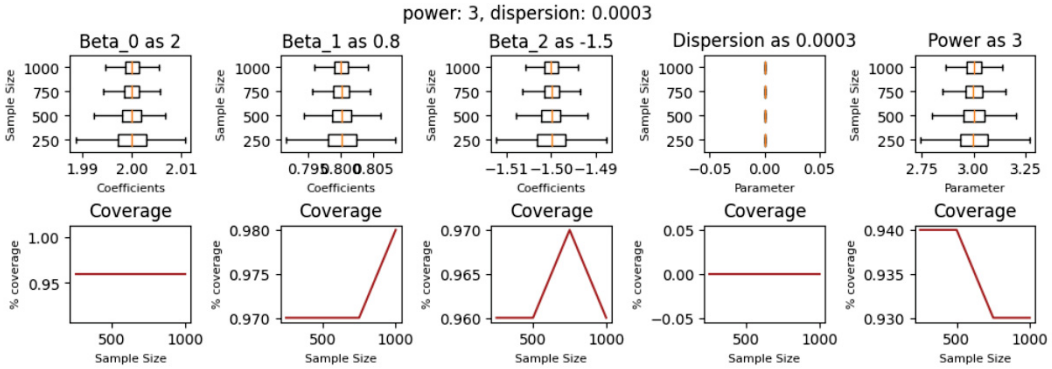


Figura 6.11: Resultado da simulação `Tweedie` com `power` 3 e dispersão 0,0003.

Os parâmetros de regressão e *power* se aproximam para todos os valores de dispersão. Esta converge no cenário alto de variância, entretanto, para os dois menores valores de dispersão, apesar das estimativas próximas, os valores não capitulam ao intervalo de confiança. A menor variância reside na borda, incorrendo no efeito de borda.

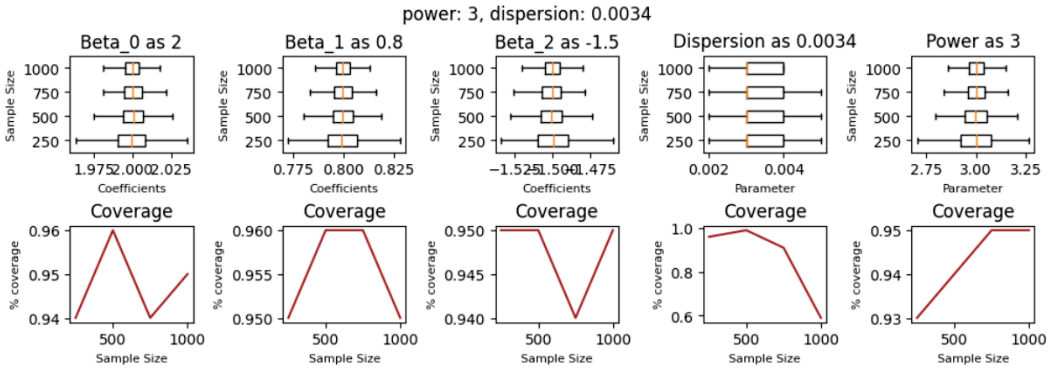


Figura 6.12: Resultado da simulação `Tweedie` com `power` 3 e dispersão 0,0034.

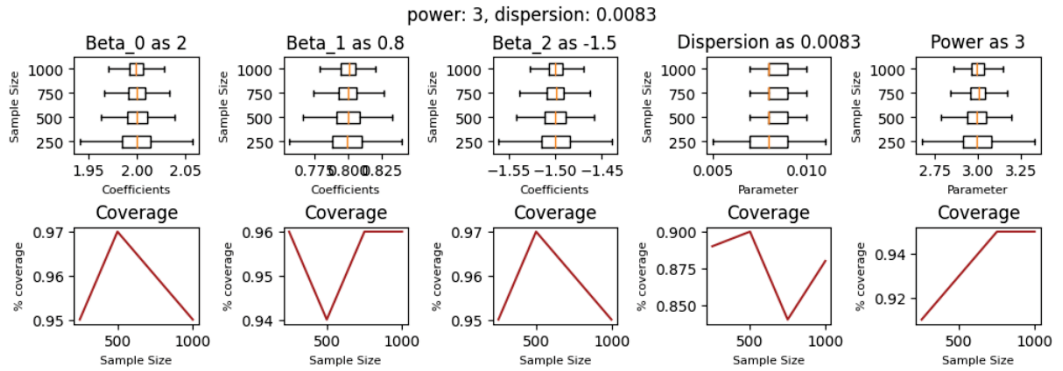


Figura 6.13: Resultado da simulação `Tweedie` com `power` 3 e dispersão 0,0083.

Replicamos as simulações com o parâmetro *power* fixo; foi possível observar a ausência de vies e consistência nos coeficientes de regressão e parâmetros de dispersão, à medida o aumento do tamanho amostral.

7 ANÁLISE ESTATÍSTICA COM A BIBLIOTECA MCGLM

Esta seção apresenta a aplicação prática do MCGLM em duas análises com a biblioteca `mcglm`, objeto desta dissertação. Os exemplos a seguir fazem alusão à flexibilidade do framework estatístico e à acessível interface do modelo `mcglm`.

7.1 ANÁLISE DE PRIVAÇÃO DE SONO

O conjunto de dados sleepstudy é um estudo de privação de sono. `https://www.rdocumentation.org/packages/lme4/versions/1.1-26/topics/sleepstudy`.

O estudo visa analisar a influência da privação de sono no tempo de reação geral de um grupo de indivíduos. No dia zero, os pacientes tiveram sua quantidade habitual de sono. A partir desse dia, eles foram restringidos a três horas de sono por noite. As observações representam o tempo médio diário de reação em uma bateria individual de testes.

Uma abordagem promissora é o ajuste por meio de modelos de efeito de dois níveis, como modelos mistos. A biblioteca `mcglm` é adequada para conduzir essa análise estatística. O trecho de código abaixo importa a biblioteca `mcglm`, bem como `pandas`, `matplotlib`, `seaborn` e `patsy` para construir as matrizes de delineamento.

Listagem 7.1: Primeira parte do código com a importação das bibliotecas.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from mcglm import MCGLM, mc_mixed, mc_id
from patsy import dmatrix
```

O trecho abaixo produz um gráfico de todos os indivíduos ao longo dos dias de estudo.

Listagem 7.2: Leitura de dados com o `pandas` e exibição gráfica dos dados com `seaborn`.

```
sleepstudy = pd.read_csv('sleepstudy.csv')

sns.lineplot(x='Days', y='Reaction', hue='Subject', data=sleepstudy)
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
```

O gráfico resultante da geração da visualização é apresentado na Figura 7.1.

Pela Figura 7.1, os indivíduos sob análise se comportam de formas distintas. Alguns usuários sentem o efeito da privação do sono, diferentemente de outros. Entretanto, em linhas gerais, há acréscimo no tempo de resposta na adição de dias em privação; os dias de privação parecem influenciar no tempo de reação.

Avançando na análise estatística, prosseguimos para a construção da matriz de delineamento e matrizes de dependências:

Listagem 7.3: Construção das matrizes de delineamento e dependência.

```
X = dmatrix('~ Days', sleepstudy, return_type="dataframe")
Z = [mc_id(sleepstudy)] + mc_mixed(formula='~ 0 + Subject / Days',
data = sleepstudy)
```

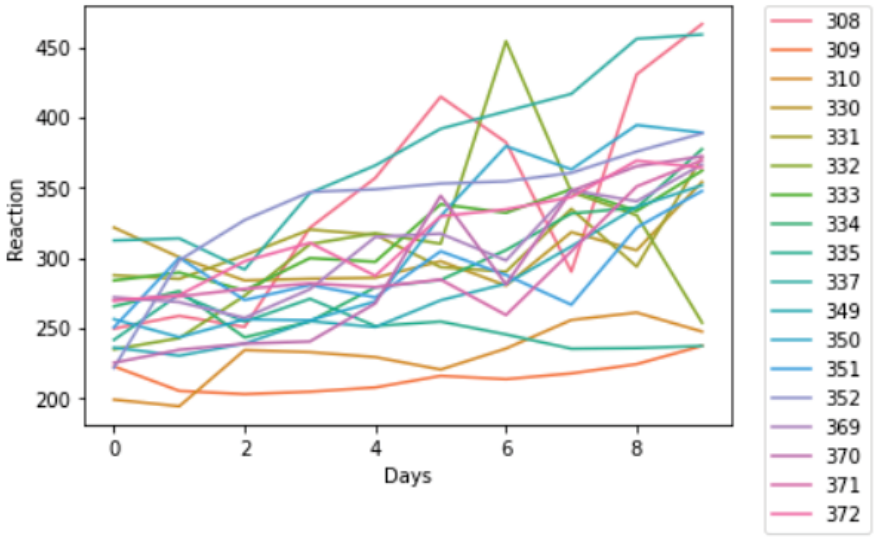


Figura 7.1: Tempo de reação dos indivíduos ao longo dos dias

Enfim, podemos fazer o ajuste do modelo MCGLM com a biblioteca `mcglm`. Na instanciação do objeto, a variável resposta é passada como o atributo `endog`, a matriz de delineamento como o atributo `exog` e as matrizes de dependência `z`. Como os parâmetros `link` e `variance` não foram definidos, a biblioteca aplica os valores padrão: `identity` e `constant`.

A formula especificada na construção das matrizes `Z` emula o `mcglm` como um modelo misto de *intercepts* e *slopes* aleatórios.

Listagem 7.4: Ajuste do modelo MCGLM.

```
mcglm = MCGLM(
    endog=sleepstudy['Reaction'],
    exog=X,
    z=Z
)
mcglmresults = mcglm.fit()
```

O método sumário apresenta as estimativas resultantes do ajustes do modelo.

Listagem 7.5: Relatório sumário do MCGLM.

```
            Multivariate Covariance Generalized Linear Model
==============================================================================
Dep. Variable:     Reaction       No. Iterations:    20
Model:             MCGLM          No. Observations:  180
link:              identity       Df Residuals:      174
variance:          constant       Df Model:          6
Method:     Quasi-Likelihood      Power-fixed:       True
Date:       Thu, 30 Jun 2022      pAIC:              1583.94
Time:              14:49:00       pBIC:              1603.1
                                  pLogLik:           -785.9704
==============================================================================
             coef     std err   z      P>|z|    [0.025   0.975]
------------------------------------------------------------------------------
Intercept 251.4051  6.632    37.906  0.000   238.406 264.404
Days      10.4673   1.502    6.968   0.000   7.523   13.412
==============================================================================
```

```
                coef      std err    z      P>|z|    [0.025  0.975]
------------------------------------------------------------------------------
dispersion_1 654.9420   70.624   9.274   0.000    516.521 793.363
dispersion_2 565.5150   264.67   2.137   0.033    46.753  1084.277
dispersion_3 32.6820    13.560   2.410   0.016    6.105   59.259
dispersion_4 11.0550    42.948   0.257   0.797   -73.121 95.231
==============================================================================
```

O cabeçalho do relatório traz informações sobre parâmetros do mcglm, características do modelo e medidas de ajuste. Os valores padrão para função de ligação e variância são `identity` e `constant`, o que neste caso, faz com que o algoritmo se comporte como um modelo misto gaussiano. Além disso, as medidas de adequação `pAIC`, `pBIC` e `pLogLik` são boas informações para reter: 1583,94, 1603,1 e -785,97.

A seção com os parâmetros de regressão reporta os parâmetros de efeitos fixo. O *intercept* definido em valor 251,40 alude ao tempo médio de resposta no dia 0, em segundos. Além disso, o valor `Days` em 10,46 alude à alteração total do tempo de resposta a cada mudança unitária em Dias, para sujeitos com efeitos aleatório definido em 0. Além disso, pelo `p-value` resultante do teste de `Wald`, podemos concluir que os dias são estatisticamente significativos no modelo estatístico para a análise do tempo de reação.

Os parâmetros de dispersão descrevem os efeitos aleatórios dentro dos indivíduos ao longo dos dias. O primeiro parâmetro de dispersão é a variação envolvendo todos os pontos, sua estimativa em 654,94. O segundo parâmetro de dispersão é a variação entre todos os intercepts, 565,51. O terceiro parâmetro de dispersão é a variância das inclinações, calculado em 32,68. O quarto parâmetro de dispersão é a covariância entre intercepts e inclinações, calculado em 11,05.

Além disso, a correlação pode ser calculada através da fórmula simples: a quarta dispersão dividida pela raiz quadrada da multiplicação entre a segunda e a terceira. A correlação é de 0,0812.

A análise residual é uma maneira de avaliar a qualidade do ajuste. O código abaixo aproveita o atributo `pearson_residuals` para a análise.

Listagem 7.6: Análise residual com a biblioteca `mcglm`.

```
plt.scatter(mcglmresults.mu, mcglmresults.pearson_residuals)
plt.xlabel('mu')
plt.ylabel('pearson residuals')
```

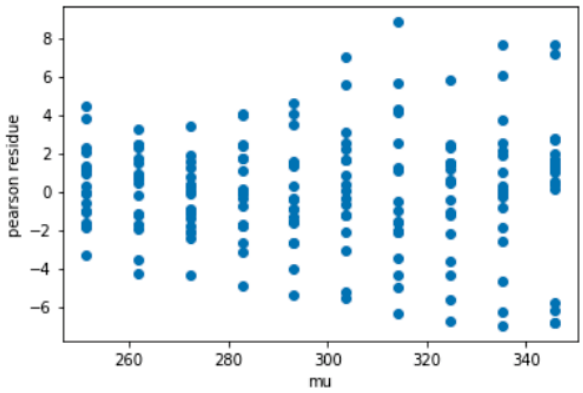


Figura 7.2: Diagrama de dispersão dos valores esperados e os resíduos de Pearson.

Os resultados da Figure 7.2 revela residuos próximos a homogeneidade.

7.2 ANÁLISE DE TRATAMENTO DE SOJA

`Soya` é um conjunto de dados que registra um experimento da Universidade Federal da Grande Dourados, em Dourados, Mato Grosso do Sul, Brasil. O experimento coletou dados sobre diferentes tratamentos de `potássio`, `água` e `blocos`, todas variáveis categóricas, para cultivo em solo controlado. As variáveis resposta a serem analisadas são: tamanho do grão, total de sementes e percentual de vagens viáveis. A seguir, o MCGLM analisa essas três respostas em um ajuste, demonstrando a flexibilidade do modelo em três tipos diferentes de respostas: contínua, contagem e limitados. O trecho de código para ler os dados:

Listagem 7.7: Leitura da base de dados com o `pandas` e criação da variável resposta.

```
soya = pd.read_csv('soya.csv')

soya['viablePeas'] = soya['viablepeas'] / soya['totalpeas']
```

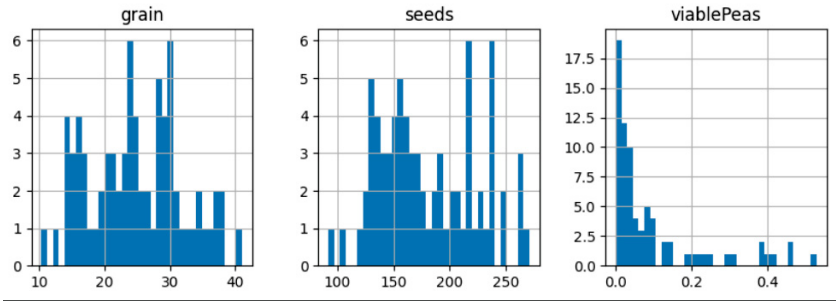


Figura 7.3: Histogramas das variáveis resposta

Os histogramas da Figura 7.3 ajudam na análise geral de distribuição das variáveis resposta.

As variáveis de análise `grain`, `seeds` e `viablePeas` são contínuas, contagens e limitadas/binomiais, respectivamente. Três modelos de probabilidade sugeridos são: `Gaussiano`, `Poisson` e `Binomial`, conforme a especificação do modelo. Para tal, as escolhas para a função *link* são: `identity`, `log` e `logit`; para a função *variance* são: `constant`, `tweedie` e `binomialP`. Finalmente, o parâmetro `ntrial` recebe uma lista preenchida com dois valores vazio e o total de vagens.

Listagem 7.8: Seção de ajuste das três variáveis resposta.

```
mcglm = MCGLM(
    endog=[soya['grain'], soya['seeds'], soya['viablepeasP']],
    exog=[X, X, X],
    z=[[mc_id(soya)], [mc_id(soya)], [mc_id(soya)]],
    link=['identity', 'log', 'logit'],
    variance=['constant', 'tweedie', 'binomialP'],
    ntrial=[None, None, soya['totalpeas'].values]
)
mcglmresults = mcglm.fit()
```

O objeto retornado pode construir o relatório `summary()` com as três análises das respostas e uma análise de correlação entre as respostas. Por legibilidade, essa análise descreve cada resposta separadamente.

A resposta do grão foi ajustada com os parâmetros padrão, `identity` e `constant`, similar à uma regressão linear. Entre as covariáveis bloco, água e potássio, pelo teste de *Wald*,

os níveis de potássio são estatisticamente significantes para o modelo; quanto maior o nível de potássio, maior o tamanho do grão. A variação é constante em 5,862.

Listagem 7.9: Sumário do treinamento para a variável tamanho de grãos.

```
   Multivariate Covariance Generalized Linear Model
==============================================================================
Dep. Variable:    grain          No. Iterations:    10
Model:            MCGLM          No. Observations:  75
link:             identity       Df Residuals:      55
variance:         constant       Df Model:          20
Method:     Quasi-Likelihood     Power-fixed:       True
Date:       Fri, 14 Oct 2022     pAIC:              448.21
Time:             09:45:15       pBIC:              516.53
                                 pLogLik:          -204.1064
 ============================================================================
                         coef  std err      z    P>|z|   [0.025   0.975]
 ----------------------------------------------------------------------------
 Intercept              14.2456   1.218   11.699   0.000   11.859   16.632
 block[T.II]             1.1324   0.880    1.286   0.198   -0.593    2.858
 block[T.III]           -0.7801   0.880   -0.886   0.376   -2.506    0.946
 block[T.IV]            -1.5492   0.880   -1.760   0.078   -3.275    0.176
 block[T.V]             -2.3871   0.880   -2.711   0.007   -4.113   -0.661
 water[T.50]             2.1661   1.531    1.415   0.157   -0.835    5.167
 water[T.62.5]           2.5404   1.531    1.659   0.097   -0.460    5.541
 pot[T.120]             11.7898   1.531    7.701   0.000    8.789   14.790
 pot[T.180]             11.8633   1.531    7.749   0.000    8.863   14.864
 pot[T.30]               6.7895   1.531    4.435   0.000    3.789    9.790
 pot[T.60]              10.3978   1.531    6.792   0.000    7.397   13.399
 water[T.50]:pot[T.120] 2.2543   2.165    1.041   0.298   -1.989    6.498
 water[T.62.5]:pot[T.120] 5.5828 2.165    2.578   0.010    1.339    9.826
 water[T.50]:pot[T.180] 1.2096   2.165    0.559   0.576   -3.034    5.453
 water[T.62.5]:pot[T.180] 9.2389 2.165    4.267   0.000    4.995   13.483
 water[T.50]:pot[T.30]   0.1067   2.165    0.049   0.961   -4.137    4.350
 water[T.62.5]:pot[T.30] -1.8678 2.165   -0.863   0.388   -6.112    2.376
 water[T.50]:pot[T.60]   2.6034   2.165    1.202   0.229   -1.640    6.847
 water[T.62.5]:pot[T.60] 3.3586 2.165   1.551   0.121   -0.885    7.602
 ============================================================================
                coef  std err      z    P>|z|   [0.025   0.975]
 ----------------------------------------------------------------------------
 dispersion_1 5.8620   1.671    3.508   0.000    2.587    9.137
==============================================================================
```

A análise residual da Figura 7.4 revela um bom ajuste geral.

A covariável `semente` é um variável de contagem. Uma maneira direta e canônica de modelá-la é via *Poisson*. Duas, das três covariáveis, são estatisticamente significativas na semente: potássio, para todos os níveis; e bloco para o nível `V`.

Listagem 7.10: Sumário do treinamento para a variável contagem de grãos.

```
   Multivariate Covariance Generalized Linear Model
==============================================================================
Dep. Variable:    seeds          No. Iterations:    10
Model:            MCGLM          No. Observations:  75
link:             log            Df Residuals:      55
variance:         tweedie        Df Model:          20
Method:     Quasi-Likelihood     Power-fixed:       True
Date:       Fri, 14 Oct 2022     pAIC:              448.21
Time:             09:45:15       pBIC:              516.53
```

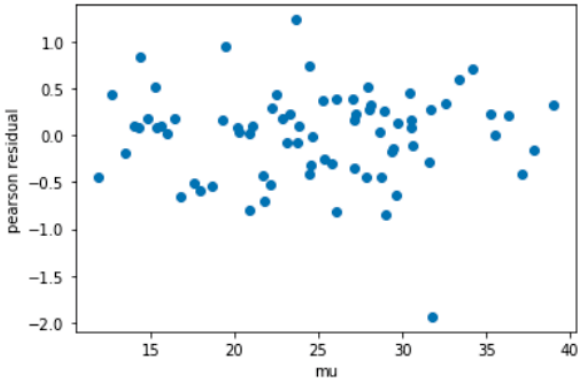


Figura 7.4: Resíduos de `Pearson` para o peso dos grãos

```
                                pLogLik:          -204.1064
==============================================================================
                     coef   std err      z    P>|z|    [0.025    0.975]
------------------------------------------------------------------------------
Intercept             4.8074    0.066   73.099    0.000     4.679     4.936
block[T.II]          -0.0182    0.039   -0.469    0.639    -0.094     0.058
block[T.III]         -0.0330    0.039   -0.845    0.398    -0.110     0.044
block[T.IV]          -0.1066    0.040   -2.678    0.007    -0.185    -0.029
block[T.V]           -0.1262    0.040   -3.154    0.002    -0.205    -0.048
water[T.50]           0.1322    0.084    1.579    0.114    -0.032     0.296
water[T.62.5]         0.1864    0.083    2.255    0.024     0.024     0.348
pot[T.120]            0.3672    0.079    4.620    0.000     0.211     0.523
pot[T.180]            0.2945    0.081    3.649    0.000     0.136     0.453
pot[T.30]             0.2980    0.081    3.695    0.000     0.140     0.456
pot[T.60]             0.3444    0.080    4.312    0.000     0.188     0.501
water[T.50]:pot[T.120] 0.1159  0.108    1.076    0.282    -0.095     0.327
water[T.62.5]:pot[T.120] 0.0698 0.107  0.653    0.514    -0.140     0.279
water[T.50]:pot[T.180] 0.2905  0.108    2.697    0.007     0.079     0.502
water[T.62.5]:pot[T.180] 0.2166 0.107  2.022    0.043     0.007     0.426
water[T.50]:pot[T.30] 0.0425   0.110    0.386    0.699    -0.173     0.258
water[T.62.5]:pot[T.30] -0.1372 0.111 -1.241    0.215    -0.354     0.080
water[T.50]:pot[T.60] 0.1161   0.108    1.073    0.283    -0.096     0.328
water[T.62.5]:pot[T.60] 0.0902 0.107   0.842    0.400    -0.120     0.300
==============================================================================
                coef   std err      z    P>|z|    [0.025    0.975]
------------------------------------------------------------------------------
dispersion_1 2.1620    0.429    5.035    0.000     1.320     3.004
==============================================================================
```

A análise residual da Figura 7.5 revela um bom ajuste.

A covariável `ViablePeas` é limitada, com `N` ensaios. Entre as covariáveis, apenas o nível de potássio 30 e o nível de água 50 são estatisticamente significativos ao modelo.

Listagem 7.11: Sumário do treinamento para a variável percentual de vagens viáveis.

```
   Multivariate Covariance Generalized Linear Model
==============================================================================
Dep. Variable: viablePeas       No. Iterations:    10
Model:              MCGLM       No. Observations:  75
link:               logit       Df Residuals:      55
variance:           binomialP   Df Model:          20
Method:     Quasi-Likelihood    Power-fixed:       True
Date:       Fri, 14 Oct 2022    pAIC:              448.21
```

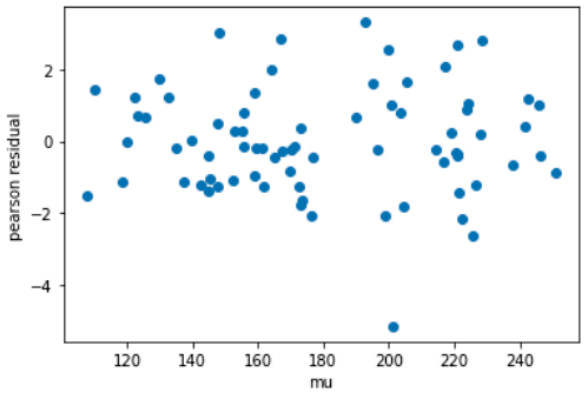


Figura 7.5: Pearson residuals para a variável contagem de grãos.

```
Time:                09:45:15       pBIC:                 516.53
                                    pLogLik:             -204.1064
==============================================================================
                         coef  std err      z    P>|z|   [0.025   0.975]
------------------------------------------------------------------------------
Intercept             -0.8360   0.197  -4.242   0.000   -1.222   -0.450
block[T.II]           -0.4383   0.188  -2.329   0.020   -0.807   -0.070
block[T.III]          -0.1671   0.183  -0.914   0.361   -0.525    0.191
block[T.IV]           -0.1877   0.183  -1.023   0.306   -0.547    0.172
block[T.V]            -0.2926   0.187  -1.564   0.118   -0.659    0.074
water[T.50]            0.5675   0.218   2.604   0.009    0.140    0.995
water[T.62.5]          0.2414   0.219   1.102   0.271   -0.188    0.671
pot[T.120]            -3.3915   0.597  -5.682   0.000   -4.561   -2.222
pot[T.180]            -3.6309   0.685  -5.300   0.000   -4.974   -2.288
pot[T.30]             -1.8455   0.329  -5.601   0.000   -2.491   -1.200
pot[T.60]             -3.1518   0.541  -5.826   0.000   -4.212   -2.091
water[T.50]:pot[T.120] 0.6881  0.672   1.024   0.306   -0.629    2.005
water[T.62.5]:pot[T.120] -0.8358 0.910 -0.918  0.358   -2.620    0.948
water[T.50]:pot[T.180] 1.2730  0.735   1.732   0.083   -0.167    2.713
water[T.62.5]:pot[T.180] 0.0683 0.842  0.081   0.935   -1.581    1.718
water[T.50]:pot[T.30] -0.2773  0.425  -0.653   0.514   -1.110    0.555
water[T.62.5]:pot[T.30] 0.8046 0.400   2.013   0.044    0.021    1.588
water[T.50]:pot[T.60] 0.4598   0.622   0.739   0.460   -0.760    1.680
water[T.62.5]:pot[T.60] 0.6312 0.630   1.002   0.316   -0.604    1.866
==============================================================================
                coef  std err      z    P>|z|   [0.025   0.975]
------------------------------------------------------------------------------
dispersion_1 1.2970   0.245   5.293   0.000    0.817    1.777
==============================================================================
```

A Figura 7.6 apresenta os resíduos de *Pearson* do ajuste *binomial*.

Pela biblioteca `mcglm`, é possível experimentar valores diferentes para funções de ligação e variância.

Os três últimos componentes `rhos` se referem ao coeficiente de correlação entre as respostas. Figura7.7. `rho1` correlaciona tamanho de grão e contagem de sementes; possui uma considerável correlação de 0,63. `rho2` e `rho3` representam a correlação entre tamanho de grão e ervilhas viáveis, contagem de sementes e ervilhas viáveis, respectivamente. Ambas as correlações são baixas.

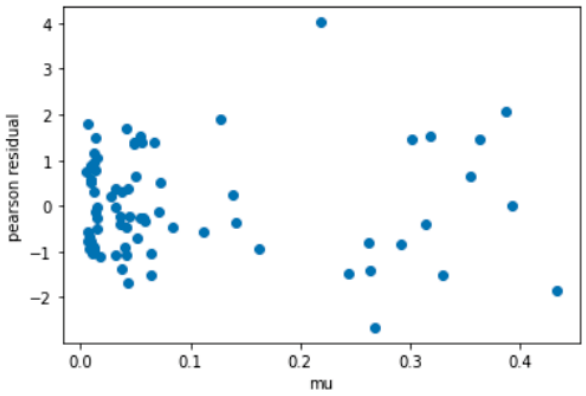


Figura 7.6: Pearson residuals para viablePeas

	coef	std err	z	P>|z|	[0.025	0.975]
rho_1	0.6368	0.107	5.928	0.000	0.426	0.847
rho_2	0.0695	0.115	0.603	0.546	-0.156	0.295
rho_3	0.0878	0.115	0.765	0.444	-0.137	0.313

Figura 7.7: Rhos report para as três respostas

8 COMENTÁRIO FINAIS

Esta dissertação apresentou a primeira implementação do modelo MCGLM em `Python`, uma estrutura estatística flexível para ajuste em uma ampla variedade de dados. A biblioteca `mcglm` estende o `statsmodels`. Ele fornece uma excelente interface para acessar parâmetros estimados juntamente com medições de qualidade do treinamento.

A biblioteca `mcglm` é uma nova alternativa para análises estatísticas na linguagem `Python`, passível de publicação na biblioteca `statsmodels` como uma nova *API*. É um novo capítulo do desenvolvimento estatístico na linguagem `Python` e na Ciência De Dados.

Limitações conhecidas envolvem a impossibilidade de escolhas para a função de ligação da covariância, como `inverse` e `exponencial matricial`, afinal apenas a função `identity` está disponível; o código não possui suporte para processamento paralelo escalável; apenas três das múltiplas opções de dependências foram implementadas. Para delimitar o escopo desta biblioteca, priorizamos a implementação exclusiva da função de ligação da covariância `identity`, que permite o ajuste de inúmeros modelos, elencados nesta dissertação. Em adição, dada a flexibilidade do código, resultante da aplicação da orientação a objetos, outras funções de ligação podem ser implementadas sem fricção no futuro.

Uma meta dessa biblioteca, e uma diferença em relação ao pacote da linguagem `R` existente, é o desenvolvimento orientado a objetos. Com o software concluído, fica evidente a adequação da filosofia de desenvolvimento à produção de modelos estatísticos. A bilioteca possui uma implementação legível, extensível, independente e testável; atributos desejáveis no desenvolvimento de modelos estatísticos.

Assumindo os traços principais do modelo MCGLM, as ferramentas estatísticas mais similares são os `Modelos Mistos` e `Copulas`, exceto pela capacidade de análise em múltiplas respostas. Como atividade futura, é possível uma análise comparativa entre as ferramentas estatísticas, explorando diferentes ajustes de modelos em estruturas de dependências únicas. Como a biblioteca `statsmodels` implementa os modelos complementares, `Python` é um ambiente adequado para a realização desta análise.

REFERÊNCIAS

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. e Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Agha, S. e Haider, A. (2014). An introduction to data mining technique. *IJAETMAS*, 3:5.

Alkemade, R. M., Boattini, E., Filion, L. e Smallenburg, F. (2022). Comparing machine learning techniques for predicting glassy dynamics. *The Journal of chemical physics*, 156 20:204503.

Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Croz, J., Hammarling, S., Demmel, J., Bischof, C. e Sorensen, D. (1990). Lapack: A portable linear algebra library for high-performance computers. *[No source information available]*, páginas 2–11.

Anderson, T. W. (1973). Asymptotically efficient estimation of covariance matrices with linear structure. *The Annals of Statistics*, 1(1):135–141.

Ardeshir, N., Sanford, C. e Hsu, D. (2021). Support vector machines and linear regression coincide with very high-dimensional features. Em *NeurIPS*.

Baudart, G., Hirzel, M., Kate, K., Mandel, L. e Shinnar, A. (2018). Yaps: Python frontend to stan.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S. e Smith, K. (2011). Cython: The best of both worlds. *Computing in Science and Engg.*, 13(2):31–39.

Bhattacharya, P. e Burman, P. (2016). 11 - linear models. Em Bhattacharya, P. e Burman, P., editores, *Theory and Methods of Statistics*, páginas 309–382. Academic Press.

Bingham, E., Chen, J., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P. e Goodman, N. (2018). Pyro: Deep universal probabilistic programming.

Bommarito, E. e Bommarito, I. (2021). An empirical analysis of the r package ecosystem.

Bonat, W. (2018). Multiple response variables regression models in r : The mcglm package. *Journal of Statistical Software*, 84.

Bonat, W. H. (2016). Modelling mixed outcomes in additive genetic models. *ArXiv*.

Bonat, W. H. e Jørgensen, B. (2016a). Multivariate covariance generalized linear models. *Journal of the Royal Statistical Society C*, 65(5):649–675.

Bonat, W. H. e Jørgensen, B. (2016b). Multivariate covariance generalized linear models. *Journal of the Royal Statistical Society C*, 65(5):649–675.

Breslow, N. E. e Clayton, D. G. (1993). Approximate inference in generalized linear mixed models. *Journal of the American Statistical Association*, 88:9–25.

Carey VJ, W. Y. (2011). Working covariance model selection for generalized estimating equations. *Statistics in Medicine*, páginas 3117–3124.

Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P. e Riddell, A. (2017). Stan : A probabilistic programming language. *Journal of Statistical Software*, 76.

Chiu TYM, Leonard T, T. K. (1996). The matrix-logarithmic covariance model. *Journal of the American Statistical Association*.

de Freitas, L. A. C. e Bonat, W. H. (2022). Hypothesis tests for multiple responses regression models in r: The htmcglm package.

De Luca, G., Magnus, J. e Peracchi, F. (2017). Weighted-average least squares estimation of generalized linear models. *SSRN Electronic Journal*.

Fahrmeir, L. e Lang, S. (2001). Bayesian inference for generalized additive mixed models based on markov random field priors. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 50:201 – 220.

Feller, W. (1957). *An introduction to probability theory and its applications / William Feller*. Wiley New York, 2nd ed. edition.

Fisher, R. A. (1919). Xv.—the correlation between relatives on the supposition of mendelian inheritance. *Transactions of the Royal Society of Edinburgh*, 52(2):399–433.

Galton, F. (1886). Regression towards mediocrity in hereditary stature. *Journal of the Anthropological Institute of Great Britain and Ireland*, páginas 246–263.

Hallin, M. (2014). *Gauss–Markov Theorem in Statistics*.

Harris, H. e. a. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Hastie, T. e Tibshirani, R. (1986). Generalized additive models (with discussion). *Statistical Science 1*, página 297–318.

Hwang, S. e Basawa, I. (2011). Godambe estimating functions and asymptotic optimal inference. *Statistics and Probablitity Letters*, 81:1121–1127.

Ihaka, R. e Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314.

Jennrich, R. I. (1969). A Newton-Raphson algorithm for maximum likelihood factor analysis. *Psychometrika*, 34.

Jørgensen, B. e Knudsen, S. J. (2004). Parameter orthogonality and bias adjustment for estimating functions. *Scandinavian Journal of Statistics*, 31(1):93–114.

Jørgensen, B. (1987). Exponential dispersion models. *Journal of the Royal Statistical Society*.

Jørgensen, B. (1997). The theory of dispersion models. *CRC Press*.

Jørgensen, B. e Kokonendji, C. C. (2015). Discrete dispersion models and their tweedie asymptotics. *AStA Advances in Statistical Analysis, 100(1)*, páginas 43–78.

Kateri, M. (2014). *Log-Linear Models*, páginas 85–124.

Kaufmann, J. e Schering, A. (2014). *Analysis of Variance ANOVA*.

Krupskii, P. e Joe, H. (2013). Factor copula models for multivariate data. *Journal of Multivariate Analysis*, 120(1):85–101.

Li, L., Wu, T. e Feng, C. X. (2020). Model diagnostics for censored regression via randomized survival probabilities. *Statistics in Medicine*, 40:1482 – 1497.

Liang, K.-Y. e Zeger, S. L. (1986). Longitudinal data analysis using generalized linear models. *Biometrika*, 73(1):13–22.

Liang, K.-Y., Zeger, S. L. e Qaqish, B. (1992). Multivariate regression analyses for categorical data. *Journal of the Royal Statistical Society. Series B (Methodological)*, 54(1):3–40.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, Boston, MA.

Martinez-Beneito, M. A. (2013). A general modelling framework for multivariate disease mapping. *Biometrika*, 100(3):539–553.

Masarotto, G. e Varin, C. (2012). Gaussian copula marginal regression. *Electronic Journal of Statistics*, 6:1517–1549.

Meent, J.-W., Paige, B., Yang, H. e Wood, F. (2018). An introduction to probabilistic programming.

Miura, K. (2011). An introduction to maximum likelihood estimation and information geometry. *Interdisciplinary Information Sciences (IIS)*, 17.

Moral, R. A., Hinde, J. e Demétrio, C. G. B. (2017). Half-normal plots and overdispersed models in r: The hnp package. *Journal of Statistical Software*, 81(10):1–23.

Müller, M. (2004). Generalized linear models.

Nelder, J. e Wedderburn, R. (1972). Generalized linear models. páginas 370–384.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. e Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. Em *Advances in Neural Information Processing Systems 32*, páginas 8024–8035. Curran Associates, Inc.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. e Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Postelnicu, T. (2011). *Probit Analysis*, páginas 1128–1131.

Pourahmadi, M. (2000). Maximum likelihood estimation of generalised linear models for multivariate normal covariance matrix. *Biometrika*, 87(2):425–435.

Salvatier, J., Wiecki, T. e Fonnesbeck, C. (2016). Probabilistic programming in python using pymc3.

Seabold, S. e Perktold, J. (2010). Statsmodels: Econometric and statistical modeling with python. *Proceedings of the 9th Python in Science Conference*, 2010.

Stigler, S. M. (1981). Gauss and the Invention of Least Squares. *The Annals of Statistics*, 9(3):465 – 474.

Trujillo, M., Hébert-Dufresne, L. e Bagrow, J. (2022). The penumbra of open source: projects outside of centralized platforms are longer maintained, more academic and more collaborative. *EPJ Data Science*, 11.

Verbeke, G., Fieuws, S., Molenberghs, G. e Davidian, M. (2014). The analysis of multivariate longitudinal data: A review. *Statistical Methods in Medical Research*, 23(1):42–59.

Verbeke, G., Molenberghs, G. e Rizopoulos, D. (2010). *Random Effects Models for Longitudinal Data*, páginas 37–96.

Virtanen, P. e. a. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.

Wachs, J., Nitecki, M., Schueller, W. e Polleres, A. (2022). The geography of open source software: Evidence from github. *ArXiv*, abs/2107.03200.

Wald, A. (1943). Tests of statistical hypotheses concerning several parameters when the number of observations is large. *Transactions of the American Mathematical society.*, páginas 426–482.

Wedderburn, R. W. M. (1974). Quasi likelihood functions, generalized linear models, and the Gauss—Newton method. *Biometrika*, 61(3):439–447.

Widyaningsih, P., Saputro, D. e Putri, A. (2017). Fisher scoring method for parameter estimation of geographically weighted ordinal logistic regression (gwolr) model. *Journal of Physics: Conference Series*, 855:012060.

Yilmaz, R., Sezgin, A., Kurnaz, S. e Arslan, Y. Z. (2019). *Object-Oriented Programming in Computer Science*, páginas 1439–1451.

APÊNDICE A – BASE DE CÓDIGO DA BIBLIOTECA *MCGLM*

Nesta seção, apresentamos a implementação em *Python* da biblioteca *mcglm*. Ela serve como guia fundamental ao desenvolvimento objeto desta dissertação e como um apêndice ao capítulo "Arquitetura de Software e Implementação". Destacamos classes e métodos fundamentais da biblioteca.

A biblioteca *mcglm* divide-se nas implementações das classes: `MCGLM`, `MCGLMMean`, `MCGLMVariance`, `MCGLMCAttributes`, `MCGLMParameters` e `MCGLMResults`. Abaixo, os códigos completos dessas implementações.

Listagem A.1: Classe `MCGLM`.

```
class MCGLM(MCGLMMean, MCGLMVariance):
    __doc__ = """
    MCGLM class that implements MCGLM stastical models. (Bonat, Jorgensen 2015)

    It extends GLM for multi-responses and dependent components by fitting
    second-moment assumptions.

    Parameters
    ----------
    endog : array_like
        1d array of endogenous response variable. In case of multiple
        responses, the user must pass the responses on a list.
    exog : array_like
        A dataset with the endogenous matrix in a Numpy fashion. Since the
        library doesn't set an intercept by default, the user must add it. In
        the case of multiple responses, the user must pass the design matrices
        as a python list.
    z : array_like
        List with matrices components of the linear covariance matrix.
    link : array_like, string or None
        Specification for the link function. The MCGLM library implements the
        following options: identity, logit, power, log, probit, cauchy,
        cloglog, loglog, negativebinomial. In the case of None, the library
        chooses the identity link. In multiple responses, user must pass
        values as list.
    variance : array_like, string or None
        Specification for the variance function. The MCGLM library implements
        the following options: constant, tweedie, binomialP, binomialPQ,
        geom_tweedie, poisson_tweedie. In the case of None, the library
        chooses the constant link. In multiple responses, user must pass
        values as list.
    offset : array_like or None
        Offset for continuous or count. In multiple responses, user must pass
        values as list.
    Ntrial : array_like or None
        Ntrial for binomial responses. In multiple responses, user must pass
        values as list.
    power_fixed : array_like or None
        Parameter that allows estimation of power when variance functions is
        either tweedie, geom_tweedie or poisson_tweedie. In multiple
        responses, user must pass values as list.
    maxiter : float or None
        Number max of iterations. Defaults to 200.
```

```
    tol : float or None
       Threshold of minimum absolute change on paramaters. Defaults to 0.0001.
    tuning : float or None
       Step size parameter. Defaults to 0.5.
    weights : array_like or None
       Weight matrix. Defaults to None.

    Examples
    ----------
    >>> import statsmodels.api as sm
    >>> data = sm.datasets.scotland.load()
    >>> data.exog = sm.add_constant(data.exog)

    >>> model = sm.GLM(data.endog, data.exog, z=[mc_id(data.exog)],
    ...                link="log", variance="tweedie",
    ...                power=2, power_fixed=False)

    >>> model_results = model.fit()
    >>> model_results.mu
    >>> model_results.pearson_residuals
    >>> model_results.aic
    >>> model_results.bic
    >>> model_results.loglikelihood

    Notes
    -----
    MCGLM is a brand new model, which provides a solid statistical model for
    fitting multi-responses non-gaussian, dependent, or independent data based
    on second-moment assumptions. When a user instantiates an mcglm object,
    she must specify attributes such as link, variance, and z matrices; it
    will drive the overall behavior of the model.
    For more details, check articles and documentation provided.
    """

    def __init__(
       self,
       endog,
       exog,
       z,
       link=None,
       variance=None,
       offset=None,
       ntrial=None,
       power=None,
       power_fixed=None,
       maxiter=50,
       tol=0.001,
       tuning=1,
       weights=None,
    ):
       super(MCGLM, self).__init__()

       self._max_iter = maxiter
       self._tol = tol
       self._tuning = tuning
       self._weights = weights

       self._n_targets = None
```

```
102         self._n_obs = None
103         self._X = None
104         self._y_values = None
105         self._y_names = None
106         self._z = None
107         self._ntrial = None
108         self._tau_initial = None
109         self._beta_initial = None
110         self._rho_initial = None
111         self._offset = None
112         self._link = None
113         self._variance = None
114         self._power_fixed = None
115         self._power_initial = None
116 
117         (
118            self._n_targets,
119            self._n_obs,
120            self._X,
121            self._y_values,
122            self._y_names,
123            self._z,
124            self._ntrial,
125            self._tau_initial,
126            self._beta_initial,
127            self._rho_initial,
128            self._offset,
129            self._link,
130            self._variance,
131            self._power_fixed,
132            self._power_initial,
133         ) = self.__calculate_static_attributes(
134            endog, exog, link, variance, z, offset, power, power_fixed, ntrial
135         )
136 
137     @property
138     def df_model(self):
139        """Calculates the degree of freedom for the model."""
140        return len(self._beta_initial[0]) + len(self._tau_initial[0])
141 
142     @property
143     def df_resid(self):
144        """Calculates the degree of freedom for the model residuals."""
145        return self._n_obs - self.df_model
146 
147     def __calculate_static_attributes(
148        self, endog, exog, link, variance, z, offset, power, power_fixed,
149        ntrial
150     ):
151        """Base method to warm up the pivotal artifacts for model training. It
152        fulfills the None with default values, sets the power value, and
153        creates the base vectors.
154 
155        Args:
156           endog (list or np.array): _description_
157           exog (list or np.array): _description_
158           link (list): _description_
159           variance (list): _description_
```

```
        z (list): _description_
        offset (list): _description_
        power (list): _description_
        power_fixed (list): _description_
        ntrial (list): _description_
    """

    def initial_power(variance):
        if variance in [
            "binomialP",
            "tweedie",
            "geom_tweedie",
            "poisson_tweedie",
            "constant",
        ]:
            return 1
        elif variance == "binomialPQ":
            return (1, 1)
        else:
            return 0

    n_targets = len(exog) if isinstance(exog, list) else 1
    n_obs = exog[0].shape[0] if n_targets > 1 else exog.shape[0]

    set_offset = offset is None
    set_link = link is None
    set_variance = variance is None
    set_power_fixed = power_fixed is None
    set_power = power is None

    tau = list()
    beta = list()
    rho = list()

    offset_list = list()
    link_list = list()
    variance_list = list()
    power_fixed_list = list()
    power_list = list()
    y_values = list()
    y_names = list()

    if n_targets == 1:
        X = [exog]
        y_values = [endog.values]
        y_names = [endog.name]
        z = [z]
        ntrial = [ntrial]
        tau = [[0 for i in range(len(z[0]))]]
        beta = [np.ones(X[0].shape[1])]
        rho = 0

        offset_list = [None] if set_offset else [offset]
        link_list = ["identity"] if set_link else [link]
        variance_list = ["constant"] if set_variance else [variance]
        power_fixed_list = [True] if set_power_fixed else [power_fixed]
        power_list = [initial_power(variance_list[0])] if set_power else
        [power]
```

```
218
219         else:
220             X = exog
221             if not set_offset:
222                 offset_list = offset
223             if not set_link:
224                 link_list = link
225             if not set_variance:
226                 variance_list = variance
227             if not set_power_fixed:
228                 power_fixed_list = power_fixed
229             if not set_power:
230                 power_list = power
231
232             for index in range(n_targets):
233                 tau.append([0 for i in range(len(z[index]))])
234                 beta.append([np.ones(exog[index].shape[1])])
235                 y_values.append(endog[index].values)
236                 y_names.append(endog[index].name)
237
238                 if set_offset:
239                     offset_list.append(None)
240                 if set_link:
241                     link_list.append("identity")
242                 if set_variance:
243                     variance_list.append("constant")
244                 if set_power_fixed:
245                     power_fixed_list.append(True)
246                 if set_power:
247                     power_list.append(initial_power(variance[index]))
248
249             rho = list()
250             for _ in range(int(n_targets * (n_targets - 1) / 2)):
251                 rho.append(0)
252
253         return (
254             n_targets,
255             n_obs,
256             X,
257             np.concatenate(y_values, axis=None),
258             y_names,
259             z,
260             ntrial,
261             tau,
262             beta,
263             rho,
264             offset_list,
265             link_list,
266             variance_list,
267             power_fixed_list,
268             power_list,
269         )
270
271     def fit(self):
272         """The interface to run the inference for MCGLM statistical model."""
273         (
274             regression_historical,
275             dispersion_historical,
```

```
276             residue,
277             varcov,
278             joint_inv_sensitivity,
279             joint_variability,
280             n_iter,
281             mu,
282             rho,
283             tau,
284             power,
285             parameters_target,
286             c_inverse,
287             c_values,
288         ) = self._fit()
289
290         regression_parameters = regression_historical[-1]
291         dispersion_parameters = dispersion_historical[-1]
292
293         p_log_likelihood = self.__p_log_likelihood(
294             self._y_values, mu, c_values, c_inverse
295         )
296         aic = self.__p_aic(p_log_likelihood, self.df_model)
297         bic = self.__p_bic(p_log_likelihood, self.df_model, len(self.
298         _y_values))
299
300         return MCGLMResults(
301             normalized_var_cov=abs(varcov),
302             nobs=self._n_obs,
303             n_targets=self._n_targets,
304             y_names=self._y_names,
305             regression=regression_parameters,
306             dispersion=parameters_target,
307             n_iter=n_iter,
308             residue=residue,
309             rho=rho,
310             tau=tau,
311             power=power,
312             link=self._link,
313             variance=self._variance,
314             power_fixed=self._power_fixed,
315             p_log_likelihood=p_log_likelihood,
316             aic=aic,
317             bic=bic,
318             df_resid=self.df_resid,
319             df_model=self.df_model,
320             mu=mu.reshape(self._n_targets, -1),
321             y_values=self._y_values,
322             X=self._X,
323             ntrial=self._ntrial,
324         )
325
326     def __p_log_likelihood(self, endog, mu, c_values, c_inverse):
327         """
328         Gaussian Pseudo-loglikelihood
329         """
330
331         def gauss(residue, det_sigma, inv_sigma):
332             import math
333
```

```
            size_residue = len(residue)
            dens = (
                -1 * (size_residue / 2) * np.log(2 * math.pi)
                - 0.5 * det_sigma
                - (residue.T * inv_sigma * residue)
            )
            return dens.mean()

        residue = endog - mu
        det_sigma = slogdet(block_diag(c_values))[1]
        log_likelihood = gauss(
            residue=residue, det_sigma=det_sigma, inv_sigma=c_inverse
        )

        return log_likelihood

    def __p_aic(self, pseudo_loglike, degrees_of_freedom):
        """
        pseudo Akaike Information criterion
        """
        return round(2 * degrees_of_freedom - 2 * pseudo_loglike, 2)

    def __p_bic(self, pseudo_loglike, degrees_of_freedom, length_endogenous):
        """
        pseudo Bayesian Information criterion
        """
        return round(
            degrees_of_freedom * np.log(length_endogenous) - 2 *
            pseudo_loglike, 2
        )

    def _fit(self):
        """This method implements the core inference for MCGLM, by all the
        means of the two-moment assumptions."""
        W = (
            np.diag(np.ones(len(self._y_values)))
            if self._weights is None
            else self._weights
        )

        regression_historical = []
        dispersion_historical = []

        # To a warm up the start
        self.__update_optimal_dispersion()

        rho = self._rho_initial
        tau = self._tau_initial
        power = self._power_initial

        # Setting up
        regression = np.array(self._beta_initial)
        dispersion = np.array(self.__create_dispersion_vector(rho, power, tau))

        regression_historical.append(regression)
        dispersion_historical.append(dispersion)
        for iter in range(self._max_iter):
            # First moment
```

```
392
393             (
394                 new_regression,
395                 quasi_score,
396                 mean_sensitivity,
397                 mean_variability,
398             ) = self.update_beta(regression, W, power, rho, tau)
399             regression_historical.append(new_regression)
400
401             # Second moment
402             mu_attributes, mu, mu_derivatives = self.calculate_mean_features(
403                 self._link, new_regression, self._X, self._offset
404             )
405
406             (
407                 new_dispersion,
408                 c_inverse,
409                 c_values,
410                 c_derivatives_componentes,
411                 var_sensitivity,
412             ) = self.update_covariates(
413                 mu_attributes, rho, power, tau, W, dispersion, mu
414             )
415
416             dispersion_historical.append(new_dispersion)
417
418             regression, dispersion, rho, power, tau = self.__iteration_update(
419                 new_regression, new_dispersion, rho, power, tau
420             )
421             if self.__check_stop_criterion(
422                 regression_historical, dispersion_historical
423             ):
424                 break
425
426         residue = self._y_values - mu
427
428         (
429             varcov,
430             joint_inv_sensitivity,
431             joint_variability,
432         ) = MCGLMParameters._parameters_attributes(
433             residue,
434             mu_derivatives,
435             W,
436             quasi_score,
437             mean_sensitivity,
438             mean_variability,
439             c_inverse,
440             c_values,
441             c_derivatives_componentes,
442             var_sensitivity,
443         )
444
445         parameters_target = self.__get_dispersion_parameters(mu, rho, tau,
446         dispersion)
447
448         return (
449             regression_historical,
```

```
450            dispersion_historical,
451            residue,
452            varcov,
453            joint_inv_sensitivity,
454            joint_variability,
455            str(iter),
456            mu,
457            rho,
458            tau,
459            power,
460            parameters_target,
461            c_inverse,
462            c_values,
463        )
464
465    def __get_dispersion_parameters(self, mu, rho, tau, dispersion):
466
467        covariances = dispersion.tolist()
468
469        if self._n_targets == 1:
470            rho = np.array([rho])
471        elif self._n_targets > 1:
472            rho = np.array([covariances.pop(0) for _ in range(len(rho))])
473
474        parameters_target = list()
475        for index in range(self._n_targets):
476            power_mu = {
477                "power": self._power_initial[index]
478                if self._power_fixed[index]
479                else covariances.pop(0),
480                "mu": mu[index * self._n_obs : (index + 1) * self._n_obs],
481            }
482
483            size_parameters = (
484                len(tau[index]) if self._power_fixed[index] else len(tau
485                [index])
486            )
487
488            scale_output = {
489                "scalelist": [
490                    round(covariances.pop(0), 3) for _ in range
491                    (size_parameters)
492                ]
493            }
494
495            parameters_target.append({**scale_output, **power_mu})
496
497        return parameters_target
498
499    def __check_stop_criterion(self, regression_historical,
500    dispersion_historical):
501        regression_lift = abs(regression_historical[-2] - regression_historical
502        [-1])
503        dispersion_lift = abs(dispersion_historical[-2] - dispersion_historical
504        [-1])
505
506        for position in range(len(regression_lift)):
507            if np.all(regression_lift[position] < self._tol) and np.all(
```

```
            dispersion_lift < self._tol
        ):
            return True
    return False

def __iteration_update(self, new_regression, new_dispersion, rho, power,
tau):

    if self._n_targets == 1:
        if self._power_fixed[0]:
            return new_regression, new_dispersion, rho, power,
            [new_dispersion]
        else:
            return new_regression, new_dispersion, rho, [new_dispersion
            [0]], [new_dispersion[1:].tolist()]

    if (isinstance(rho, list)) or (isinstance(rho, np.ndarray)):
        new_rho = new_dispersion[0 : len(rho)]
        stack_index = len(rho)
    else:
        new_rho = new_dispersion[0]
        stack_index = 1

    new_power = list()
    new_tau = list()
    for position in range(len(tau)):
        # power section
        if self._power_fixed[position]:
            new_power.append(power[position])
        else:
            new_power.append(new_dispersion[stack_index])
            stack_index += 1

        # tau section.
        additions = 0
        temp_tau = list()
        for index in range(len(tau[position])):
            temp_tau.append(new_dispersion[stack_index + index])
            additions += 1
        stack_index += additions
        new_tau.append(temp_tau)

    return new_regression, new_dispersion, new_rho, new_power, new_tau

def __create_dispersion_vector(self, rho, power, tau):
    covariance = list()

    if self._n_targets > 1:
        if isinstance(rho, list):
            covariance.extend(rho)
        else:
            covariance.append(rho)
    for n_outputs in range(len(tau)):
        if not self._power_fixed[n_outputs]:
            covariance.append(power[n_outputs])
        covariance = covariance + tau[n_outputs]
    return covariance
```

```
566     def __update_optimal_dispersion(self):
567         """Update optimal dispersion method calculates optimal values for
568         regression parameters and the dispersion. It harnesses the GLM API of
569         statsmodels for calculating those values."""
570 
571         def logit_est(endog, exog, offset):
572             """For the Logit link function, the method adjusts a GLM with
573             Binomial family. It retrieves the parameters specified.
574 
575             Args:
576                 endog (array-like): outcome variable.
577                 exog (array-like): independent variables.
578                 offset (int): shift on response variable.
579 
580             Returns:
581                 tuple: regression parameters, dispersion parameter.
582             """
583             mdl = GLM(endog, exog, family=sm.families.Binomial(),
584             offset=offset)
585             mdl_results = mdl.fit()
586             return mdl_results.params.values, mdl_results.scale
587 
588         def loglog_est(endog, exog, offset):
589             """For the LogLog link function, the method adjusts a GLM with
590             Binomial family. It retrieves the parameters specified.
591 
592             Args:
593                 endog (array-like): outcome variable.
594                 exog (array-like): independent variables.
595                 offset (int): shift on response variable.
596 
597             Returns:
598                 tuple: regression parameters, dispersion parameter.
599             """
600             mdl = GLM(endog, exog, family=sm.families.Binomial(LogLog()),
601             offset=offset)
602             mdl_results = mdl.fit()
603             return mdl_results.params.values, mdl_results.scale
604 
605         def cloglog_est(endog, exog, offset):
606             """For the CLogLog link function, the method adjusts a GLM with
607             Binomial family. It retrieves the parameters specified.
608 
609             Args:
610                 endog (array-like): outcome variable.
611                 exog (array-like): independent variables.
612                 offset (int): shift on response variable.
613 
614             Returns:
615                 tuple: regression parameters, dispersion parameter.
616             """
617             mdl = GLM(
618                 endog, exog, family=sm.families.Binomial(CLogLog()),
619                 offset=offset
620             )
621             mdl_results = mdl.fit()
622             return mdl_results.params.values, mdl_results.scale
623 
```

```
624         def cauchy_est(endog, exog, offset):
625             """For the Cauchy link function, the method adjusts a GLM with
626             Binomial family. It retrieves the parameters specified.
627 
628             Args:
629                 endog (array-like): outcome variable.
630                 exog (array-like): independent variables.
631                 offset (int): shift on response variable.
632 
633             Returns:
634                 tuple: regression parameters, dispersion parameter.
635             """
636             mdl = GLM(endog, exog, family=sm.families.Binomial(cauchy()),
637             offset=offset)
638             mdl_results = mdl.fit()
639             return mdl_results.params.values, mdl_results.scale
640 
641         def probit_est(endog, exog, offset):
642             """For the Cauchy link function, the method adjusts a GLM with
643             Binomial family. It retrieves the parameters specified.
644 
645             Args:
646                 endog (array-like): outcome variable.
647                 exog (array-like): independent variables.
648                 offset (int): shift on response variable.
649 
650             Returns:
651                 tuple: regression parameters, dispersion parameter.
652             """
653             mdl = GLM(endog, exog, family=sm.families.Binomial(probit()),
654             offset=offset)
655             mdl_results = mdl.fit()
656             return mdl_results.params.values, mdl_results.scale
657 
658         def identity_est(endog, exog, offset=None):
659             """For the Identity link function, the method adjusts a OLS. It
660             retrieves the parameters specified.
661 
662             Args:
663                 endog (array-like): outcome variable.
664                 exog (array-like): independent variables.
665                 offset (int): shift on response variable.
666 
667             Returns:
668                 tuple: regression parameters, dispersion parameter.
669             """
670             mdl = sm.OLS(endog, exog, offset=offset)
671             mdl_results = mdl.fit()
672             return mdl_results.params, mdl_results.scale
673 
674         def log_est(endog, exog, offset):
675             """For the Log link function, the method adjusts a GLM with Tweedie
676             (power=1) family. It retrieves the parameters specified.
677 
678             Args:
679                 endog (array-like): outcome variable.
680                 exog (array-like): independent variables.
681                 offset (int): shift on response variable.
```

```
682
683             Returns:
684                 tuple: regression parameters, dispersion parameter.
685             """
686             mdl = GLM(
687                 endog, exog, family=sm.families.Tweedie(var_power=1),
688                 offset=offset
689             )
690             mdl_results = mdl.fit()
691             return mdl_results.params, mdl_results.scale
692
693         def power_est(endog, exog, offset):
694             """For either the Power or Reciprocal link function, the method
695             adjusts a GLM with Tweedie(power=2) family. It retrieves the
696             parameters specified.
697
698             Args:
699                 endog (array-like): outcome variable.
700                 exog (array-like): independent variables.
701                 offset (int): shift on response variable.
702
703             Returns:
704                 tuple: regression parameters, dispersion parameter.
705             """
706             mdl = GLM(
707                 endog, exog, family=sm.families.Tweedie(var_power=2),
708                 offset=offset
709             )
710             mdl_results = mdl.fit()
711             return mdl_results.params.values, mdl_results.scale
712
713         def negative_binomial_est(endog, exog, offset):
714             """For the Negative Binomial link function, the method adjusts a
715             GLM with Tweedie(power=2) family. It retrieves the parameters
716             specified.
717
718             Args:
719                 endog (array-like): outcome variable.
720                 exog (array-like): independent variables.
721                 offset (int): shift on response variable.
722
723             Returns:
724                 tuple: regression parameters, dispersion parameter.
725             """
726             mdl = GLM(
727                 endog, exog, family=sm.families.Tweedie(var_power=2),
728                 offset=offset
729             )
730             mdl_results = mdl.fit()
731             return mdl_results.params.values, mdl_results.scale
732
733         first_estimation = {
734             "logit": logit_est,
735             "identity": identity_est,
736             "power": power_est,
737             "log": log_est,
738             "probit": probit_est,
739             "cauchy": cauchy_est,
```

```
            "cloglog": cloglog_est,
            "loglog": loglog_est,
            "negativebinomial": negative_binomial_est,
            "inverse_power": log_est,
            "reciprocal": power_est,
        }

        for target in range(self._n_targets):

            self._beta_initial[target], dispersion = first_estimation[
                self._link[target]
            ](
                self._y_values[
                    target * self._n_obs : target * self._n_obs + self._n_obs
                ],
                self._X[target],
                self._offset[target],
            )
            self._tau_initial[target][0] = dispersion
```

Listagem A.2: Classe `MCGLMMean`.

```
    AVAILABLE_LINK_FUNCTIONS = {
        "logit": Logit(),
        "identity": Power(power=1.0),
        "power": Power(),
        "log": Log(),
        "probit": probit(),
        "cauchy": cauchy(),
        "cloglog": CLogLog(),
        "loglog": LogLog(),
        "negativebinomial": NegativeBinomial(),
        "inverse_power": inverse_power(),
        "reciprocal": Power(),
    }


    class MCGLMMean(MCGLMCAttributes):
        """
        MCGLMMean is the class for the first moment adjustment within MCGLM
        inference. It handles lifecycle completely, ranging from mu and
        derivatives to quasi-likelihood calculations.
        This class has two interfaces: 'calculate_mean_features' which
        calculates mu attributes, and 'update_beta' that applies
        quasi-likelihood estimation and retrieves a new beta.
        This class implements the Estimating Equation Quasi-score (Wedderburn,
        1974) and the second-order optimization algorithm (Jennrich, 1969) and
        (Widyaningsih et al., 2017).

        References
        ----------
        Wedderburn, R. W. M. (1974). Quasi-likelihood functions, generalized
        linear models, and the Gauss-Newton method. Biometrika, 61-3:439-447.

        Jennrich, R. I. (1969). A Newton-Raphson algorithm for maximum
        likelihood factor analysis. Psychometrika, 34.

        Widyaningsih, P., Saputro, D. e Putri, A. (2017). Fisher scoring
```

```
        method for parameter estimation of geographically weighted ordinal
        logistic regression (gwolr) model. Journal of Physics: Conference
        Series, 855:012060.
        """

        def __init__(self):
            super(MCGLMCAttributes, self).__init__()

        def __get_link_function(self, link: str):
            """Check whether the link function is available

            Parameters
            ----------
                link : str
                    A link function
            Returns
            -------
                statsmodels.genmod.families.links : a corresponding object for
                the link function.
            """
            assert link in AVAILABLE_LINK_FUNCTIONS, (
                f"The link function " + str(link) + " isn't available"
            )
            return AVAILABLE_LINK_FUNCTIONS.get(link.lower())

        def __linear_predictor(self, X, beta):
            """Method linear predictor applies linear operation between
            covariates and the regression parameters.

            Parameters
            ----------
                X : array_like
                    Design matrix with covariates
                beta : array-like
                    Regression parameters
            Returns
            -------
                array_like : The calculated output vector.
            """
            return np.dot(X, beta)

        def _link_function_attributes(
            self, link: str, beta: np.array, X: np.array, offset: int = 0
        ):
            """A protected method for calling the __link_function_attributes

            Parameters
            ----------
                link : str
                    Link function.
                beta : array_like
                    Regression Parameters.
                X : array_like
                    Matrix with covariates.
                offset : (int, optional)
                    Offset add value. Defaults to 0.
            Returns
            -------
```

```
95              dict : Dict value with the mean and its derivatives.
96           """
97           return self.__link_function_attributes(link, beta, X, offset)
98
99        def __link_function_attributes(
100          self, link: str, beta: np.array, X: np.array, offset: int = 0
101       ):
102          """The method __link_function_attributes calculates the vector of
103          expected values and its derivatives. It returns data as dictionary
104
105          Parameters
106          ----------
107             link : str
108                Link function.
109             beta : array-like
110                Regression Parameters.
111             X : array-like
112                Matrix with covariates.
113             offset : int, optional
114                Offset add value. Defaults to 0.
115          Returns
116          -------
117             dict : Dict value with the mean and its derivatives.
118          """
119          link_func = self.__get_link_function(link.lower())
120          eta = self.__linear_predictor(X, beta)
121          if offset is None:
122             offset = 0
123          eta = eta + offset
124
125          mu = link_func.inverse(eta)
126          deriv = X * link_func.inverse_deriv(eta)[:, None]
127          return dict(mu=mu, deriv=deriv)
128
129       def calculate_mean_features(self, link, beta, X, offset):
130          """Base method to calculate every attribute related to the mean.
131
132          Parameters
133          ----------
134             link : str
135                Link function.
136             beta : array-like
137                Regression Parameters.
138             X : array-like
139                Matrix with covariates.
140             offset : int, optional)
141                Offset add value. Defaults to 0.
142          Returns
143          -------
144             tuple : Mean attributes, the raw mean and its derivatives.
145          """
146          mu_attributes_per_response = list(
147             map(self.__link_function_attributes, link, beta, X, offset)
148          )
149          mu = np.array(
150             list(
151                itertools.chain(
152                   *[mu_value.get("mu") for mu_value in
```

```
                    mu_attributes_per_response]
                )
            )
        )
        d = block_diag(
            *[mu_value.get("deriv") for mu_value in
            mu_attributes_per_response]
        )
        return mu_attributes_per_response, mu, d

    def update_beta(self, beta, W, power, rho, tau):
        """The method update_beta takes the current beta, leverages the
        quasi-likelihood estimator to calculate the next regression
        parameters.

        Parameters
        ----------
            beta : array-like
                Regression Parameters.
            W : array-like
                Weight matrix
            power : float
                Power parameter
            rho : float
                Correlation parameters
            tau : float
                Dispersion parameters.
        Returns
        -------
            tuple : A tuple with the new regression parameters, the
            quasi-score parameter, sensitivity and the variability matrix.
        """
        mu_attributes, mu, derivative_mu = self.calculate_mean_features(
            self._link, beta, self._X, self._offset
        )
        c_inverse = self.c_inverse(mu_attributes, power, rho, tau)

        score, sensitivity, variability = self.__quasi_score(
            derivative_mu, c_inverse, self._y_values, mu, W
        )
        new_beta = self.__update_fisher_score(sensitivity, score, beta)

        return new_beta, score, sensitivity, variability

    def __update_fisher_score(self, sensitivity, score, beta):
        """A private method that implements the second-order optimization
        algorithm Fisher-scoring.

        Parameters
        ----------
            sensitivity : array-like
                Sensitivity matrix
            score : array-like
                Quasi-score output.
            beta : array-like
                Regression parameters.

        Returns
```

```
211             -------
212                array-like : New regression vector.
213             """
214             linear_system = solve(sensitivity, score)
215             index = 0
216             adjusted_betas = []
217             for beta_value in beta:
218                adjusted_betas.append(linear_system[index : index + len
219                (beta_value)])
220                index += len(beta_value)
221             adjusted_betas = np.array(adjusted_betas)
222             new_beta = beta - adjusted_betas
223
224             return new_beta
225
226         def __quasi_score(self, mu_derivative, c_inverse, y, mu, W):
227             """Quasi-score functions are estimating equations for the Maximum
228             Likelihood Estimator (Wedderburn, 1974). This method harnesses the
229             Numpy backbone to produce a classic method of statistical models.
230
231             Parameters
232             ----------
233                mu_derivative : array-like
234                   First-order derivatives of means.
235                c_inverse : array-like
236                   Inverse of C matrix
237                y : array-like
238                   A vector with outcome variable.
239                mu : array-like
240                   A vetor with mean parameters.
241                W : array-like
242                   A matrix of weights.
243
244             Returns
245             -------
246                tuple : A tuple with score, sensitivity and variability.
247             """
248
249             residue = y - mu
250
251             mu_derivative_transpose = mu_derivative.transpose()
252             mu_derivative_and_c = np.dot(mu_derivative_transpose, c_inverse)
253
254             score = np.dot(np.dot(mu_derivative_and_c, W), residue)
255             sensitivity = np.dot(np.dot(-mu_derivative_and_c, W),
256             mu_derivative)
257             variability = np.dot(np.dot(mu_derivative_and_c, W**2),
258             mu_derivative)
259
260             return (score, sensitivity, variability)
```

Listagem A.3: Classe `MCGLMVariance`.

```
1    class MCGLMVariance(MCGLMCAttributes):
2    """
3    The MCGLMVariance class handles the second optimization of the MCGLM
4    second-moment assumptions, therefore, the step for variance. It implements
5    every step of Variance within the scope of the MCGLM algorithm, using many
```

```
    attributes to be specified as attributes. A general class must inherit
    this MCGLMVariance and leverages its methods properly. MCGLM is in charge
    of setting the fundamental python attributes and modules orchestration for
    a complete mcglm adjustment.

    The variance step on the optimization sketch boils down to Pearson
    estimating equations and the chaser algorithm for optimization. The latter
    uses tuning to set the step size of each iteration.

    Heavy operations regarding C components, pivotal to the chaser
    optimization step, are implemented on the MCGLMAttricutes class, inherited
    here. The method _c_complete, the one that crafts all of the three
    attributes thoroughly, is comprehensive for the variance calculation in
    this class.
    """

    def __init__(self):
        super(MCGLMCAttributes, self).__init__()

    def update_covariates(self, mu_attributes, rho, power, tau, W, dispersion,
    mu):
        """The method update_covariates implements a cycle of iteration for
        the second-moment estimation, the variance.

        Parameters
        ----------
            mu_attributes : dict
                A dict with mean and derivatives.
            rho : array_type
                Parameters of correlation.
            power : float
                A parameter for Power Tweedie distribution.
            tau : float
                Dispersion parameters.
            W : array_type
                A weight matrix.
            dispersion : array-type
                A vector with dispersion parameters.
            mu : array_type
                A vector with mean parameters.
        Returns
        -------
            tuple: A tuple with new vector of dispersion vector, atributes of
            matrix C, sensitivity.
        """
        c_inverse, c_derivative, c_values = self.c_complete(
            mu_attributes, power, rho, tau
        )
        (
            pearson_score,
            sensitivity,
            c_intermediate_components,
        ) = self.__pearson_estimating_equation(mu, W, c_inverse, c_derivative)

        step = self.__chaser_step(pearson_score, sensitivity)
        new_covariates = dispersion - step

        return (
```

```
            new_covariates,
            c_inverse,
            c_values,
            c_intermediate_components,
            sensitivity,
        )

    def __chaser_step(self, score, sensitivity):
        """The protected method chaser step calculates the step for a
        optimization step.

        Parameters
        ----------
            score : array_type
                A vector with quasi-score values.
            sensitivity : array_type
                The sensitivity matrix.
        Returns
        -------
            array_type: The absolute change to operate on vector.
        """
        return self._tuning * solve(sensitivity, score)

    def __pearson_estimating_equation(self, mu, W, c_inverse, c_derivative):
        """The estimating equation for the dispersion parameters.

        Parameters
        ----------
            mu : array_type
                A vetor with mean parameters
            W : array_type
                A weight matrix
            c_inverse : array_type
                The inverse of C Matrix
            c_derivative : array_type
                The derivatives of C Matrix
        Returns
        -------
            tuple : a tuple with score, sensitivity matrix and the matrix C
            normalized by Pearson.
        """
        residue = self._y_values - mu

        c_pearson = [np.dot(c_inverse, d_c) for d_c in c_derivative]

        pearson_score = [
            self.__core_pearson(matrix_component, c_inverse, residue, W)
            for matrix_component in c_pearson
        ]
        sensitivity = self.generate_sensitivity(c_pearson, W)
        return (pearson_score, sensitivity, c_pearson)

    def __core_pearson(self, c_component, c_inverse, residue, W):
        """The protected method core_pearson handles the inner-components of
        Pearson estimation equations operations.

        Parameters
        ----------
```

```
            c_component : array_type
               A matrix with components of C.
            c_inverse : array_type
               The inverse of matrix C.
            residue : array_type
               The difference between mean and an outcome variable.
            W : array_type
               A weight matrix.
         Returns
         -------
            array_type : A matrix with the core pearson.
         """
         weighted_c_components = np.dot(c_component, W)
         sum_diagonal = np.sum(np.diag(weighted_c_components))
         residue_inv_C = np.dot(c_inverse, residue)

         core_pearson = np.subtract(
            np.dot(np.dot(residue.transpose(), weighted_c_components),
            residue_inv_C),
            sum_diagonal,
         )
         return core_pearson

      @staticmethod
      def generate_sensitivity(c_intermediate_components, W):
         """The method to create the sensitivity matrix.

         Parameters
         ----------
            c_intermediate_components : array_type
               Intermediate components of C matrix.
            W : array_type
               A weight matrix.
         Returns
         -------
            array_type : A sensitivity matrix
         """
         sensitivity = np.array([])

         for position_row in range(len(c_intermediate_components)):
            transpose_matrix_position = (
               c_intermediate_components[position_row].copy().transpose()
            )
            transpose_matrix_position = np.dot(W, transpose_matrix_position)
            for position_col in range(len(c_intermediate_components)):
               sensitivity = np.append(
                  sensitivity,
                  -np.sum(
                     np.multiply(
                        transpose_matrix_position,
                        c_intermediate_components[position_col],
                     )
                  ),
               )
         sensitivity = sensitivity.reshape(
            len(c_intermediate_components), len(c_intermediate_components)
         )
         return sensitivity
```

Listagem A.4: Classe `MCGLMCAttributes`.

```
1
2  class MCGLMCAttributes:
3      """
4      The class "MCGLMCAttributes" has the sake of calculating every C
5      operations, used on throughout adjustments of mean and variance. This
6      class has two interfaces, "c_inverse" and "c_complete"; one for each of
7      two adjustment steps of MCGLM.
8
9      The interface "c_inverse" crafts only inverse C, and the "c_complete" adds
10     its derivatives and other features onto response. A Quasi-likelihood
11     estimation needs only the inverse of "C" matrix. Therefore c_inverse saves
12     computational resources by avoiding unnecessary operations on mean step
13     adjustment.
14     """
15
16     def c_inverse(self, mu, power, rho, tau, full_response=False):
17         """
18         A method to generate only the inverse of the C matrix, explicitly made
19         for the mean treatment step. This method interacts with sigma and
20         omega amenities by list of each parameter.
21
22         Parameters
23         ----------
24             mu : array_like
25                 A vetor with mean parameters.
26             power : float
27                 Power parameter.
28             rho : float
29                 Correlation parameter.
30             tau : float
31                 Dispersion parameter.
32         Returns
33         -------
34             array_like or tuple : The inverse of C matrix and its components.
35         """
36         c_inverse = self.__generate_c_inverse(mu, power, rho, tau,
37         full_response)
38         return c_inverse
39
40     def c_complete(self, mu, power, rho, tau):
41         """
42         A method to generate the whole list of C components, explicitly made
43         for the variance treatment step. This method interacts with sigma and
44         omega crafting practices, passing the list of each parameter.
45
46         Parameters
47         ----------
48             mu : array_like
49                 A vetor with mean parameters.
50             power : float
51                 Power parameter.
52             rho : float
53                 Correlation parameter.
54             tau : float
55                 Dispersion parameter.
56         Returns
57         -------
```

```
58          tuple : A tuple with every component of C.
59       """
60       (
61          diagonal_matrix,
62          omega,
63          sigma_raw,
64          sigma_chol,
65          sigma_chol_inv,
66          sigma_between,
67          sigma_between_derivative,
68          sigma_chol_block_matrix,
69          sigma_chol_inv_block_matrix,
70          c_inverse,
71       ) = self.__generate_c_inverse(mu, power, rho, tau, full_response=True)
72
73       sigma_derivatives = self.__generate_sigma_derivatives(omega, mu, power)
74
75       core_matrix = np.kron(sigma_between, diagonal_matrix)
76       sigma_chol_block_matrix_transpose = sigma_chol_block_matrix.transpose()
77       c_derivatives = self.__generate_derivative_c(
78          sigma_chol,
79          sigma_chol_inv,
80          sigma_derivatives,
81          core_matrix,
82          sigma_chol_block_matrix,
83          sigma_chol_block_matrix_transpose,
84          sigma_between_derivative,
85          diagonal_matrix,
86          self._n_targets,
87       )
88       c_values = self.__generate_c_values(
89          sigma_between,
90          diagonal_matrix,
91          sigma_chol_block_matrix_transpose,
92          sigma_chol_block_matrix,
93       )
94
95       return c_inverse, c_derivatives, c_values
96
97    def __generate_c_inverse(self, mu, power, rho, tau, full_response=False):
98       """This method generates and retrieves the inverse of the C matrix,
99       which is a pivotal component of the quasi-score calculation.
100      Notwithstanding the mentioned, this method crafts many important
101      artifacts throughout the time as omega, sigma, sigma Cholesky, the
102      inverse of Cholesky sigma, sigma among responses, sigma block diagonal
103      matrix alongside its inverse.
104      Owing to the previously mentioned worthy artifacts crafted by the
105      method, it can retrieve either only the c inverse matrix or all
106      artifacts. The former is helpful for the method _c_inverse, whereas
107      the former is for _c_complete.
108
109      Parameters
110      ----------
111         mu : array-like
112            A vetor with mean parameters.
113         power : float
114            Power parameter.
115         rho : float
```

```
            Correlation parameter.
         tau : float
            Dispersion parameter.
      Returns
      -------
         array_like : A matrix with the inverse matrix of C.
      """
      diagonal_matrix = diagonal(self._n_obs, np.ones(self._n_obs))

      omega = self._generate_omega(tau)
      build_sigma = map(
         self._calculate_sigma,
         mu,
         power,
         omega,
         self._variance,
         self._ntrial,
      )
      sigma_raw, sigma_chol, sigma_chol_inv = self.__parser_sigma
      (build_sigma)

      sigma_between, sigma_between_derivative = self._sigma_between_values(
         rho=rho, n_resp=self._n_targets
      )
      raw_c_components = self.__generate_c_inverse_and_blocks(
         sigma_chol, sigma_chol_inv, sigma_between, diagonal_matrix
      )

      (
         sigma_chol_block_matrix,
         sigma_chol_inv_block_matrix,
         c_inverse,
      ) = self.__parser_c_inverse(raw_c_components)

      if not full_response:
         return c_inverse
      else:
         return (
            diagonal_matrix,
            omega,
            sigma_raw,
            sigma_chol,
            sigma_chol_inv,
            sigma_between,
            sigma_between_derivative,
            sigma_chol_block_matrix,
            sigma_chol_inv_block_matrix,
            c_inverse,
         )

   def __generate_sigma_derivatives(self, omega, mu, power):
      """Base method to generate derivatives related to sigma.

      Parameters
      ----------
         omega : array-like
            The omega matrix.
         mu : array-like
```

```
174             A vetor with mean parameters.
175          power : float
176             The power parameter.
177       Returns
178       -------
179          _type_: _description_
180       """
181       build_sigma = map(
182          self._calculate_sigma_derivatives,
183          mu,
184          power,
185          self._variance,
186          self._z,
187          self._power_fixed,
188          self._ntrial,
189          omega,
190       )
191
192       sigma_derivative = self.__parser_sigma_derivatives(build_sigma)
193       return sigma_derivative
194
195    def __generate_derivative_c(
196       self,
197       sigma_chol,
198       sigma_chol_inv,
199       sigma_derivatives,
200       core_matrix,
201       sigma_chol_block_matrix,
202       sigma_chol_block_matrix_transpose,
203       sigma_between_derivative,
204       diagonal_matrix,
205       n_target,
206    ):
207       """Base method to generate derivatives related to C matrix.
208
209       Parameters
210       ----------
211          sigma_chol : array_like
212             Sigma matrix decomposed by the Cholesky operation.
213          sigma_chol_inv : array_like
214             Inverse sigma matrix decomposed by the Cholesky operation.
215          sigma_derivatives : array_like
216             Derivatives related to the Sigma.
217          core_matrix : array_like
218             An intermediate matrix for Sigma operation
219          sigma_chol_block_matrix : array_like
220             A block diagonal matrix with all Sigmas.
221          sigma_chol_block_matrix_transpose : array_like
222             The transpose matrix of a block diagonal matrix with all
223             Sigmas.
224          sigma_between_derivative : array_like
225             The deriviatives between sigmas.
226          diagonal_matrix : array_like
227             A diagonal matrix.
228          n_target : int
229             Total outcome variables.
230       Returns
231       -------
```

```
232            array_like : A matrix with derivatives related to the C Matrix.
233        """
234        if n_target == 1:
235
236            core_matrix_csr = sigma_between_derivative
237            sigma_chol_block_matrix_csr = sigma_chol_block_matrix
238            sigma_chol_block_matrix_transpose_csr = csr_matrix(
239                sigma_chol_block_matrix_transpose
240            )
241            sigma_between_derivative_csr = sigma_between_derivative
242
243        else:
244            core_matrix_csr = csr_matrix(core_matrix)
245            sigma_chol_block_matrix_csr = csr_matrix(sigma_chol_block_matrix)
246            sigma_chol_block_matrix_transpose_csr = csr_matrix(
247                sigma_chol_block_matrix_transpose
248            )
249            sigma_between_derivative_csr = []
250            for sigma in sigma_between_derivative:
251                sigma_between_derivative_csr.append(csr_matrix(sigma))
252
253            diagonal_matrix_csr = csr_matrix(diagonal_matrix)
254
255        list_d_chol_sigma = list(
256            map(
257                self.__derivative_cholesky,
258                sigma_derivatives,
259                sigma_chol_inv,
260                sigma_chol,
261            )
262        )
263        bdiag_d_chol_sigma = list(
264            map(
265                self.__mc_transform_list_bdiag,
266                list_d_chol_sigma,
267                list(range(n_target)),
268                [n_target for _ in range(n_target + 1)],
269            )
270        )
271        bdiag_d_chol_sigma_csr = []
272        for sigmas in bdiag_d_chol_sigma:
273            bdiag_d_chol_sigma_csr.append([csr_matrix(sigma) for sigma in
274            sigmas])
275        if n_target == 1:
276            bdiag_d_chol_sigma_csr = bdiag_d_chol_sigma_csr[0]
277            d_c = [
278                mc_sandwich_power_csr(
279                    core_matrix_csr,
280                    d_chol_sigma,
281                    sigma_chol_block_matrix_transpose_csr,
282                ).toarray()
283                for d_chol_sigma in bdiag_d_chol_sigma_csr
284            ]
285            D_C = d_c
286        else:
287            bdiag_d_chol_sigma = list(itertools.chain(*list
288            (bdiag_d_chol_sigma_csr)))
289
```

```
290            d_c = [
291               mc_sandwich_power_csr(
292                  core_matrix_csr,
293                  d_chol_sigma,
294                  sigma_chol_block_matrix_transpose_csr,
295               ).toarray()
296               for d_chol_sigma in bdiag_d_chol_sigma
297            ]
298            d_c_rho = self.__derivative_rho(
299               sigma_between_derivative_csr,
300               sigma_chol_block_matrix_csr,
301               sigma_chol_block_matrix_transpose_csr,
302               diagonal_matrix_csr,
303            )
304            D_C = d_c_rho + d_c
305         return D_C
306
307      def __generate_c_values(
308         self,
309         sigma_between,
310         diagonal_matrix,
311         sigma_chol_block_matrix_transpose,
312         sigma_chol_block_matrix,
313      ):
314         """base method to generate the C matrix.
315
316         Parameters
317         ----------
318            sigma_between : array_like
319               Matrix with inner values among sigmas.
320            diagonal_matrix : array_like
321               A diagonal matrix.
322            sigma_chol_block_matrix_transpose : array_like
323               The transpose matrix of a block diagonal matrix with all
324               Sigmas.
325            sigma_chol_block_matrix : array_like
326               A block diagonal matrix with all Sigmas.
327         Returns
328         -------
329            array_like : A matrix with derivatives related to the C Matrix.
330         """
331         kron_middle = np.kron(sigma_between, diagonal_matrix)
332
333         c_computation = np.dot(
334            np.dot(
335               sigma_chol_block_matrix_transpose,
336               kron_middle,
337            ),
338            sigma_chol_block_matrix,
339         )
340         return c_computation
341
342      def __derivative_rho(
343         self,
344         d_sigmab,
345         sigma_chol_block_matrix,
346         sigma_chol_block_matrix_transpose,
347         core_matrix,
```

```
348     ):
349         """Base method to calculate derivatives related to correlation
350         parameters.
351
352         Parameters
353         ----------
354             d_sigmab : array_like
355                 Matrix with inner values among sigmas.
356             sigma_chol_block_matrix : array_like
357                 A block diagonal matrix with all Sigmas.
358             sigma_chol_block_matrix_transpose : array_like
359                 The transpose matrix of a block diagonal matrix with all
360                 Sigmas.
361             core_matrix : array_like
362                 A diagonal matrix.
363         Returns
364         -------
365             array_like : A matrix with derivatives related to the C Matrix.
366         """
367
368         def multiplication_matrix(
369             sigmab,
370             sigma_chol_block_matrix,
371             sigma_chol_block_matrix_transpose,
372             core_matrix,
373         ):
374             kron_matrix = kron(sigmab, core_matrix)
375             return (sigma_chol_block_matrix_transpose.dot(kron_matrix)).dot(
376                 sigma_chol_block_matrix
377             )
378
379         return [
380             multiplication_matrix(
381                 sigmab,
382                 sigma_chol_block_matrix,
383                 sigma_chol_block_matrix_transpose,
384                 core_matrix,
385             ).toarray()
386             for sigmab in d_sigmab
387         ]
388
389     def _generate_omega(self, tau):
390         """Base method to calculate derivatives related to correlation
391         parameters.
392
393         Parameters
394         ----------
395             tau : list
396                 List with dispersion parameters
397         Returns
398         -------
399             list : the results of matrix linear sum between tau and dependence
400             matrices.
401         """
402         omega = []
403         for target in range(self._n_targets):
404             omega.append(mc_matrix_linear_predictor(tau=tau[target], z=self._z
405             [target]))
```

```
406
407         return omega
408
409     def __generate_c_inverse_and_blocks(
410         self,
411         sigma_chol,
412         sigma_chol_inv,
413         sigma_between,
414         diagonal_matrix,
415     ):
416         """Base method to calculate derivatives related to correlation
417         parameters.
418
419         Parameters
420         ----------
421             sigma_chol : array_like
422                 The sigma matrix decomposed by Cholesky.
423             sigma_chol_inv : array_like
424                 The inverse of sigma matrix decomposed by Cholesky.
425             sigma_between : array_like
426                 The inner-matrix among sigmas.
427             diagonal_matrix : array_like
428                 A diagonal matrix.
429         Returns
430         -------
431             array_like : A matrix with derivatives related to the C Matrix.
432         """
433         sigma_chol_block_matrix = block_diag(*sigma_chol)
434         sigma_chol_inv_block_matrix = block_diag(*sigma_chol_inv)
435
436         try:
437             ls_sigma_diagonal = inv(sigma_between)
438         except Exception as e:
439             ls_sigma_diagonal = np.array([[1]])
440
441         C_inverse = np.dot(
442             np.dot(
443                 sigma_chol_inv_block_matrix,
444                 np.kron(ls_sigma_diagonal, diagonal_matrix),
445             ),
446             sigma_chol_inv_block_matrix.transpose(),
447         )
448
449         return (
450             sigma_chol_block_matrix,
451             sigma_chol_inv_block_matrix,
452             C_inverse,
453         )
454
455     def _calculate_sigma(
456         self, mu, power, omega, variance, Ntrial, covariance="identity"
457     ):
458         """Base method to calculate sigma
459
460         Parameters
461         ----------
462             mu : array_like
463                 A vector with expected values.
```

```
464            power : float
465               A power parameter.
466            omega : array_like
467               The omega resulted matrix.
468            variance : str
469               The variance function.
470            Ntrial : int
471               The number of trial. Parameter for Binomial distribution.
472         Returns
473         -------
474            array_like : A matrix with Sigma values.
475         """
476         if isinstance(variance, list):
477            variance = variance[0]
478
479         if variance == "constant":
480            sigma_raw = omega
481            sigma_chol = cholesky(sigma_raw).T
482            sigma_chol_inv = inv(sigma_chol)
483         elif variance in ["tweedie", "binomialP", "binomialPQ"]:
484            if variance == "tweedie":
485               variance = "power"
486
487            variance_components = self.__generate_variance(
488               variance_type=variance, mu=mu.get("mu"), power=power,
489               Ntrial=Ntrial
490            )
491            sigma_raw = mc_sandwich(
492               omega,
493               variance_components.get("variance_sqrt_output"),
494               variance_components.get("variance_sqrt_output"),
495            )
496            sigma_chol = cholesky(sigma_raw).T
497            sigma_chol_inv = inv(sigma_chol)
498         elif variance in ["poisson_tweedie", "geom_tweedie"]:
499            diagonal_value = [mu.get("mu") ** 2, mu.get("mu")][
500               variance == "poisson_tweedie"
501            ]
502
503            variance_components = self.__generate_variance(
504               variance_type="power", mu=mu.get("mu"), power=power,
505               Ntrial=Ntrial
506            )
507            sigma_raw = diagonal(len(mu.get("mu")), diagonal_value) + np.dot(
508               np.dot(variance_components.get("variance_sqrt_output"), omega),
509               variance_components.get("variance_sqrt_output"),
510            )
511            sigma_chol = cholesky(sigma_raw).T
512            sigma_chol_inv = inv(sigma_chol)
513         return dict(
514            sigma_raw=sigma_raw, sigma_chol=sigma_chol,
515            sigma_chol_inv=sigma_chol_inv
516         )
517
518      def _calculate_sigma_derivatives(
519         self, mu, power, variance, z, power_fixed, Ntrial, omegas,
520         covariance="identity"
521      ):
```

```
        """
        Base method for computing variance-covariance matrix, based on
        variance function and omega matrix. This method will implement for
        cases where covariance is equal to identity, and variance falls in the
        list:
        ['constant', 'tweedie', 'binomialP', 'binomialPQ', 'power',
        'geom_tweedie', 'poisson_tweedie']

        Parameters
        ----------
           mu : array_like
              A vector with expected values.
           power : float
              A power parameter.
           variance : str
              The variance function.
           z : list
              The list with z matrices for dependencies specification.
           power_fixed : boolean
              The specification of power estimation.
           Ntrial : int
              The number of trial. Parameter for Binomial distribution.
           omegas : list
              a list with omegas.
        """
        if isinstance(variance, list):
           variance = variance[0]
        sigma_derivative = None

        if variance == "constant":
           sigma_derivative = z
        elif variance in ["tweedie", "binomialP", "binomialPQ"]:
           variance_type = variance if variance != "tweedie" else "power"

           variance_components = self.__generate_variance(
              variance_type=variance_type, mu=mu.get("mu"), power=power,
              Ntrial=Ntrial
           )
           sigma_derivative = [
              mc_sandwich(
                 d_omega,
                 variance_components.get("variance_sqrt_output"),
                 variance_components.get("variance_sqrt_output"),
              )
              for d_omega in z
           ]
           if not power_fixed:
              if variance in ["tweedie", "binomialP"]:

                 sigma_derivative_power = mc_sandwich_power(
                    omegas,
                    variance_components.get("variance_sqrt_output"),
                    variance_components.get
                    ("derivative_variance_sqrt_power"),
                 )
                 sigma_derivative.insert(0, sigma_derivative_power)
                 #sigma_derivative.append(sigma_derivative_power)
```

```
            elif variance == "binomialPQ":
               sigma_derivative_p = mc_sandwich(
                  omegas,
                  variance_components.get("variance_sqrt_output"),
                  variance_components.get("derivative_variance_sqrt_p"),
               )
               sigma_derivative_q = mc_sandwich(
                  omegas,
                  variance_components.get("variance_sqrt_output"),
                  variance_components.get("derivative_variance_sqrt_q"),
               )
               sigma_derivative = [
                  sigma_derivative,
                  sigma_derivative_p,
                  sigma_derivative_q,
               ]
         elif variance in ["poisson_tweedie", "geom_tweedie"]:
            variance_components = self.__generate_variance(
               variance_type="power", mu=mu.get("mu"), power=power,
               Ntrial=Ntrial
            )
            sigma_derivative = [
               mc_sandwich(
                  d_omega,
                  variance_components.get("variance_sqrt_output"),
                  variance_components.get("variance_sqrt_output"),
               )
               for d_omega in z
            ]
            if not power_fixed:
               sigma_derivative_power = mc_sandwich_power(
                  omegas,
                  variance_components.get("variance_sqrt_output"),
                  variance_components.get("derivative_variance_sqrt_power"),
               )
               sigma_derivative.insert(0, sigma_derivative_power)
         return dict(sigma_derivative=sigma_derivative)

      def _sigma_between_values(self, rho, n_resp):
         def forcesymmetric(sigmab):
            sigmab_t = sigmab.transpose()
            sigmabsymmetric = np.matmul(sigmab_t, sigmab_t.transpose())
            diagonal_value = sigmabsymmetric.diagonal()
            if np.max(diagonal_value) == np.min(diagonal_value):
               return sigmabsymmetric
            else:
               sigmabsymmetric = sigmab
               sigmabsymmetric[np.triu_indices(n_resp, k=1)] = rho
               return sigmabsymmetric

         """sigma between method computes between for sequence calculations.

         It responds out with 2-position-tuple with sigma between and its
         derivative.
         """
         if n_resp == 1:
            return (1, 1)
         else:
```

```
638         sigmab = diagonal(n_resp, np.full(n_resp, 1))
639         sigmab[np.tril_indices(n_resp, k=-1)] = rho
640         sigmab = forcesymmetric(sigmab)
641
642         d_sigmab = self._mc_derivative_sigma_between(n_resp=n_resp)
643         return (sigmab, d_sigmab)
644
645     @lru_cache(maxsize=64)
646     def _mc_derivative_sigma_between(self, n_resp):
647        list_derivatives = list()
648        position = list(combinations(range(n_resp), 2))
649        n_par = int(n_resp * (n_resp - 1) / 2)
650        for index in range(n_par):
651           derivative = np.zeros((n_resp, n_resp))
652           derivative[position[index][0], position[index][1]] = 1
653           derivative[position[index][1], position[index][0]] = 1
654           list_derivatives.append(derivative)
655
656        return list_derivatives
657
658     def __derivative_cholesky(self, d_sigma, inv_chol, chol):
659        def faux(d_sigma, inv_chol, chol, inv_chol_transpose):
660           csr_d_sigma_matrix = csr_matrix(d_sigma)
661           csr_inv_chol_matrix = csr_matrix(inv_chol)
662           csr_inv_chol_transpose_matrix = csr_matrix(inv_chol_transpose)
663
664           matrix_operations = csr_inv_chol_matrix.dot(csr_d_sigma_matrix)
665           matrix_operations = matrix_operations.dot
666           (csr_inv_chol_transpose_matrix)
667
668           matrix_operations = tril(matrix_operations)
669           matrix_operations = matrix_operations.toarray()
670
671           diagonal_indices = np.diag_indices_from(matrix_operations)
672           matrix_operations[diagonal_indices] = (
673              matrix_operations[diagonal_indices] / 2
674           )
675           return np.dot(chol, matrix_operations)
676
677        cholesky_element = []
678        inv_chol_transpose = inv_chol.transpose()
679        for derivative in d_sigma:
680           if not isinstance(derivative, list):
681              cholesky_element.append(
682                 faux(derivative, inv_chol, chol, inv_chol_transpose)
683              )
684           else:
685              for der in derivative:
686                 cholesky_element.append(
687                    faux(der, inv_chol, chol, inv_chol_transpose)
688                 )
689
690        if isinstance(derivative, list):
691           cholesky_element.reverse()
692        return cholesky_element
693
694     def __mc_transform_list_bdiag(self, list_mat, response_number,
695     number_of_labels):
```

```
696         def build_block_diag(value, response_number):
697             block_value = block_diag(value)
698 
699             matrix_base = np.zeros(
700                 (
701                     block_value.shape[0] * number_of_labels,
702                     block_value.shape[1] * number_of_labels,
703                 )
704             )
705 
706             lower = block_value.shape[0] * response_number
707             upper = block_value.shape[1] + block_value.shape[1] *
708             response_number
709 
710             matrix_base[lower:upper, lower:upper] = block_value
711 
712             return matrix_base
713 
714         output = [build_block_diag(d_chol, response_number) for d_chol in
715         list_mat]
716         return output
717 
718     def _generate_variance(
719         self, variance_type: str, mu: np.array, power: int = 1, Ntrial: list =
720         1
721     ):
722         return self.__generate_variance(variance_type, mu, power, Ntrial)
723 
724     def __generate_variance(self, variance_type, mu, power=1, Ntrial=1):
725         """Method to apply the variance function on the mean values.
726 
727         Parameters
728         ----------
729             variance_type : array_like
730                 Type of variance
731             mu : array_like
732                 A vector with expected values.
733             power : float
734                 A power parameter.
735             Ntrial : int
736                 The number of trial. Parameter for Binomial distribution.
737         Returns
738         -------
739             array_like : A matrix with Sigma values.
740         """
741         if variance_type == "power":
742             return self.__power_variance(mu, power)
743         elif variance_type == "binomialP":
744             return self.__binomialp_variance(mu, power, Ntrial)
745         elif variance_type == "binomialPQ":
746             return self.__binomialpq_variance(mu, power, Ntrial)
747 
748     def __power_variance(self, mu, power):
749         mu_power = mu**power
750         sqrt_mu_power = np.sqrt(mu_power)
751         n_len = len(mu)
752 
753         variance_sqrt_output = diagonal(n=n_len, values=sqrt_mu_power)
```

```
754         derivative_variance_sqrt_power = diagonal(
755             n=n_len, values=((mu_power * np.log(mu)) / (2 * sqrt_mu_power))
756         )
757         derivative_variance_sqrt_mu = (mu ** (power - 1) * power) / (2 *
758         sqrt_mu_power)
759 
760         return dict(
761             variance_sqrt_output=variance_sqrt_output,
762             derivative_variance_sqrt_power=derivative_variance_sqrt_power,
763             derivative_variance_sqrt_mu=derivative_variance_sqrt_mu,
764         )
765 
766     def __binomialp_variance(self, mu, power, ntrial):
767         constant = 1 / ntrial
768         mu_power = mu**power
769         mu_power1 = (1 - mu) ** power
770         mu1mu = constant * (mu_power * mu_power1)
771         sqrt_mu1mu = np.sqrt(mu1mu)
772         n_len = len(mu)
773 
774         variance_sqrt_output = diagonal(n=n_len, values=sqrt_mu1mu)
775         derivative_variance_sqrt_power = diagonal(
776             n=n_len,
777             values=(np.log(1 - mu) * mu1mu + np.log(mu) * mu1mu) / (2 *
778             sqrt_mu1mu),
779         )
780         derivative_variance_sqrt_mu = (
781             constant * (mu_power1 * (mu ** (power - 1)) * power)
782             - constant * (((1 - mu) ** (power - 1)) * mu_power * power)
783         ) / (2 * sqrt_mu1mu)
784 
785         return dict(
786             variance_sqrt_output=variance_sqrt_output,
787             derivative_variance_sqrt_power=derivative_variance_sqrt_power,
788             derivative_variance_sqrt_mu=derivative_variance_sqrt_mu,
789         )
790 
791     def __binomialpq_variance(self, mu, power, ntrial):
792         constant = 1 / ntrial
793         p = power[0]
794         q = power[1]
795 
796         mu_p = mu**p
797         mu1_q = (1 - mu) ** q
798         mu_p_mu_q = mu_p * mu1_q
799         mu1mu = mu_p_mu_q * constant
800         sqrt_mu1mu = np.sqrt(mu1mu)
801         n_len = len(mu)
802 
803         denominator1 = 2 * sqrt_mu1mu
804         denominator2 = denominator1 * ntrial
805 
806         variance_sqrt_output = diagonal(n=n_len, values=sqrt_mu1mu)
807 
808         derivative_variance_sqrt_p = diagonal(
809             n=n_len, values=(mu_p_mu_q * np.log(mu)) / denominator2
810         )
811 
```

```
812         derivative_variance_sqrt_q = diagonal(
813             n=n_len, values=(mu_p_mu_q * np.log(1 - mu)) / denominator2
814         )
815
816         derivative_variance_sqrt_mu = (
817             constant * (mu1_q * (mu ** (p - 1)) * p)
818             - constant * ((1 - mu) ** (q - 1) * mu_p * q)
819         ) / denominator1
820
821         return dict(
822             variance_sqrt_output=variance_sqrt_output,
823             derivative_variance_sqrt_p=derivative_variance_sqrt_p,
824             derivative_variance_sqrt_q=derivative_variance_sqrt_q,
825             derivative_variance_sqrt_mu=derivative_variance_sqrt_mu,
826         )
827
828     def __parser_sigma(self, build_sigma_map):
829         """parser method for _calculate_sigma output attributes
830         Args:
831             build_sigma_map (list): a list with dicts, with sigmas
832         """
833         list_sigmas = [
834             [
835                 build_sigma_iteration.get("sigma_raw"),
836                 build_sigma_iteration.get("sigma_chol"),
837                 build_sigma_iteration.get("sigma_chol_inv"),
838             ]
839             for build_sigma_iteration in build_sigma_map
840         ]
841
842         sigma_raw = [sigmas[0] for sigmas in list_sigmas]
843         sigma_chol = [sigmas[1] for sigmas in list_sigmas]
844         sigma_chol_inv = [sigmas[2] for sigmas in list_sigmas]
845
846         return (sigma_raw, sigma_chol, sigma_chol_inv)
847
848     def __parser_sigma_derivatives(self, sigma_derivatives_map):
849         list_sigmas = [
850             [build_sigma_iteration.get("sigma_derivative")]
851             for build_sigma_iteration in sigma_derivatives_map
852         ]
853         sigma_derivative = [sigmas[0] for sigmas in list_sigmas]
854         return sigma_derivative
855
856     def __parser_c_inverse(self, raw_c_components):
857
858         sigma_chol_block_matrix = raw_c_components[0]
859         sigma_chol_inv_block_matrix = raw_c_components[1]
860         c_inverse = raw_c_components[2]
861
862         return (sigma_chol_block_matrix, sigma_chol_inv_block_matrix,
863         c_inverse)
```

Listagem A.5: Classe `MCGLMParameters`.

```
1
2   class MCGLMParameters:
3   """According to MCGLM specification, grounded for frequentist inference
```

```
    traits, the estimation of resulting parameters converge asymptotically to
    a gaussian distribution with tuple mean-variance = (actual parameters,
    inverse of matrix Godambe). This property allows the calculation of
    pivotal traits regarding the parameters, such as: hypothesis testing and
    confidence interval.

    This class implements every method related to this trait.
    """

    @staticmethod
    def _parameters_attributes(
        resid,
        mu_deriv,
        W,
        quasi_score,
        mean_sens,
        mean_varia,
        c_inv,
        c,
        c_deriv,
        var_sensi,
    ):
        """The parameters of MCGLM converge assymtoptically to a Normal
        Distribution. This trait allows some statistical inferences as
        hypothesis testing, confidence interval and so on. This method crafts
        three important matrices: Varcov: variance covariance matrix,
        joint_inv_sensitivity: inverse sensitivity, and joint_variability: the
        joint variability distribution."""

        variance_variability = MCGLMParameters.generate_var_variability(
            resid, W, c_inv, c, c_deriv
        )

        inv_cw = np.dot(c_inv, W)

        s_cov_beta = MCGLMParameters._mc_cross_sensitivity(
            cov_product=c_deriv,
            columns_size=len(quasi_score),
        )

        v_cov_beta = MCGLMParameters._mc_cross_variability(
            cov_product=c_deriv,
            inv_cw=inv_cw,
            res=resid,
            d=mu_deriv,
        )

        p1 = np.append(mean_varia, v_cov_beta.transpose(), axis=0)

        p2 = np.append(v_cov_beta, variance_variability, axis=0)
        joint_variability = np.append(p1, p2, axis=1)

        inv_J_beta = inv(mean_sens)
        inv_S_beta = inv_J_beta
        inv_S_cov = inv(var_sensi)
        mat0 = np.zeros((s_cov_beta.shape[1], s_cov_beta.shape[0]))

        cross_term = mc_sandwich(s_cov_beta, -inv_S_cov, inv_S_beta)
```

```
        p1 = np.append(inv_S_beta, cross_term, axis=0)
        p2 = np.append(mat0, inv_S_cov, axis=0)

        joint_inv_sensitivity = np.append(p1, p2, axis=1)
        varcov = mc_sandwich(
            joint_variability, joint_inv_sensitivity, joint_inv_sensitivity.
            transpose()
        )

        return (
            varcov,
            joint_inv_sensitivity,
            joint_variability,
        )

    def _mc_cross_sensitivity(cov_product, columns_size):
        nrow = len(cov_product)
        return np.zeros((nrow, columns_size))

    def generate_var_variability(res, w, c_inv, c_val, c_comp):

        return MCGLMParameters._calculate_variability(
            product=c_comp,
            inv_C=c_inv,
            C=c_val,
            res=res,
            W=w,
        )

    def _calculate_variability(product, inv_C, C, res, W):
        n_par = len(product)
        we = [np.dot(product[index], inv_C) for index in range(n_par)]

        k4 = res**4 - 3 * np.diag(C) ** 2
        sensitivity = MCGLMVariance.generate_sensitivity(product, W=W**2)
        W = np.diag(W).flatten()

        variability = MCGLMParameters._mc_variability(sensitivity, we, k4, W)

        return variability

    def _mc_variability(sensitivity, we, k4, w):
        variability = np.array([])
        for position_row in range(len(we)):
            wi = np.diag(we[position_row])
            for position_col in range(len(we)):
                wj = np.diag(we[position_col])

                k4_operation = np.sum(k4 * wi * w * wj * w)

                variability = np.append(
                    variability,
                    -2 * sensitivity.item((position_row, position_col)) +
                    k4_operation,
                )

        return variability.reshape((len(we), len(we)))

```

```
120     def _covprod(a, w, res):
121         calculation_sandwich = np.dot(np.dot(res, w), res)
122         calculation_residue = np.dot(res.transpose(), a)
123         product = np.dot(calculation_sandwich, calculation_residue)
124         return product
125 
126     def _mc_cross_variability(cov_product, inv_cw, res, d):
127 
128         wlist = [np.dot(cov, inv_cw) for cov in cov_product]
129         a = np.dot(d.transpose(), inv_cw)
130 
131         n_beta = a.shape[0]
132         n_cov = len(cov_product)
133         cross_variability = []
134         for cov in range(n_cov):
135             for beta in range(n_beta):
136                 cross_variability.append(
137                     MCGLMParameters._covprod(a[beta, :], wlist[cov], res)
138                 )
139         cross_variability = np.array(cross_variability).reshape(n_cov, n_beta).
140         T
141         return cross_variability
```

Listagem A.6: Classe `MCGLMResults`.

```
1 
2     class MCGLMResults(GLMResults):
3     """MCGLM Class for generating and manipulating results of mcglm training.
4     The main output goes by the method summary(), the classical statsmodels
5     output. Therefore, the user can access the attributes "aic", "bic" e
6     loglikelihood.
7 
8     Args:
9         GLMResults: Class of statsmodels library for presenting results of GLM.
10    """
11 
12    def __init__(
13        self,
14        normalized_var_cov,
15        nobs,
16        n_targets,
17        y_names,
18        regression,
19        dispersion,
20        n_iter,
21        residue,
22        rho,
23        tau,
24        power,
25        link,
26        variance,
27        power_fixed,
28        p_log_likelihood,
29        aic,
30        bic,
31        df_resid,
32        df_model,
33        mu,
```

```
34          y_values,
35          X,
36          ntrial,
37      ):
38          self._normalized_var_cov = normalized_var_cov
39          self._nobs = nobs
40          self._n_targets = n_targets
41          self._y_names = y_names
42          self._regression = regression
43          self._dispersion = dispersion
44          self._n_iter = n_iter
45          self._residue = residue
46          self._rho = rho
47          self._tau = tau
48          self._power_list = power
49          self._link_list = link
50          self._variance_list = variance
51          self._power_fixed_list = power_fixed
52          self._p_log_likelihood = p_log_likelihood
53          self._p_aic = aic
54          self._p_bic = bic
55          self._df_resid = df_resid
56          self._df_model = df_model
57          self._mu_value = mu
58          self._y_values = y_values
59          self._X = X
60          self._ntrial = ntrial
61          self.params = None
62          #self.bse = None
63
64          self._use_t = False
65          self.model = None
66
67      @property
68      def aic(self):
69          return self._p_aic
70
71      @property
72      def bic(self):
73          return self._p_bic
74
75      @property
76      def loglikelihood(self):
77          return self._p_log_likelihood
78
79      @property
80      def bse(self):
81          """The standard errors of the parameter estimates."""
82          if (not hasattr(self, "cov_params_default")) and (
83              self.normalized_cov_params is None
84          ):
85              bse_ = np.empty(len(self.params))
86              bse_[:] = np.nan
87          else:
88              bse_ = np.sqrt(np.diag(self.normalized_cov_params))
89          return bse_
90
91      @property
```

```
92      def tvalues(self):
93          """
94          Return the t-statistic for a given parameter estimate.
95          """
96          return self.params / self.bse
97
98      @property
99      def mu(self):
100         return self._mu_value
101
102     @property
103     def pvalues(self):
104         """The two-tailed p values for the t-stats of the params."""
105         if self.use_t:
106             df_resid = getattr(self, "df_resid_inference", self._df_resid)
107             return stats.t.sf(np.abs(self.tvalues), df_resid) * 2
108         else:
109             return stats.norm.sf(np.abs(self.tvalues)) * 2
110
111     @property
112     def pearson_residuals(self):
113         """
114         Pearson residuals. The Pearson residuals are defined as
115         (`endog` - `mu`)/sqrt(VAR(`mu`)) where VAR is the distribution
116         specific variance function. See statsmodels.families.family and
117         statsmodels.families.varfuncs for more information.
118         """
119         residuals = []
120         if self._n_targets == 1:
121             mu = [self.mu]
122         else:
123             mu = self.mu.reshape(self._n_targets, -1)
124
125         for index in range(self._n_targets):
126             residue = (
127                 self._y_values[index * self._nobs : index * self._nobs + self.
128                 _nobs]
129                 - mu[index]
130             )
131
132             variance = None
133             if self._variance_list[index] in ("binomialP", "binomialPQ"):
134                 variance = self._variance_list[index]
135             else:
136                 variance = "power"
137
138             variance_sqrt_output = (
139                 MCGLMCAttributes()
140                 ._generate_variance(
141                     variance,
142                     mu[index],
143                     self._power_list[index],
144                     self._ntrial[index],
145                 )
146                 .get("variance_sqrt_output")
147             )
148
149             residuals.append(residue / np.diag(variance_sqrt_output))
```

```
150
151         return residuals
152
153     @property
154     def vcov(self):
155         return self.normalized_cov_params
156
157     def anova(
158         self, indexes_covariates=[[1, 2, 2, 2, 2]], covariate_name=[["x1",
159         "x2"]]
160     ):
161
162         total_anovas = list()
163         for position_target in range(self.model.n_targets):
164             dispersion = self.model.dispersion[position_target]["scalelist"]
165             [1:].copy()
166             dispersion_vcov = self._dispersion_vcov[position_target][1:, 1:].
167             copy()
168
169             covariates_positions = [
170                 list(j) for i, j in groupby(indexes_covariates
171                 [position_target])
172             ]
173             covariates_names = covariate_name[position_target]
174
175             index = 0
176             index_name = 0
177             anovas = list()
178             for covs in covariates_positions:
179
180                 current_cov = np.array(dispersion[index : index + len(covs)])
181                 current_vcov = dispersion_vcov[
182                     index : index + len(covs), index : index + len(covs)
183                 ]
184
185                 try:
186                     solve_vcoc = inv(current_vcov)
187                 except Exception as e:
188                     solve_vcoc = current_vcov
189
190                 chi_square = np.dot(
191                     current_cov.transpose(), np.dot(solve_vcoc, current_cov)
192                 )
193                 df = len(covs)
194                 pvalue = ncx2.pdf(chi_square, df, nc=0)
195
196                 index += len(covs)
197                 anovas.append(
198                     {
199                         "covariate_name": covariates_names[index_name],
200                         "chi_square": round(chi_square, 4),
201                         "df": df,
202                         "pvalue": round(pvalue, 4),
203                     }
204                 )
205
206                 index_name += 1
207             total_anovas.append(anovas)
```

```
208
209         return total_anovas
210
211     def __add_table_two_columns(
212         self, summ, yname="output", xname=None, title=None, alpha=0.05
213     ):
214         top_left = [
215             ("Dep. Variable:", None),
216             ("Model:", ["MCGLM"]),
217             ("link:", [self.link]),
218             ("variance:", [self.variance]),
219             ("Method:", ["Quasi-Likelihood"]),
220             ("Date:", None),
221             ("Time:", None),
222         ]
223
224         top_right = [
225             ("No. Iterations:", [self._n_iter]),
226             ("No. Observations:", [self._nobs]),
227             ("Df Residuals:", [self._df_resid]),
228             ("Df Model:", [self._df_model]),
229             ("Power-fixed:", [self.power_fixed]),
230             ("pAIC", [self.paic]),
231             ("pBIC", [self.pbic]),
232             ("pLogLik", [round(self._p_log_likelihood, 4)]),
233         ]
234
235         if hasattr(self, "cov_type"):
236             top_left.append(("Covariance Type:", [self.cov_type]))
237
238         if title is None:
239             title = "Multivariate Covariance Generalized Linear Model"
240
241         summ.add_table_2cols(
242             self,
243             gleft=top_left,
244             gright=top_right,
245             yname=yname,
246             xname=xname,
247             title=title,
248         )
249         summ.add_table_params(
250             self, yname=yname, xname=xname, alpha=alpha, use_t=self.use_t
251         )
252
253         return summ
254
255     def __add_dispersion(self, summ, title=None, alpha=0.05):
256         summ.add_table_params(
257             self,
258             alpha=alpha,
259             xname=["dispersion_" + str(i + 1) for i in range(len(self.
260             params))],
261             use_t=self.use_t,
262         )
263
264         return summ
265
```

```
    def __add_power(self, summ, alpha=0.05):
        summ.add_table_params(
            self,
            alpha=alpha,
            xname=[
                "power_" + (str(i + 1) if len(self.params) > 1 else "")
                for i in range(len(self.params))
            ],
            use_t=self.use_t,
        )

        return summ

    def __add_rho_section(self, summ, alpha=0.05):
        self.normalized_cov_params = self._rho_vcov
        self.params = np.array(self._rho) if isinstance(self._rho, int) else
        self._rho
        if isinstance(self._rho, (np.ndarray, np.generic)):
            summ.add_table_params(
                self,
                alpha=alpha,
                xname=[
                    "rho_" + (str(i + 1) if len(self.params) > 1 else "")
                    for i in range(len(self.params))
                ],
                use_t=self.use_t,
            )

            if hasattr(self, "constraints"):
                summ.add_extra_txt(
                    [
                        "Model has been estimated subject to linear "
                        "equality constraints."
                    ]
                )
        return summ

    def summary(self, yname=None, xname=None, title=None, alpha=0.05):
        """
        It generates the summary report as the sketch of classical
        "statsmodels" library. The summary shows all parameters found
        thoroughly, for each response.
        """
        self._dispersion_vcov = []
        self.betas_vcov = []
        self._rho_vcov = []
        self.power_vcov = []

        index_position = 0
        cov_params = self._normalized_var_cov

        if self._n_targets == 1:
            self._rho_vcov = np.array([np.nan])
            mu = [self.mu]
        else:
            mu = self.mu

        for index in range(self._n_targets):
```

```
324
325            n_betas = len(self._regression[index])
326            self.betas_vcov.append(
327               cov_params[
328                  index_position : index_position + n_betas,
329                  index_position : index_position + n_betas,
330               ]
331            )
332            index_position += n_betas
333
334         if self._n_targets > 1:
335            self._rho_vcov = cov_params[
336               index_position : index_position + len(self._rho),
337               index_position : index_position + len(self._rho),
338            ]
339
340            index_position += len(self._rho)
341
342         for index in range(self._n_targets):
343
344            if self._power_fixed_list[index]:
345               self.power_vcov.append(np.array([np.nan]))
346            else:
347               self.power_vcov.append(
348                  np.array(
349                     cov_params[
350                        index_position : index_position + 1,
351                        index_position : index_position + 1,
352                     ]
353                  )
354               )
355               index_position += 1
356
357            n_dispersion = len(self._dispersion[index]["scalelist"])
358
359            self._dispersion_vcov.append(
360               cov_params[
361                  index_position : index_position + n_dispersion,
362                  index_position : index_position + n_dispersion,
363               ]
364            )
365
366            index_position += n_dispersion
367
368         self.scale = 1
369
370         smry = Summary()
371
372         for target_index in range(self._n_targets):
373            exog_names = self._X[target_index].columns.tolist()
374            y_name = self._y_names[target_index]
375
376            self.link = self._link_list[target_index]
377            self.variance = self._variance_list[target_index]
378            self.power_fixed = self._power_fixed_list[target_index]
379            self.params = self._regression[target_index]
380            self.normalized_cov_params = self.betas_vcov[target_index]
381
```

```
382             self.loglikehood = self._p_log_likelihood
383             self.paic = self._p_aic
384             self.pbic = self._p_bic
385 
386             smry = self.__add_table_two_columns(smry, yname=y_name,
387             xname=exog_names)
388 
389             self.params = np.array(self._dispersion[target_index]["scalelist"])
390             self.normalized_cov_params = self._dispersion_vcov[target_index]
391 
392             #self.bse = np.sqrt(np.diag(self.normalized_cov_params))
393             smry = self.__add_dispersion(smry)
394             self.params = np.array([self._dispersion[target_index]["power"]])
395             self.normalized_cov_params = self.power_vcov[target_index]
396 
397             smry = self.__add_power(smry)
398 
399         smry = self.__add_rho_section(smry)
400 
401         return smry
```