Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Estatística

Notas em Computação Científica e Estatística - Primeiro Semestre de 1999

M. V. Castilho, M. G. F. Costa, W. J. Cunha, F. F. Ferreira, J. H. Ferreira, O. F. Neves, M. D. Oliveira, C. B. Sousa e D. D. Vianna

Relatório Técnico RTE-01/99

Relatório Técnico
Série Ensino

Sumário

Prefácio

O presente relatório técnico contém alguns trabalhos selecionados, desenvolvidos dentro da disciplina *Computação Científica e Estatística I*, no primeiro semestre de 1999, e apresentados como seminários.

O primeiro deles apresenta uma biblioteca para simulação de dados provenientes de análises químicas, feitas por dois laboratórios diferentes. A biblioteca pode ser usada em problemas Monte Carlo, para testar a eficiência de metodologias estatísticas, para comparação destes dois laboratórios.

O segundo apresenta uma biblioteca para geração de uma estatística C, que fornece uma medida de incerteza no posto (*ranking*) de um indicador de desempenho. Tal índice é calculado usando-se a técnica *boostrap*, objeto do terceiro trabalho, e lá tratada em detalhes.

O quarto trabalho lida com a área de confiabilidade, apresentando o estudo de uma implementação do estimador de Kaplan-Meier para a função de sobrevivência.

O quinto trabalho apresenta um estudo preliminar sobre o uso de redes neurais na modelagem de dispositivos elétricos.

O sexto trabalho apresenta uma biblioteca para previsão de séries temporais, por uma metodologia alternativa, vinda da área de estatística espacial, não muito usual para este tipo de problema.

O sétimo trabalho apresenta uma biblioteca para geração de números aleatórios. Sendo o mais básico de todos, não por acaso, é referenciado por quatro dos trabalhos aqui apresentados.

Finalmente, o oitavo trabalho descreve os métodos de integração numérica Simpson e Monte Carlo e propõe um estudo comparativo entre estas duas técnicas.

Estes trabalhos, todos abordando interessantes tópicos em computação científica e estatística, primam por indicar incitantes extensões.

Assim, desejo a todos, muito bom proveito.

Frederico R. B. Cruz
Organizador

Belo Horizonte, julho de 1999

Geração de Par de Vetores Aleatórios para Simulação em Estudo de Comparação de Métodos de Análise Química

Márcio V. Castilho[1]

[1]Companhia Vale do Rio Doce,
Rodovia BR 262 - km 296,
Caixa Postal 09,
33030-970 - Santa Luzia - MG
E-mail: mveloso@cvrd.com.br

Resumo

Apresenta-se um programa de computador para gerar dados que simulem um problema de comparação de métodos de análise química. Muitos trabalhos existentes apresentam métodos estatísticos para o tratamento deste problema, e a simulação é usada como ferramenta para testar a eficiência dos vários métodos propostos. Nota-se entretanto, que os programas de simulação citados na literatura não são aplicáveis a muitas situações reais, o que impede de se concluir sobre importantes aplicações práticas dos citados métodos estatísticos. O presente trabalho apresenta as considerações mais importantes ao se gerar aleatoriamente vetores de dados para simular o problema em questão, mostrando sua implementação em C++ e um exemplo de teste do programa.

1 Introdução

O problema de comparação de métodos de análise química vem sendo estudado há muito tempo. Na área clínica, por exemplo existem publicações datadas de 1965 [1] propondo procedimentos estatísticos para este fim. A literatura mais recente cita um procedimento de regressão proposto por Deming [2] que foi primeiramente apresentado em um livro de 1943. Riu e Rius apresentaram uma alternativa que propõe ser bastante completa [3], pois além de considerar os erros nas variáveis (coisa que o método de Deming trata com sucesso), também considera a heteroscedasticidade da variância, que é um fato relevante em todo o amplo campo da análise química.

Uma das formas mais aceitas atualmente de se verificar o desempenho de diferentes estimadores como os citados acima, é a simulação. Muitas vezes isto vem sendo feito utilizando geradores de números aleatórios, que são parte integrante de uma grande quantidade de programas estatísticos disponíveis no mercado. Entretanto, para o fim específico aqui aludido, torna-se necessário utilizar programas mais sofisticados, pois o problema em questão possui uma série de peculiaridades que precisam ser levadas em conta para que a simulação seja a mais realista possível.

O trabalho aqui desenvolvido fornece um programa de simulação necessária para o teste das diversas abordagens existentes na literatura a respeito. Para tanto, ele deverá produzir um par de vetores que irá representar dois métodos de análise distintos. Este programa poderá depois ser utilizado em conjunto com programas específicos para cada estimador desejado, de forma a se poder comparar seu desepenho. Uma possível maneira de proceder a isto seria através de um simulador de Monte Carlo.

O presente artigo apresenta o problema, ou seja, que considerações são necessárias para que os vetores a produzir sejam semelhantes aos obtidos em situações reais, propõe uma lógica para se atingir este objetivo, e apresenta as classes e métodos necessários à implementação do programa utilizando a liguagem C++.

Ao final, o programa 'main' apresentado demonstra um exemplo de uso do programa desenvolvido para satisfazer as condições citadas.

2 O Problema da Simulação

Muitos trabalhos apresentados ao longo dos anos para tratar o problema de comparação de laboratórios ou métodos de análise química fazem uso da simulação por meio de geradores de números aleatórios.

Riu e Rius [3], por exemplo, apresentam uma proposta de método testada por simulação de três bancos de dados. Esta simulação, entretanto, não contempla a possibilidade de os erros aleatórios serem muito diferentes entre os dois conjuntos de dados testados. Também não permite testar os procedimentos comparados no artigo [3] para o caso em que os erros de ambos os métodos são grandes, do ponto de vista prático. Em seu artigo, Cornbleet e Gochman [4], por sua vez assumem erros relativos constantes, e exploram mais as possíveis variações para este erro, e seu efeito no desempenho de alguns estimadores estatísticos. Sendo assim, este trabalho somente inclui um tipo de relacionamento do erro com a concentração química da substância, exluindo importantes relações funcionais do erro aleatório encontradas na prática, como por exemplo a relação linear, a função proposta por Rocke e Lorenzato [5], ou ainda a famosa "trompa" de Horwitz [6]. A modelagem do erro também é utilizada de forma aproximada e incompleta em Hartmann *et al.* [7]. Sarabia *et al.* [8] fazem uma simulação mais completa que as anteriores, porém com pouca flexibilidade para se concluir sobre os fatores que afetam os estimadores estudados, e ainda é limitada quanto à possibilidade de simular dados de uma distribuição assimétrica, como frequentemente encontramos em dados de pesquisa geológica (área de pesquisa do autor do presente artigo).

Um simulador que considere todos os fatores citados acima precisa contemplar os passos descritos a seguir.

3 Especificação do Programa

O programa assumirá como desenvolvida uma função capaz de gerar números aleatórios que seguem as distribuições uniforme, normal, log-normal, e gamma, uma vez especificados os seus parâmetros.

As entradas necessárias ao seu funcionamento são:

1. Semente (seed) para gerar números aleatórios utilizando biblioteca específica para este fim;
2. Tamanho (n) dos vetores aleatórios;
3. Limites inferior e superior para o vetor X;
4. Forma da distribuição de probabilidades para o vetor X, a escolher entre uniforme, normal, log-normal e gamma;
5. Parâmetros da reta que relaciona Y com X;
6. Relação matemática entre o erro de X e o valor de X. Esta poderá ser de três tipos: linear, potência, e função R-L (proposta por Rocke e Lorenzato [5]);
7. Analogamente, a relação matemática que relaciona o erro de Y com o valor de Y;
8. Parâmetros para as funções de erro citadas acima, para X e para Y.

A saída do programa será o par de vetores (X, Y), de igual tamanho n. Para efeito da avaliação a ser feita posteriormente com este par de vetores, também será útil que como saída se tenham os parâmetros da reta que relaciona um com o outro (item 5 acima).

4 Estruturação Lógica

Provavelmente existem várias maneiras diferentes de se produzir os vetores desejados. A abordagem a ser tomada aqui seguirá os seguintes passos:

1. Geração do vetor X

   Como os resultados analíticos de dois métodos distintos apresentam uma dependência linear, a simulação pode ser realizada simplesmente gerando o primeiro vetor, e supondo uma relação linear conhecida, calcula-se o segundo.

   Nos casos reais, deseja-se comparar dois métodos dentro de uma faixa de teores especificada (normalmente a faixa de interesse econômico, ou interesse técnico que se deseja estudar), por isso aqui se pede como entradas os limites do vetor a ser produzido. O uso da semente geradora de números aleatórios aqui especificada como entrada permite a reprodução dos resultados obtidos por qualquer pessoa interessada. Também a distribuição de probabilidades dos dados X aqui gerados é importante. Existem casos práticos em que estes resultados são marcadamente assimétricos, e se aproximam bem das distribuições log-normal e gamma. Já em experimentos planejados, é comum se dispor de dados uniformemente distribuidos, ou em alguns casos, normais. Por isto, aqui é importante se especificar qual situação se deseja simular.

2. Geração do vetor Y

   Pela abordagem proposta, uma vez gerado o vetor X, especifica-se os coeficientes da reta que relaciona Y com X, e assim o segundo vetor pode ser obtido a partir do primeiro.

3. Geração dos erros em cada ponto

   Resultados de análise química possuem a peculiaridade de apresentar erros aleatórios dependendo do nível de teor. Este comportamento é chamado de heteroscedasticidade da variância. Isto acontece para ambos os métodos sendo considerados. Sendo assim, torna-se necessário incorporar aos valores de X e de Y gerados anteriormente, erros aleatórios que dependem dos valores X e Y em cada ponto.

   A geração destes erros se dará utilizando funções que conhecidamente relacionam os mesmos com o nível de teor. Elas podem ser de três tipos:

   (a) Relação Linear
   Em faixas de teor limitadas, como por exemplo quando o teor mais alto está em torno de dez vezes o teor mais baixo, comumente a aproximação linear é de boa qualidade.

   (b) Relação Potência
   Em faixas de teor amplas, a aproximação linear não se ajusta mais tão bem aos dados reais de erro de análise, sendo melhor utilizar uma função do tipo potência:

   $$\sigma = \alpha \cdot X^{\beta} \tag{1}$$

   (c) Relação de Rocke-Lorenzato
   Esta terceira opção assume importância quando a faixa de concentrações considerada é larga, entretanto, com resultados próximos ao limite de detecção do método analítico. Neste caso os autores do artigo na referência [5] mostram que pode-se melhor modelar o erro como função de dois componentes, um normal e um log-normal. A relação por eles proposta é da forma:

   $$\sigma = \sqrt{\alpha^2 + X^2 e^{\beta^2}(e^{\beta^2})} \tag{2}$$

   onde α é o erro normal e β é o erro log-normal.

4. Incorporação dos Erros aos Valores

   Para isto, usa-se novamente o gerador de números aleatórios para gerar um dado normal com média X e desvio padrão igual ao erro de X, calculado pelas relações do item anterior. Igualmente se procede para o valor correspondente de Y, usando o erro de Y, calculado da mesma forma. Estes novos valores devem substituir os valores originais para X e Y, pois agora eles possuem um erro aleatório já incorporado em ambas as variáveis.

5. Inserção de "Outliers"

   Dados reais, praticamente sempre incluem pontos "estranhos", que não parecem pertencer ao grupo de dados, porém muitas vezes não se sabe explicar sua ocorrência, e eles precisam ser tratados em conjunto com os demais. Dentro das várias abordagens existentes, poucas estão preparadas para este tipo de situação, apesar de ser uma realidade freqüente.

   A incorporação de "outliers" no programa será possível, apesar de não estar sendo apresentado no atual trabalho. O código aqui gerado poderá ser complementado posteriormente para incluir esta opção.

6. Saída

   Para a saida no presente trabalho será feita uma impressão simples dos dois vetores produzidos. Posteriormente, é desejável que o programa disponibilize-os na forma de um arquivo que possa ser lido por outro programa, ou utilizado em outro programa em C++, que irá então calcular as estimativas da reta que relaciona Y com X, para comparar com os coeficientes utilizados na simulaçáo.

5 Classe Proposta

```
// file Simulab.hpp

class SimuLab {
      int n;
      float *vetx, *errox;
      float *vety, *erroy;
public:
   SimuLab(int t);
   ~SimuLab(void);
   void GeraXUniforme(int *seed, float min, float max);
   void GeraY(float a, float b);
   void GeraErroXLinear(float alfa, float beta);
   void GeraErroXPotencia(float alfa, float beta);
   void GeraErroXRL(float alfa, float beta);
   void GeraErroYLinear(float alfa, float beta);
   void GeraErroYPotencia(float alfa, float beta);
   void GeraErroYRL(float alfa, float beta);
   void Sorteia(int *seed);
   void ImprimeXY(void);
};

// end of file Simulab.hpp
```

Figura 5.1: Classe Proposta

A Figura 5.1 apresenta a classe proposta para estruturação do programa em C++, de forma a atender aos requisitos descritos anteriormente.

6 Implementação Proposta

O anexo do presente relatório mostra a implementação desenvolvida para a classe apresentada. O seu teste é feito usando-se o programa ‘main’, também apresentado no anexo.

7 Conclusões

O presente texto mostra os detalhes necessários à implementação do programa projetado, a linha de raciocínio usada para projetar o algoritmo, a classe e implementação propostas para desenvolvimento na linguagem C++, assim como o teste do programa. Não foram incluidos no presente trabalho os métodos para geração de dados seguindo distribuições diferentes da uniforme, pois estes somente estarão disponíveis ao término da disciplina, quando a aluna responsável por este projeto (geração de números aleatórios) o tiver entregado. A inclusão do mesmo no atual trabalho será uma adaptação simples de se realizar.

A Implementação

```
// file simulab.cpp

#include <math.h>
#include <stdio.h>
#include "simulab.hpp"
#include "rand.c"

float GeraUniforme(int *seed, float min, float max){
     float y = (max-min)*urand(seed)+min;
     return(y);
}

float Eleva(float a, float b){
     return(exp(b*log(a)));
}

float GeraErroLinear(float x, float alfa, float beta){
     return(alfa+beta*x);
}

float GeraErroPotencia(float x, float alfa, float beta){
     return(alfa*(Eleva(x, beta)));
}

float GeraErroRL(float x, float alfa, float beta){
     return(sqrt(Eleva(alfa, 2)+ Eleva(x, 2)*
exp(Eleva(beta, 2))*exp(Eleva(beta, 2)-1)));
}

SimuLab::SimuLab(int tamanho){
     n = tamanho;
     vetx = new float[n]; errox = new float[n];
     vety = new float[n]; erroy = new float[n];
     for (int i=0; i<n; i++){
         vetx[i]=0.0; errox[i]=0.0;
         vety[i]=0.0; erroy[i]=0.0;
     }
}

SimuLab::~SimuLab(void){
     delete vetx;  delete errox;
     delete vety;  delete erroy;
}

void SimuLab::GeraXUniforme(int *seed, float min, float max){
        for (int i=0; i<n; i++){
            vetx[i]= GeraUniforme(seed, min, max);
        }
}

void SimuLab::GeraY(float a, float b){
     for (int i=0; i<n; i++){
 vety[i]= a + b*vetx[i];
     }
```

```
}

void SimuLab::GeraErroXLinear(float alfa, float beta){
for (int i=0; i<n; i++){
    errox[i]= GeraErroLinear(vetx[i], alfa, beta);
}
}

void SimuLab::GeraErroXPotencia(float alfa, float beta){
for (int i=0; i<n; i++){
    errox[i]= GeraErroPotencia(vetx[i], alfa, beta);
}
}

void SimuLab::GeraErroXRL(float alfa, float beta){
for (int i=0; i<n; i++){
    errox[i]= GeraErroRL(vetx[i], alfa, beta);
}
}

void SimuLab::GeraErroYLinear(float alfa, float beta){
for (int i=0; i<n; i++){
    erroy[i]= GeraErroLinear(vety[i], alfa, beta);
}
}

void SimuLab::GeraErroYPotencia(float alfa, float beta){
for (int i=0; i<n; i++){
    erroy[i]= GeraErroPotencia(vety[i], alfa, beta);
}
}

void SimuLab::GeraErroYRL(float alfa, float beta){
for (int i=0; i<n; i++){
    erroy[i]= GeraErroRL(vety[i], alfa, beta);
}
}

void SimuLab::Sorteia(int *seed){
     for (int i=0; i<n; i++){
vetx[i]=GeraUniforme(seed, (vetx[i]-3*errox[i],
(vetx[i]+3*errox[i]));
vety[i]=GeraUniforme(seed, (vety[i]-3*erroy[i]),
(vety[i]+3*erroy[i]));
     }
}

void SimuLab::ImprimeXY(void){
     fprintf(stdout, "      X         Y\n");
     for (int i=0; i<n; i++){
 fprintf(stdout, "%7.2f   %7.2f\n", vetx[i], vety[i]);
     }
}

// end of file simulab.cpp
```

B Programa Principal

```
// file smain.cpp

#include "simulab.hpp"

int main ( ) {
    SimuLab meuTeste(10);
    int semente= 333;
    meuTeste.GeraXUniforme(&semente, 0., 10.);
    meuTeste.GeraY(0., 1.);
    meuTeste.GeraErroXLinear(0., 0.5);
    meuTeste.GeraErroYRL(0.1, 0.3);
    meuTeste.Sorteia(&semente);
    meuTeste.ImprimeXY();
};

// end of file smain.cpp
```

Referências

[1] Barnett, Roy N. (1965) A scheme for the comparison of quantitative methods. *The American Journal of Clinical Pathology*, Vol 43, No.6, 562-569.

[2] Deming, W. E. (1943) *Statistical Adjustment of Data.* New York: John Wiley and Sons, Inc.

[3] Riu, J. and Rius, F. X. (1996) Assessing the accuracy of analytical methods using linear regression with errors in both axes. *Analytical Chemistry*, 68, 1851-1857.

[4] Cornbleet, P. J. and Gochman, N. (1979) Incorrect least-squares regression coefficients in Method-Comparison analysis. *Clinical Chemistry*, Vol. 25, No. 3.

[5] Rocke, D. M. e Lorenzato, S. (1995) A Two-component Model for Measurement Error in Analytical Chemistry. *Technometrics*, Vol. 37, No. 2, 176-184.

[6] Horwitz, W. (1982) Evaluation of analytical methods used for regulation of foods and drugs. *Analytical Chemistry*, Vol. 54, No. 1.

[7] Hartmann, C., Smeyers-Verbeke, J., Penninckx, W., Massart, D. L. (1997) Detection of bias in method comparison by regression analysis. *Analytica Chimica Acta*, 338, 19-40.

[8] Sarabia, L. A., Ortiz, M. C., Tomás, X. (1997) Performance of the orthogonal least median squares regression. *Analytica Chimica Acta*, 348, 11-18.

Um Exemplo de Aplicação de Estatística Bayesiana Empírica em Ranking de Indicadores de Desempenho

Mônica G. F. Costa[1]

[1]Departamento de Estatística - ICEx - UFMG,
Caixa Postal 702,
30123-970 - Belo Horizonte - MG
E-mail: mcosta@est.ufmg.br

Resumo

Neste trabalho serão apresentados resultados de um projeto que tem como ponto principal avaliar a incerteza na ordenação de *ranking*, bem como desenvolver um programa em C++ para gerar uma medida de avaliação. Como ilustração, usaremos a probabilidade de morte dos filhos nascidos vivos dos 143 municípios que compunham, em 1991, a mesoregião Sul/Sudoeste de Minas Gerais.

1 Introdução

Em vários estudos de indicadores de desempenho, os resultados são apresentados através de *ranking*. É de suma importância fazer um estudo mais completo para avaliar a incerteza na ordenação do *ranking*, pois o *ranking* é baseado na razão de risco, que por sua vez está sujeita a flutuações estatísticas. Não se pode esquecer que a principal utilidade deste tipo de estudo reside no fato deles servirem de apoio a pesquisadores que irão usar os indicadores construídos e necessitarão conhecer a precisão destas estimativas.

A estatística Bayesiana empírica, um ramo muito importante da estatística [7], pode ser bem aplicada em estudos de avaliação de *ranking*. Existem outros métodos de validação da incerteza da ordenação de *ranking* [6]. Um método simples é calcular o intervalo de confiança da razão de risco e checar se estão ou não sobrepostos. Neste trabalho usaremos o método Bayesiano empírico para calcular a medida C e avaliar a incerteza da ordenação.

Na próxima seção serão apresentados o objetivo do projeto, uma breve descrição do modelo e da metodologia utilizada para calcular a medida C e o programa em C++ para gerar esta medida C. E, finalmente na última seção teremos as conclusões.

2 Objetivo

O ponto principal deste projeto é elaborar um programa em C++ que dado uma serie de indicadores de desempenho de entrada, possa gerar como saída uma medida de validação de incerteza na ordenação do *ranking* destes indicadores, como exemplo usaremos a probabilidade de morte dos filhos nascidos vivos dos 143 municípios que compunham, em 1991, a mesoregião Sul / Sudoeste de Minas Gerais.

3 Medida de Validação da Ordenação do *Ranking*

Nos casos em que o número de itens ordenados é grande e está além da capacidade de memória humana, propõe-se calcular um simples valor de medida C, que irá avaliar a incerteza na ordenação do *ranking*, e poderá ser apresentado com a tabela. Exemplos de aplicação podem ser visto em Anderson, *et al.* [1] e Aitkin, *et al.* [2]. A medida C encontrada será a mudança aleatória esperada na ordenação do ranking.

A seguir apresentamos o método para geração desta medida.

Dado p_j, uma série com a probabilidade de morte, iremos calcular r_j, o posto do j-ésimo item:

$$r_j = \text{posto}(p_j)$$

A medida é calculada usando o método *boostrap* [4], que pode ser resumido como se segue.

- Calcule para cada unidade uma nova proporção p_j^* que tem distribuição Beta (ver Equação (1) no próximo tópico).
- Obtenha o r_{jl}^*, o valor do novo posto por unidade j, com l replicações,
$$r_{jl}^* = \text{posto}(p_{jl}^*)$$
- Calcule d_{jl}, o desvio do posto original para o novo posto, que é dado por
$$d_{jl} = |r_j - r_{jl}^*|$$
- E depois repita o procedimento L vezes e calcule
$$\overline{d_j} = \sum_{l=1}^{L} \frac{d_{jl}}{L}.$$

Com a medida $\overline{d_j}$, tem-se a mudança esperada no ranking para cada unidade j. Usaremos a regra prática de $L = 20$.

A medida geral pode ser calculada como

$$C = \sum_{j=1}^{K} \frac{\overline{d_j}}{K}.$$

e é naturalmente interpretada como a mudança esperada na ordenação das unidades.

Quando $C = 1$, significa que, em média, a mudança de ordenação é de uma posição, devido a variação aleatória.

4 O Modelo Beta/Binomial na Análise Bayesiana Empírica

O modelo Beta/Binomial é muito apropriado para analisar dados de probabilidade de morte, porque a medida é obtida baseada em dois potenciais resultados, sobrevivência e morte. Em [3], há uma clara exposição deste modelo.

Tem-se que $\widehat{p_j}$ é a probabilidade estimada de morte, n_j, o número de observações na unidade j, $j = 1, \ldots, k$, e k, o número de unidades, no nosso exemplo, os 143 municípios.

A distribuição relevante para desenhar p_j^* é:

$$\text{Beta}\left[n_j\widehat{p}_j + \widehat{M}\overline{p};\ \ n_j(1-\widehat{p}_j) + \widehat{M}(1-\overline{p})\right] \tag{1}$$

onde $\overline{p}$ é a probabilidade estimada sobre todos as unidades.

$\widehat{M}$ é também estimado dos dados e determina a redução dos $\widehat{p}_j$, onde

$$\widehat{M} = \frac{\overline{p}}{\phi},$$

$$\phi = \frac{\sum_{j=1}^{k} n_j(p_j - \overline{p})^2}{\sum_{j=1}^{k} n_j} - \frac{\overline{p}}{\overline{n}}$$

e

$$\overline{n} = \frac{\sum_{j=1}^{k} n_j}{k}.$$

A estimativa de ShrunKen ou estimativa encolhida dos p_j^* é:

$$\frac{\widehat{M}}{\widehat{M}+n_j}\overline{p} + \frac{n_j}{\widehat{M}+n_j}\widehat{p}_j.$$

Um $\widehat{M}$ alto sugere que a variação do desempenho das unidades é devido a variação aleatória.

5 Programa em C++

Para obter a medida C foi necessário criar um programa, que foi feito por programação orientada por objetos, em C++ [5,8].

No arquivo `medidac.hpp`, foram incluídos o arquivo `listaR.hpp`, pois trabalhamos com tipo abstrato de dados `lista` de reais, e `stdio.h`, porque no programa iremos ler arquivos externos usando a variável `FILE` e imprimir os resultados usando arquivo o `stdout`. Foi criada uma classe `MedidaC`, que tem um construtor *default* e três funções, `LeiaDados`, `CalculeRank` e `Calcule`. No arquivo `medidac.cpp` foi criada a implementação destas funções. Aqui também foram incluídos os arquivos `math.h`, para as funções matemáticas, `stdio.h`, para possibilitar o uso do arquivo `stdout`, `listaR.cpp`, para a implementação do arquivo `listaR.hpp` e o arquivo `medidac.hpp`, onde foi definida a classe `MedidaC`. No arquivo `mainc.cpp` também incluimos os arquivos `medidac.cpp` e `stdio.h`. No `mainc.cpp` devem-se fornecer os nomes dos arquivos externos com os dados, para calcular a medida C e imprimir o resultado final.

Foram fornecidos dois arquivos de dados, um com os valores das probabilidades de morte e o outro com os valores das reamostras Beta(α, β).

O programa em C++ é apresentado a seguir:

5.1 Arquivo medidac.hpp

```
// begin file medidac.hpp

#ifndef _MEDIDA_C_
#define _MEDIDA_C_
```

```
#include <stdio.h>
#include "listaR.hpp"

const int NU_REPL = 20;

class MedidaC {
 private:
     ListaR pj;
     ListaR pjRank;
     ListaR *pjEstrela;
 public:
     MedidaC(void);
     ~MedidaC(void);

     void LeiaDados(FILE *entrada);
     void LeiaDados(FILE *entrada, FILE *entradab);
     ListaR CalculeRank(ListaR *pj);
     float Calcule(void);
};

#endif
```

5.2 Arquivo medidac.cpp

```
// begin file medidac.cpp

#include <math.h>
#include <stdio.h>
#include "listaR.cpp"
#include "medidac.hpp"

MedidaC::MedidaC() {
  pjEstrela = new ListaR[NU_REPL];
}

MedidaC::~MedidaC() {
  delete pjEstrela;
}

void MedidaC::LeiaDados(FILE *entrada)  {
  float dados;
  while ( fscanf(entrada, "%f", &dados) != EOF ) {
        pj.Insere(dados, pj.Final());
  }
  fprintf(stdout, "pj:\n");
  pj.Imprime();
}

void MedidaC::LeiaDados(FILE *entrada, FILE *entradab)  {
  int index = 0;
  float dadosBeta;
```

```
  MedidaC::LeiaDados(entrada);
  while ( fscanf(entradab, "%f", &dadosBeta) != EOF ) {
     pjEstrela[index].Insere(dadosBeta, pjEstrela[index].Final());
     index++;
     if ( index == NU_REPL )
        index = 0;
  }
  for (index = 0; index < NU_REPL; index++) {
    fprintf(stdout, "pjEstrela[%d]:\n", index+1);
    pjEstrela[index].Imprime();
  }
}

ListaR MedidaC::CalculeRank(ListaR *pj)  {
  ListaR  pjaux = *pj;
  ListaR rank;
  int i, j;
  int indiceMaior;
  float maior;

  for ( i = 0; i < pj->Tamanho(); i++)
      rank.Insere(-1, rank.Final());
  for ( i = 0; i < pj->Tamanho(); i++ ) {
    // ache maior
    maior = -1.0;
    for ( j = 0; j < pj->Tamanho(); j++ ) {
        if (pjaux.Recupera(j) > maior)  {
           maior  =  pjaux.Recupera(j);
           indiceMaior = j;
        }
    }
    pjaux.Retira(indiceMaior);
    rank.Retira(indiceMaior);
    pjaux.Insere(-1E+38, indiceMaior);
    rank.Insere(i, indiceMaior);
  }
  fprintf(stdout, "rank:\n");
  rank.Imprime();
  return rank;
}

float MedidaC::Calcule(void)  {
  ListaR djBarra;
  ListaR pjRank = CalculeRank(&pj);
  ListaR pjEstrelaRank;
  int i, j;
  float dif;

  // calcule djBarra
  for ( i = 0; i < pj.Tamanho(); i++)
    djBarra.Insere(0,i);
```

```
  for ( i = 0; i < NU_REPL; i++)  {
    //  gere rank
    fprintf(stdout, "pjEstrelaRank[%d]:\n", i+1);
    pjEstrelaRank = CalculeRank(&pjEstrela[i]);
    // calcule diferenca
    for ( j = 0; j < pj.Tamanho(); j++)  {
       dif = fabs(pjRank.Recupera(j) - pjEstrelaRank.Recupera(j));
       dif += djBarra.Recupera(j);
       djBarra.Retira(j);
       djBarra.Insere(dif, j);
    }
  }
  fprintf(stdout, "djBarra:\n");
  djBarra.Imprime();

  // calcule estatistica C
  float C = 0.0;
  for ( i = 0; i < pj.Tamanho(); i++)  {
    C += djBarra.Recupera(i);
  }
  C /= (pj.Tamanho()*NU_REPL);
  return C;
}
```

5.3 Arquivo mainc.cpp

```
//  begin file  mainC.cpp

#include <stdio.h>
#include "medidac.cpp"

int main ( ) {

   FILE *entra, *entrab;
   MedidaC minhaMedida;
   float C;

   entra = fopen("entrada.txt", "r");
   entrab = fopen("entrab.txt", "r");

   minhaMedida.LeiaDados(entra, entrab);
   C = minhaMedida.Calcule();
   fprintf(stdout, "A estatistica C eh %f \n", C);
   fclose(entra);
   fclose(entrab);
}
```

6 Conclusão

Foi apresentado um projeto que pretende avaliar a incerteza na ordenação do ranking. Esta avaliação foi feita através de um exemplo prático onde foi utilizado o programa em C++ e as probabilidades de morte dos filhos nascidos vivos dos 143 municípios da mesoregião Sul/Sudoeste de Minas Gerais, em 1991.

O resultado encontrado foi $C = 8,553147$, ou seja, em média, a mudança de ordenação do ranking é de aproximadamente 9 posições, devido a variação aleatória. Este valor poderá ser apresentado com a tabela de ranking. Para futuras estudos com estes dados os pesquisadores deverão levar em consideração a mudança de ordenação da posição.

Através deste exemplo foi possível adquirir um conhecimento maior da programação em C++.

Vale a pena lembrar que, dada uma biblioteca que implementa um gerador de números aleatórios com distribuição Beta(α, β), podemos então incluir algumas funções na classe `MedidaC` para calcular α, β e as reamostras. Para isso devemos incluir as seguintes funções na classe `MedidaC`:

```
class MedidaC {
  ...
public:
  ...
  float CalculeAlpha(void);
  float CalculeBeta(void);
  ListaR GereReamostra(int *semente);
  ...
```

E, no arquivo `medidac.cpp`, implementá-las:

```
...
float MedidaC::CalculeAlpha(void) {
  // nj e pj serao fornecidos, ou seja, numero de mortes, nascidos vivos
  pbarra = sum(mortes) / sum(individuos); // taxa global
  nbarra = sum(individuos) / numero de unidades(municipios); // n global
  qfi = sum((nj* (pj-pbarra)^2)) / sum(nj) - (pbarra / nbarra);
  Mhat = pbarra / qfi;
  alpha = (nj * pj + Mhat * pbarra);
}

float MedidaC::CalculeBeta(void) {
  beta = (nj * (1-pj)) + (Mhat * (1-pbarra));
}

ListaR MedidaC::GereReamostra(int *semente) {
  float alpha, beta;
  alpha = CalculeAlpha();
  beta = CalculeBeta();
  Aleatorio meuAleatorio;
  ListaR minhaLista = meuAleatorio.GereBeta(semente, pj.Tamanho(), alpha, beta);
  return minhaLista;
}
```

```
...

float MedidaC::Calcule() {
...
  // gerar reamostra
  pjestrela = GereReamostra(&semente);
...
```

Referências

[1] Aitkin, M. and Longford, N. (1986), "Statistical Modeling Issues in School Effectiveness Studies" (with discussion), *Journal of the Royal Statistical Society, Serie A*, 149, 1-43.

[2] Anderson, J., and Mattson, S., (1998), "Random Ranking of Hospitals Is Unsound", *CHANCE*, vol II, NO.3, 34-39.

[3] Carlin, B. P. and Louis, T. A. (1996), *Bayes and Empirical Bayes Methods for Data Analysis*, New York: Chapman and Hall.

[4] Efron, B. and Tibshirani, R. J. (1993), *An Introduction to the Bootstrap,* New York: Chapman and Hall.

[5] G. Buzzi-Ferraris.(1994), *Scientific C ++: Building Numerical Libraries the Objected-Oriented Way*, Addison- Wesley Publishing Company, Cambridge.

[6] Goldstein, H. and Spiegelhalter, D. J. (1996), "League Tables and Their Limitations: Statistical Issues in Comparisons of Institutional Perfomance" (with discussion), *Journal of the Royal Statistical Society, Serie A,* 159, 385-443.

[7] Lee, Peter M. (1997), *Bayesian Statistics: An Introduction* , London, Wiley.

[8] N. Ziviani. (1993), *Projeto de Algoritmos - Com Implementação em Pascal e C*, Livraria Pioneira Editora, São Paulo.

Gerador Bootstrap

Wellington J. Cunha[1]

[1]Departamento de Estatística - ICEx - UFMG,
Caixa Postal 702,
30123-970 - Belo Horizonte - MG
E-mail: welljc@est.ufmg.br

Resumo

Apresentamos detalhes sobre um estudo que teve como objetivo a disponibilização de um aplicativo estatístico em C++. Descrevemos também o método estatístico de reamostragem que foi utilizado para gerar amostras *bootstrap* e estimar o erro padrão da média amostral.

1 Introdução

Em muitos trabalhos estatísticos se faz necessário o cálculo de estimativas da média, mediana, desvio padrão e intervalos de confiança para estas estatísticas. Neste trabalho, propomos desenvolver um programa que gere amostras *bootstrap* [2, 3] para estimar o erro padrão da média através destas replicações, o que nos deixa a um passo de calcularmos o intervalo de confiança para este estimador da média. Isto se faz necessário, uma vez que em muitos casos práticos estes estimadores são difíceis de serem calculados pelos métodos tradicionais.

Usaremos neste nosso trabalho a estimativa do erro padrão da média. Ela foi considerada mais adequada para verificação e depuração do aplicativo desenvolvido, uma vez que é fácilmente obtida pelos métodos tradicionais. Isto não compromete o aplicativo uma vez que facilmente poderá ser estendido para outras estatísticas mais complicadas.

Temos portanto, o objetivo de apresentar um programa em C++, padrão ANSI [1], capaz de capturar um banco de dados e fazer n replicações *bootstrap* desta amostra usando-a para estimar o erro padrão da média amostral.

A fim de que este trabalho seja uma ferramenta para outros trabalhos, organizamos este texto de modo a iniciar com uma breve descrição do método estatístico usado. Logo após, descrevemos o algoritmo, seguido das classes, funções e o programa principal para exemplificar o uso da biblioteca. Finalizamos com uma conclusão.

2 Métodos Estatísticos

Para o desenvolvimento de nosso programa, nos basearemos na teoria estatística existente sobre a técnica de reamostragem chamada *bootstrap* [2, 3]. Esta técnica consiste em retirar, aleatoriamente, de um banco de dados n amostras de mesmo tamanho, que são as amostras *bootstrap*. Estas amostras podem ser tratadas posteriormente para o cálculo de medidas estatísticas. No nosso caso, a estatística considerada será a média amostral. Verificaremos a precisão da média amostral através de uma estimativa do erro padrão das replicações *bootstrap*.

Para descrever o método *bootstrap* para estimar o erro padrão, desenvolvido por Efron [3], suponhamos que observamos um conjunto independente de dados $x_1, x_2, \ldots, x_n$, que, por conveniencia, denotaremos pelo vetor

$$X = (x_1, x_2, \ldots, x_n),$$

e a partir destes dados calcularemos nossa estatística de interesse $S(X)$. No nosso caso, é a média amostral.

Uma amostra *bootstrap*

$$X^* = (x_1^*, x_2^*, \dots, x_n^*)$$

é obtida retirando aleatoriamente n amostras, com reposição, a partir da amostra original $x_1, x_2, \dots, x_n$. Por exemplo, com $n = 5$, poderiamos obter

$$X^* = (x_5, x_5, x_4, x_3, x_1).$$

Como vemos na Figura 3.1, para cada *amostra bootstrap* obtida, calculamos a *replicação bootstrap* da estatística S. Isto é, $S(X^{*b})$ é o valor da estatística S avaliada sobre X^{*b}. O estimador *bootstrap* do erro padrão é o desvio padrão das replicações bootstrap,

$$\hat{se}_{boot} = \left\{ \sum_{b=1}^{B} \frac{[S(X^{*b}) - S(.)]^2}{(B-1)} \right\}^{\frac{1}{2}}$$

onde $S(.) = \sum_{b=1}^{B} \frac{S(X^{*b})}{B}$ e B é o número de amostras *bootstrap* utilizadas.

3 Algoritmo

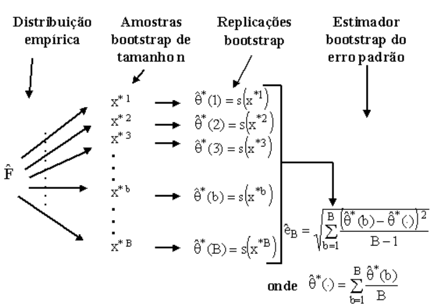

Figura 3.1: Esquema do Processo Bootstrap

O algoritmo para este trabalho, ilustrado na Figura 3.1, é:

algoritmo
    leia amostra original
    retire B amostras bootstrap *da amostra original*
    defina estatística
    calcule replicação da estatística de cada amostra bootstrap
    calcule estimador bootstrap *do erro padrão*
    imprima estimador bootstrap
fim algoritmo

4 Classes

Utilizamos a classe Vector descrita em [1], e o gerador de números aleatórios *rand.c*.

4.1 boot.hpp

```
// file boot.hpp
// Classe Bootstrap que utiliza vector
// do Buzzi

#ifndef boot_hpp
#define boot_hpp

class Boot : Vector {  //friend Vector;
  Vector dados;
public:
  Boot(const Vector &vetDados);
  ~Boot();
  Vector AmostraBoot(int *x);
  Vector ReplicaBoot(int num,float (*est)(const Vector &vet));
  float ErroPadBoot(const Vector &vetDados);
};

#endif
```

4.2 boot.cpp

```
//file boot.cpp
#ifndef boot_cpp
#define boot_cpp

#include <math.h>
#include "rand.c"
#include "c:\prgcpp\buzzi\vector.cpp"
#include "boot.hpp"
#include <stdio.h>

Boot::Boot(const Vector &vetDados){
  dados=vetDados;
};

Boot::~Boot(){};

Vector Boot::AmostraBoot(int *x){
  int tamanho=dados.Size();
  Vector aux(tamanho);
  for (int i=1; i<=tamanho;i++){
    int p=(tamanho*urand(x))+1;
    (*x)++;
    aux[i]=dados.GetValue(p);
  };
  return(aux);
};
```

```
Vector Boot::ReplicaBoot(int num, float (*est)(const Vector &vet)){
  Vector vetor(num);
  int *rdx;
  for(int i=1; i<=num;i++){
    *rdx=i*num;
    vetor[i]=est(AmostraBoot(rdx));
  };
  return(vetor);
};

float Boot::ErroPadBoot(const Vector &vetDados){
  int dim=vetDados.Size();
  float media=0;
  for (int i=1;i<=dim;i++){
    media=media+vetDados.GetValue(i);
  };
  media=media/dim;
  float erro=0;
  for(int t=1; t<=dim;t++){
    erro=erro+((vetDados.GetValue(t))-media)*
          ((vetDados.GetValue(t))-media);
  };
  erro=sqrt(erro/(dim-1));
  return(erro);
};
//end of file boot.cpp
#endif boot_cpp
```

4.3 mainboot.cpp

```
#include "utility.cpp"
#include "matrix.cpp"
#include <stdio.h>
#include "boot.cpp"
\\ deve se incluir todas estas bibliotecas

\\funcao da estatistica de interesse
  float fmedia(const Vector &vdados){
  int dim=vdados.Size();
  float med=0.;
  for (int i=1; i<=dim;i++){
    med=med+vdados.GetValue(i);
  };
  med=med/dim;
  return(med);
};

\\ a formatacao de dados.txt e de uma linha
\\ para tamanho do vetor e outra linha para
\\ as entradas separadas por espaco

int main(){
```

```
  Vector entdados("dados.txt"); //dados formatado
  Boot teste(entdados);
  int num;
  fprintf(stdout,"numero de replicacoes? ");
  fscanf(stdin,"%i",&num);
  Vector repdados(num);
  repdados=teste.ReplicaBoot(num,*fmedia);
  fprintf(stdout,"O erro padrao da amostra e:
  \%f\n",teste.ErroPadBoot(repdados));
  return(0);
};
```

5 Aplicação

Usaremos para verificação do nosso programa o problema descrito em [3]. O problema consiste em responder se um determinado tratamento prolonga o tempo de sobrevivência. Para tanto foi estudado um banco de dados sobre o tempo de sobrevivência, em dias, de 16 ratos que foram distribuidos aleatoriamente em dois grupos, *tratamento* e *controle*. A Tabela 5.1 apresenta os dados do estudo. Dezesseis ratos foram distribuidos aleatoreamente nos grupos de controle e de tratamento. São mostrados seus tempos de sobrevivência, em dias, seguidos de algumas estimativas.

Tabela 5.1 : Dados de Estudo

Grupo	Dados	Tam.	Média	Err.Pad.
Tratamento:	94 197 16			
	38 99 141			
	23	(7)	86,86	25,24
Controle:	52 104 146			
	10 50 31			
	40 27 46	(9)	56,22	14,14
Diferença:			30,63	28,93

Uma comparação das médias para os 2 grupos oferece a princípio motivos para otimismo. Sejam $x_1, x_2, x_3, \ldots, x_7$ os dados de tempo de sobrevivência no grupo de tratamento e $y_1, y_2, y_3, \ldots, y_9$, o tempo de sobrevivência do grupo de controle. As médias dos grupos são:

$$\overline{x} = \sum_{i=1}^{7} \frac{x_i}{7} \text{ e } \overline{y} = \sum_{i=1}^{9} \frac{y_i}{9}.$$

Assim, a diferença $\overline{x} - \overline{y}$ é igual a 30,63, sugerindo um consideravel prolongamento do tempo de sobrevivência para o tratamento. Mas quão exato é este estimador?

Para responder a esta pergunta precisamos estimar a precisão das médias amostrais $\overline{x}$ e $\overline{y}$. Especialmente para a média amostral é fácil obter uma fórmula para precisão. O que não acontece se por exemplo quiséssemos determinar a precisão da mediana. Isso tornaria nosso programa extremamente útil.

A estimativa para o erro padrão da média $\overline{x}$, baseada em n valores independentes, $x_1, x_2, x_3, \ldots, x_n$, é dada pela fórmula $\sqrt{\frac{S^2}{n}}$ onde $S^2 = \sum_{i=1}^{n} \frac{(x_i - \overline{x})^2}{n-1}$.

Se o erro padrão estimado no experimento dos ratos for muito pequeno, então saberemos que $\overline{x}$ e $\overline{y}$ estão próximos de seus valores esperados. Isto significa que a diferença obtida de $30,63$ foi uma boa estimativa para o verdadeiro prolongamento do tempo de sobrevivência, capacitanto, portanto, o tratamento. Por outro lado, se o erro padrão estimado for muito grande, então a diferença estimada será inapropriada para afirmar sobre o tratamento. Falando de maneira aproximada, um estimador será menor do que o erro padrão por volta de 68% das vezes e menor do que 2 erros padrão por volta de 95% das vezes.

Observemos que para variáveis aleatórias independentes o cálculo do erro padrão da diferença das médias é dado por

$$se(\overline{x}-\overline{y}) = \sqrt{\overline{x}^2+\overline{y}^2}$$

já que a variâncias da diferença de duas quantidades independente é a soma de suas variâncias. Assim, encontramos o valor e 28,93 na Tabela 5.1.

Pelos cálculos da Tabela 5.1, vemos que o erro é muito grande, pois $30,63 \div 28,93 = 1,05$, indicando uma grande variabilidade.

Usando o programa, que estima o erro da média através do *bootstrap*, encontramos os resultados da Tabela 5.2 que fornecem o mesmo indicativo de grande variabilidade. Efron [3] afirma que uma variação de 50 até 200 replicações fornece uma boa estimativa para o erro padrão.

Tabela 5.2: Estimativas *Bootstrap* do Erro Padrão da Média Amostral

Grupo	replicações				
	50	100	250	500	1000
Tratamento	23,04	22,08	23,30	23,11	23,07
Controle	13,96	13,24	13,25	13,29	13,35
Diferença	26,94	25,75	26,8	26,66	26,65

6 Conclusões

Com este trabalho disponibilizamos uma ferramenta que poderá auxiliar pesquisadores a formular respostas satisfatórias, por caminhos mais fáceis e rápidos. Estamos procurando não só resolver o problema de estimar o erro padrão da média amostral, mas também de qualquer estatística que poderá ser implementada através de uma função que a determine. Por exemplo no problema descrito na seção 5, se desejássemos calcular a mediana em vez da média amostral, não teríamos uma fórmula, o que tornaria nosso programa bastante útil.

Referências

[1] Buzzi-Ferraris, Guido. *Scientific C++: Building Numerical libraries the Object-Oriented Way.* Cambridge: Addison-Wesley Publishing Company, 1994.

[2] Efron, B., The Jacknife, *Bootstrap and Other Resampling Plans.* Philadelphia: Society for Industrial and Applied Mathematics, 1982.

[3] Efron, B., and Tibshirani, Robert J. An *Introduction to the Bootstrap.* New York: Chapman & Hall, 1993.

Estimador da Função de Sobrevivência

FLÁVIO F. FERREIRA[1] E CLEIDE B. SOUSA[1]

[1]Departamento de Estatística - ICEx - UFMG,
Caixa Postal 702,
30123-970 - Belo Horizonte - MG
E-mail: {flavio,cleide}@est.ufmg.br

Resumo

A análise do tempo de falha se refere ao conjunto de técnicas estatísticas para a análise de dados de durabilidade provenientes de uma população homogênea. Esta análise responde a várias perguntas relacionadas ao tempo de vida de produtos ou componentes. Este trabalho trata de um assunto da área de confiabilidade e análise de sobrevivência, onde estudamos o estimador de Kaplan-Meier da função de sobrevivência, para amostras com poucas observações.

1 Introdução

A prática nos mostra que produtos saindo da mesma linha de produção e operando sob condições similares, apresentam tempos de falha (tempo em que um produto ou componente deixa de funcionar) distintos. Assim sendo, um estudo de tempo de falha só deve ser descrito em termos probabilísticos e qualquer análise estatística de dados de confiabilidade deve ser baseada em termos bem definidos. Tais conceitos envolvem o tipo de teste a ser realizado e a sua elaboração em termos probabilísticos.

Inicialmente, temos que definir de forma clara e precisa, o conceito de falha. Iremos considerar uma falha quando um equipamento ou componente deixa de funcionar quando este se encontra sob teste. Temos que definir também o significado do evento censura, que será muito importante no decorrer do trabalho. Quando trabalhamos com dados de confiabilidade, temos que lidar com dois tipos de censura (do tipo I e II).

Censura do tipo I é o evento que leva o componente a falhar por influências externas, não previstas no teste (por exemplo, deficiência de infraestrura do laboratório onde se realiza o teste). A censura do tipo II ocorre quando um componente ou equipamento permanece funcionando além do período pré-fixado pelo teste, ou seja, fixa-se um tempo para a realização do teste e o equipamento permanece funcionando além desse tempo.

Esta é uma das principais funções probabilísticas usadas para descrever estudos provenientes de testes de durabilidade. A função de confiabilidade é definida como a probabilidade de um produto desenvolver sua função sem falhar até um certo tempo t. Em termos probabilísticos, isto é escrito em função da variavel de interesse T, tempo até a ocorrência da falha, como $R(t) = P(T > t)$.

É importante ressaltar que, o estimador de Kaplan Meier é específico para o caso em que há censura nas observações. O limite-produto, ou Kaplan-Meier [1], como é usualmente chamado, é um estimador não paramétrico para a função de sobrevivência. A função de sobrevivência determina a probabilidade de um produto (ou indivíduo) desenvolver sua função sem falhar até um certo tempo t. Com isso, faremos um estudo voltado a pequenos conjuntos de dados e verificaremos o comportamento desse estimador.

Apresentamos a seguir o plano de ação e concluímos com o encerramento do projeto.

2 Metodologia

Como o interesse desse estudo é calcular o estimador de Kaplan-Meier, utilizamos a seguinte expressão para efetuar esse cálculo:

$$\hat{R}(t) = \prod_{i=1}^{J} \frac{(n_i - d_i)}{n_i} \qquad (2.1)$$

onde n_i é o número de itens sob risco (não falhou e não foi censurado) no tempo t_i, d_i é o número de falhas no tempo t_i e t_i, o tempo de falha.

Além disso, calculamos uma estimativa para a variância do estimador de Kaplan-Meier, que é dada por:

$$\widehat{Var}(\hat{R}(t)) = \hat{R}(t)^2 * \sum_{i=1}^{J} \frac{d_i}{n_i * (n_i - d_i)} \qquad (2.2)$$

A partir do cálculo deste valor é possível construir intervalos de confiança para $R(t)$, em um certo tempo t. Utilizamos um intervalo aproximado de 95% de confiança para $R(t)$ da forma:

$$\hat{R}(t) \pm 1,96 * \sqrt{\widehat{Var}(\hat{R}(t))} \qquad (2.3)$$

Entretanto, em valores extremos de $R(t)$, próximo de um ou zero, este intervalo de confiança pode apresentar limite inferior negativo ou limite superior maior que um. Este problema pode ser resolvido utilizando uma transformação para $R(t)$. Por exemplo, a variância para amostras de

$$\hat{U}(t) = log[-log\hat{R}(t)] \qquad (2.4)$$

é estimada como

$$\widehat{Var}(\hat{U}(t)) = \frac{\hat{A}(t)}{\hat{B}(t)} \qquad (2.5)$$

onde:

$$\hat{A}(t) = \sum_{i=1}^{J} \frac{d_i}{n_i * (n_i - d_i)}$$

e

$$\hat{B}(t) = \sum_{i=1}^{J} \log \frac{(n_i - d_i)}{n_i^2}^2$$

Um intervalo aproximado de 95% de confiança para $U(t)$ é então facilmente obtido e o correspondente intervalo para $R(t)$ é dado por

$$\hat{R}(t)^{exp\left(\pm 1,96 * \sqrt{\widehat{Var}(\hat{U}(t))}\right)} \qquad (2.6)$$

que assume valores no intervalo [0,1].

3 Comentários sobre o programa

Nesta seção apresentamos alguns comentários sobre a implementação do programa na linguagem C++.

Pf2.hpp: Neste arquivo estão definidas as variáveis e a estrutura do programa principal.

Main.cpp: Neste arquivo encontramos os dados iniciais para a execução do programa. Ele contém as observações (tempo de falhas) para o cálculo do estimador de Kaplan-Meier (ver 2.1). A variável tam é o tamanho do vetor de observações. O objeto atribui_p(p) foi utilizado para atribuir um valor inteiro para p, onde p é o nível que se deseja para um intervalo de 95% de confiança para $\hat{R}(p)$. Se $p = 0$, é desejável um intervalo de confiança no tempo $t = 0$, onde os limites superior e inferior desse intervalo terão valor 1, pois em $t = 0$ não há falhas e, portanto, a variância é zero e $\hat{R}(t) = 1$. Se $p =$ tamanho do vetor $\hat{R}(t)$, não terá sentido o cálculo do intervalo de confiança, pois esta variância é infinitamente grande. Para um exemplo de funcionamento utilizamos $p = 3$, ou seja, estamos querendo um intervalo de confiança para $t = 3$. O objeto InsiraElem(elem, censura, pos) insere as observações (elem) na posição = pos, no vetor de observações (obs) e insere o indicador de censura (censura) na posição = pos do vetor Delta. O indicador de censura é 0 ou 1, onde 0 significa que houve falha e 1 censura (critério padrão no estudo de confiabilidade).

Pf2.cpp: Neste arquivo encontram-se os procedimentos para a execução do programa, ou seja, as classes. Definimos a classe KM, onde temos:

KM::KM() constroi os vetores Obs (observações), Delta (vetor que indica se a i-ésima observação foi ou não censurada (1 e 0, respectivamente)), Falha_v (vetor com o número de falhas no tempo t), IndSobRisco_v (vetor de indivíduos sob risco de falhar (ou em teste)) e Risco_v (vetor da estimativa de $\hat{R}(t)$).

KM:: KM é o destrutor.

KM::Ordene é um algoritmo criado para ordenar crescentemente as observações em paralelo com o respectivo vetor Delta. Esta seria uma justificativa para não utilizar a implementação sugerida (utility.cpp). Este algoritmo ordena as observações baseando-se no maior e no menor valor. Funciona da seguinte maneira: encontro o maior valor das observações. Em seguida, utilizando um pivô (primeira posição no vetor), atribui-se o maior valor na variável menor e percorre-se todo o vetor de observações para encontrar o menor valor. Encontrando esse valor, coloca-se este na primeira posição do vetor de observações e a primeira observação na posição do menor valor. Em seguida, o pivô para a ser a segunda observação, repetindo estes passos até que o pivô seja a posição da última observação e, nesta posição, encontra-se o maior valor encontrado primeiramente. O procedimento de troca de observações é feito juntamente com o vetor Delta, ou seja, se trocar o i-ésimo elemento do vetor de observações de posição, o mesmo deve ser feito com o i-ésimo elemento do vetor Delta. Quando houver observações empatadas (falha e censura simultâneamente) o critério de desempate é o Delta, isto é, a censura deve vir após a falha.

KM::IndSobRisco() cria o vetor de indivíduos sob risco de falha, que na verdade será um vetor ordenado iniciando com o valor igual ao tamanho do vetor e decrescendo até o valor 1.

KM::Falha() constroi o vetor de falhas, que indica o número de falhas no tempo t. Quando há empates, o número de falhas é igual ao número de empates, desde que o Delta correspondente ao empate seja zero. Se o i-ésimo valor de Delta for 1, o i-ésimo elemento do vetor falha será igual a zero. Se houver mais empates nas observações e os correspondentes elementos do vetor Delta forem zero, a i-ésima posição do vetor falha será igual ao número de empates e as i-ésimas posições seguintes do vetor de falha empatadas serão zero. Quando não houver empates e o Delta for zero (indicador de falha) o i-ésimo valor de falha será igual a 1.

KM::Risco() $\hat{R}(t)$ é o estimador da função de sobrevivência, o qual chamamos aqui de Risco (ver 2.1). Note que, o tamanho do vetor de risco é sempre menor ou igual que o tamanho do vetor de observação. Somente será igual quando existir apenas uma censura, pois $\hat{R}(0) = 1$. O i-ésimo risco só é calculado quando o i-ésimo elemento de falha é diferente de zero.

KM::STD() calcula a variância de $\hat{R}(p)$ (ver 2.2). Depende do valor da variável p atribuida em atribui_p(p), em que $0<p<tam$.

KM::IC95 calcula o intervalo de 95% de confiança para $\hat{R}(t)$ (ver 2.3). Como o intervalo de confiança deve estar entre 0 e 1, há uma correção quando esta condição não é satisfeita (ver 2.4, 2.5 e 2.6).

KM::Imprima() imprime os resultados do cálculo do estimador de Kaplan-Meier ($\hat{R}(t)$ =Risco_v), da variância de $\hat{R}(p)$ e os respectivos limites superior e inferior de 95% de confiança para $\hat{R}(p)$.

4 Implementação

Nesta seção apresentamos a implementação computacional do programa em linguagem C++, que segue uma estrutura formada pelos arquivos main.cpp, pf2.hpp e pf2.cpp.

4.1 Arquivo main.cpp

```
// arquivo main.cpp

#include "pf2.cpp"

int main()
{
  const int t = 6;
  KM EstKM(6);
  EstKM.atribui_p(3);
  EstKM.InsiraElem(3.7, 0, 0);
  EstKM.InsiraElem(9.4, 0, 1);
  EstKM.InsiraElem(5.1, 0, 2);
  EstKM.InsiraElem(2.9, 0, 3);
  EstKM.InsiraElem(7.2, 0, 4);
  EstKM.InsiraElem(1.1, 0, 5);
  EstKM.Ordene();
  EstKM.Falha();
  EstKM.IndSobRisco();
  EstKM.Risco();
  EstKM.STD();
  EstKM.IC95();
  EstKM.Imprima();
}
```

```
// final do arquivo main.cpp
```

4.2 Arquivo pf2.hpp

```
// arquivo pf2.hpp

#ifndef PF2HPP
#define PF2HPP

class KM
{
  float *Obs;
  int   *Delta;
  int   *Falha_v;
  float *IndSobRisco_v;
  float *Risco_v;
  int tam, i, p, x;
  float Var, ICsup, ICinf

public:

  KM (int tam);
  void atribui_p(int temp);
  void InsiraElem(float elem, int censura, int pos);
  void Ordene();
  void Falha();
  void IndSobRisco();
  void Risco();
  void STD();
  void IC95();
  void Imprima();
  ~KM();
};

#endif

// final do arquivo pf2.hpp
```

4.3 Arquivo pf2.cpp

```
// arquivo pf2.cpp

#include "pf2.hpp"
#include <stdio.h>
#include <math.h>

KM::KM(int t)
{
  tam = t;
  Obs = new float[t];
  Delta = new int[t];
```

```
  Falha_v = new int[t];
  IndSobRisco_v = new float[t];
  Risco_v = new float[t];
};

KM::~KM()
{
  delete Obs;
  delete Delta;
  delete Falha_v;
  delete IndSobRisco_v;
  delete Risco_v;
};

void KM::atribui_p(int temp)
{
  p= temp;
}

void KM::InsiraElem(float elem,
int censura, int pos)
{
  Obs[pos] = elem;
  Delta[pos] = censura;
}

void KM::Ordene()
{
  float maior = 0, menor = 0, troca;
  int j, troca1, aux;
  for (i = 0; i < tam; i++)
   {
    if (Obs[i] > maior)

     maior = Obs[i];
   }

   for (i = 0; i < tam; i++)
    {
     menor = maior;

     for (j = i; j < tam; j++)
      {
       if (Obs[j] < menor)
       {
        menor = Obs[j];
        aux = j;
        troca = Obs[i];
        Obs[i] = menor;
        Obs[aux] = troca;
        troca1 = Delta[i];
        Delta[i] = Delta[aux];
```

```
        Delta[aux] = troca1;
       }

       if ((Obs[j] == menor)&&(Delta[j] == 0))
        {
         menor = Obs[j];
         aux = j;
         troca = Obs[i];
         Obs[i] = menor;
         Obs[aux] = troca;
         troca1 = Delta[i];
         Delta[i] = Delta[aux];
         Delta[aux] = troca1;
        }
      }
    }
}

void KM::IndSobRisco()
{
  Obs[tam] = 0;
  for (i = 0; i < tam; i++)
   IndSobRisco_v[i] = tam-i;
}

void KM::Falha()
{
  int k;
  for (i = 0; i < tam; i++)
  {
   k = 1;

   if ((Obs[i] == Obs[i+k])&&(Delta[i] == 1))

    Falha_v[i] = 0;

   if ((Obs[i] == Obs[i+k])&&(Delta[i] == 0))
    {
     for (int j = i; j < tam; j++)
      {
       if (Obs[i] == Obs[i+k])
        {
         Falha_v[i+k] = 0;
         k = k+1;
        }
      }
    }

   if (k > 1)
    {
     Falha_v[i] = k;
     i = i + k;
```

```
    }

   if ((Obs[i] != Obs[i+1])&&(Delta[i] == 1))

    Falha_v[i] = 0;

   if ((i > 0)&&(Obs[i] != Obs[i+1])&&
      (Obs[i] != Obs[i-1])&&(Delta[i] == 0))

    Falha_v[i] = 1;

   if ((i == 0)&&(Obs[i] != Obs[i+1])
      &&(Delta[i] == 0))

    Falha_v[i] = 1;
  }
}

void KM::Risco()
{
  int x = 1;
  Risco_v[0] = 1.0;
  for (i = 0; i < tam; i++)
   {
    if (Falha-v[i] != 0)
     {
      Risco_v[x]=(1-(Falha[i]/
                /IndSobRisco_v[i]))*
                *Risco_v[x-1];
      x = x+1;
     }
   }
}

void KM::STD()
{
  D = 0.0;
  if ((p == 0) || (p == x))
   Var = 0;

  if ((p > 0) && (p < x))
   {
    for (i = 1; i <= p;i++)
     {
      D = D + Falha_v[i-1]/
        /(IndSobRisco_v[i-1]*
        *(IndSobRisco_v[i-1]-Falha_v[i-1]));
     }

    Var = Risco_v[p] * Risco_v[p] * D;
   }
}
```

```
void KM::IC95()
{
  D1 = 0;
  D2 = 0;

  if (( p >= 0 ) && ( p < x ))
   {
    ICsup = Risco_v[p] + 1.96 * sqrt(Var);
    ICinf = Risco_v[p] - 1.96 * sqrt(Var);

    if ((ICsup > 1) || (ICinf < 0))
     {
      for (i = 1; i <= p; i++)
       {
        D1 = D1+Falha_v[i-1]/(IndSobRisco_v[i-1]*
             *(IndSobRisco_v[i-1]-Falha_v[i-1]));

        D2 = D2+log((IndSobRisco_v[i-1]-Falha_v[i-1])/
             /IndSobRisco_v[i-1]);
       }

      Var = D1/(D2 * D2);

      ICsup = exp(exp( 1.96 * sqrt(Var)) * log(Risco_v[p]));
      ICinf = exp(exp(-1.96 * sqrt(Var)) * log(Risco_v[p]));
     }
   }
}

void KM::Imprima()
{
  fprintf (stdout,"\nO vetor de Risco e:\n");
  for (i = 0; i < x; i++)
  fprintf (stdout,"\n%3.4f",Risco_v[i]);
  fprintf (stdout,"\nA Variancia e:%3.4f", Var);
  fprintf (stdout,"\nLim. Sup. Confianca =%3.4f",ICsup);
  fprintf (stdout,"\nLim. Inf. Confianca =%3.4f\n",ICinf);
}

// final do arquivo pf2.cpp
```

5 Resultados

Apresentamos os resultados finais de um estudo do estimador de Kaplan-Meier, onde verificamos seu comportamento em amostras com poucas observações. A implementção do programa em uma linguagem orientada por objeto foi muito bem sucedida. O programa está pronto para ser utilizado em qualquer situação em que se deseja o estimador de Kaplan-Meier, de modo que as observações devem ter no mínimo uma censura (ou do tipo I ou do tipo II), pois este estimador é aplicado somente em dados censurados. É necessário que se tenha no mínimo 5 observações, pois o estimador de Kaplan-

Meier torna-se mais eficiente a medida em que o tamanho da amostra cresce. E para uma amostra muito pequena (<5) esta estimativa pode estar longe da realidade, conforme a literatura.

Foram feitos vários testes com dados reais (ver Weibull, Kalbfleish e Crownder) e comparando os resultados obtidos nesta implementação com os exemplos das literaturas citadas acima, notamos que não houve diferenças nos resultados dos mesmo, o que significa que para estes casos testados, a implementação está correta.

Para ilustrar um resultado obtido, vamos dar um exemplo de aplicação do estimador de Kaplan-Meier. Um laboratório americano esta testando a durabilidade de um novo componente eletrônico utilizado para a fabricação de lâmpadas incandescentes. Para tal, foram colocadas 6 lâmpadas sob teste. Estas estavam ligadas a um computador que registrava o início e o fim do experimento para cada lâmpada, sendo que todas estavam sob as mesmas condições no experimento. O computador registrou 5 tempos de falhas e uma das lâmpadas sofreu uma queda durante o experimento e resultou numa censura do tipo I. Os tempos (em meses) de funcionamento das lâmpadas foram os seguintes: Obs $= [3.7, 9.4, 5.1, 2.9, 7.2c, 1.1]$. O c indica a ocorrência de uma censura.

Com estes dados os resultados obtidos pela implementação foram: Delta $= [0,0,0,0,1,0]$; IndSobRisco_v $= [6,5,4,3,2,1]$; Falha_v $= [1,1,1,1,0,1]$; $\hat{R}(t) = [1.000, 0.833, 0.666, 0.500, 0.333, 0.000]$; para $p = 2$: $\hat{R}(2) = 0.666$, $\mathrm{Var}(\hat{R}(2)) = 0.0417$ e IC95% $= [0.9001; 0.0999]$. Portanto, a probabilidade de uma lâmpada funcionar até o tempo 2 (até 3.9 meses) sem falhar é de 0.66; a variância nesse caso foi de 0.0417 e, com 95% de confiança, podemos afirmar que o intervalo acima pode conter o verdadeiro valor de $R(t)$ (probabilidade exata da lâmpada funcionar até o período de 2.9 meses).

Futuramente pretendemos fazer uma comparação empírica entre esse estimador e o estimador de Nelson-Aalen [2], considerando amostras de tamanho pequeno. Semelhante estudo foi realizado para o caso de grandes conjuntos de observações [2].

Referências

[1] W.S. Borges, E.A. Colosimo, M.A. Freitas Métodos Estatísticos e Melhoria da Qualidade: Construindo Confiabilidade em Produtos,*ABE*, 1996.

[2] G.A. Bohoris. Comparison of the Cumalative-Hazard and Kaplan-Meier Estimators of the Survivor Function, *IEEE Reliability*, Vol.43(2): 230-231, 1994.

[3] Crowder, M.J.; Kimber, A.C. Statistical Analysis of Reliability Data. London: Chapman&Hall 1991.

[4] Kalbfleisch, J.D. and Prentice, R.L. The Statistical Analysis of Failure Time Data. New York: Wiley 1980.

[5] Weibull, W. A Statistical Representation of Fatigue Failure in Solids. Royal Institute Technology, (Stockholm) 1954.

Modelagem de Dispositivos Elétricos Utilizando Redes Neurais

José H. Ferreira[1]

[1]Departamento de Eletricidade - FUNREI,
Praça Frei Orlando, 170
36300-000 - Sã João del Rei - MG
E-mail: jhissa@funrei.br

Resumo

Este trabalho apresenta o estudo e a implementação de uma classe para uma rede neural artificial (RNA) do tipo multicamadas (MLP) treinada com o algoritmo *backpropagation*. Uma aplicação deste trabalho é a modelagem de dispositivos elétricos em programas de simulação de sistemas de energia, aproveitando a característica adaptativa destas redes.

1 Introdução

A complexidade do modelo físico pode levar a uma ineficiência computacional durante a sua simulação. Por outro lado, a utilização de modelos equivalentes, linearizados, pode trazer um resultado indesejado em razão da aproximação do modelo.

As técnicas de modelagem utilizadas para a simulação de circuitos elétricos se baseiam nos aspectos físicos do dispositivo — regidos pelas equações físicas do seu comportamento, ou pela sua característica idealizada — que procura aproximar a sua característica não linear por um modelo linearizado [Ferreira, 94].

A característica adaptativa da rede neural artificial [Haykin, 99] traz um novo paradigma na solução de problemas, substituindo o trabalho de programação pelo aprendizado. Elas se tornam assim um modelo viável para a solução de uma ampla gama de problemas.

Dentre as principais aplicações das redes neurais podemos citar [Neural Network Toolbox, 94]:

- Aeroespacial: autopilotos de alto desempenho e confiabilidade, simulações de vôos, detecção de falhas e simulação de componentes de aeronaves.
- Automotiva: sistemas de direção inteligente.
- Bancos: leituras de cheques, documentos e análise de tendências.
- Defesa: reconhecimento facial, armas inteligentes, seleção de alvos, processamento de imagens de sistemas de reconhecimento (radar, etc) e identificação.
- Eletrônica: predição de seqüências de código, controle do processo de criação e fabricação de circuitos integrados (CI), síntese de voz e visão artificial.
- Financeira: análise financeira, previsão de valores e gerênciamento de investimentos.
- Manufatura: controle, análise e acompanhamento do processo de manufatura, projeto de novos produtos, diagnóstico em tempo real, identificação de partículas e inspeção visual da qualidade, manutenção de máquinas.

- Médica: análise de células cancerosas, análise de exames de raio-x, tomografia e ressonância eletromagnética, projeto de próteses, otimização do tempo de transplantes, redução das despesas hospitalares e melhora da qualidade, observação e testes de emergências.
- Robótica: controle de trajetória, máquinas de carga automatizadas, manipuladores robóticos e sistemas de visão.
- Fala: reconhecimento de fala, compressão de voz, classificação de vogais e síntese de voz.
- Telecomunicações: compressão de imagem e dados, sistemas de informação automático, tradução em tempo real.

As principais características das RNA que serão exploradas na realização do presente projeto podem ser resumidas a seguir [Haykin, 99]:

- Não linearidade: é uma característica importante, principalmente na modelagem de dispositivos físicos.
- Mapeamento de Entrada-Saída (I/O): representa um paradigma importante de aprendizado supervisionado que pode ser explorado para o reconhecimento dos padrões do dispositivo a ser modelado.
- Adaptabilidade: é a capacidade das redes neurais de se adaptarem a eventuais variações ambientais, adaptando-se a novas condições de operação.
- Tolerância a Falhas: permite uma robustez computacional do modelo.

O algoritmo *backpropagation (backprop*) foi inicialmente proposto por Werbos em 1974 e popularizado por Rumelhart, Hinton e Williams em 1986 [Mehrotra, 97]. É baseado no método do gradiente descendente e utiliza, em geral, a função do erro médio quadrático. É utilizado no treinamento supervisionado de redes *feedforward* multicamadas em problemas de classificação de classes não linearmente separáveis [Haykin, 99] e como aproximador de funções [Mehrotra, 97]. A função de ativação utilizada é não linear, porém de característica monotônica e diferenciável, como a tangente hiperbólica ou a função sigmoidal. O número de camadas da rede é proporcional à complexidade do problema.

O algoritmo *quickprop* é uma modificação do *backprop* na qual o tamanho do ajuste dos pesos é determinado assumindo que a superfície do erro é quadrática e que a sua derivada em relação a um dos pesos é independente em relação aos demais pesos conectados àquele nó.

A principal característica do algoritmo *rprop* é que o valor do ajuste não é função da amplitude do gradiente. Se o sinal da derivada para um determinado peso permanece o mesmo por várias iterações, então a amplitude do ajuste aumenta. Se o sinal oscila, então a amplitude do ajuste diminui.

O algoritmo *Levenberg-Marquardt* é o mais rápido para o treinamento de redes multicamadas de tamanho moderado, mesmo considerando uma inversão matricial a cada iteração. Porém, se a ordem da rede é elevada, o algoritmo se torna impraticável nos computadores atuais.

O algoritmo do *Gradiente-Conjugado* requer uma pesquisa linear a cada iteração. Existem diversas implementações deste método.

Outras heurísticas podem ser empregadas. A escolha inicial dos pesos deve ser randômica e de pequeno valor. A taxa de aprendizado é normalmente inicializada com um valor elevado nas primeiras iterações e rapidamente diminuída, pois os estudos demonstram que pequenas alterações nos pesos, quando a rede estiver convergindo para um mínimo local, evitam oscilações em seus valores.

Seja a estrutura mostrada na Figura 1.1, na qual os nós são particionados em camadas numeradas de 0 (nó de entrada) a L (nó de saída). Uma rede *backprop* é caracterizada por $L \geq 2$ e contém camadas ocultas numeradas de 1 a $L-1$.

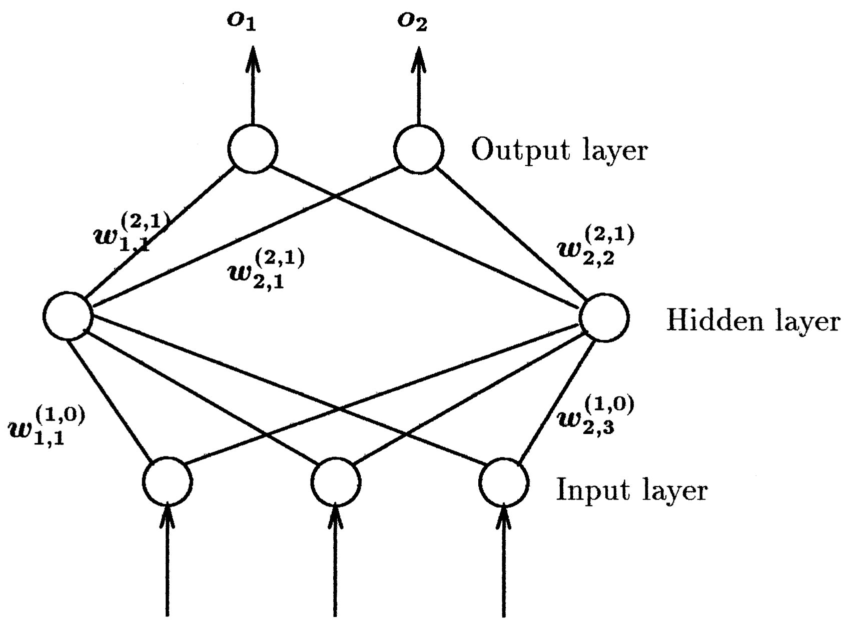

Figura 1.1: Estrutura Básica da Rede *backprop*

Considere um conjunto de treinamento cuja dimensão seja P, dado por:

$$\{(x_p, d_p) : p = 1, \ldots, P\}$$

onde x e d representam a entrada e a saída desejada para a rede. A saída da rede é formada por um vetor de dimensão K dado por:

$$o_p = (o_{p,1}, o_{p,2}, \ldots, o_{p,K})$$

O treinamento consiste em modificar os pesos das ligações dos nós com o objetivo de minimizar a soma dos erros médios quadráticos entre a saída desejada e a obtida.

$$\sum_{p=1}^{P} \sum_{j=1}^{K} (\ell_{p,j})^2 \tag{1}$$

onde

$$(\ell_{p,j})^2 = (o_{p,j} - d_{p,j})^2. \tag{2}$$

Tem-se:

- o *i-ésimo* nó da camada de entrada apresenta um valor de $x_{p,i}$ para o *p-ésimo* padrão.
- a entrada do *j-ésimo* nó da camada oculta é:

$$\nu_j^{(1)} = \sum_{i=0}^{n} \omega_{j,i}^{(1,0)} x_{p,i}, \tag{3}$$

onde o peso $\omega_{j,i}^{(1,0)}$ representa a conexão entre o *i-ésimo* nó da camada de entrada e o *j-ésimo* nó da camada oculta.

- a saída do *j-ésimo* nó na camada oculta é:

$$x_{p,j}^{(1)} = \phi(\sum_{i=0}^{n} \omega_{j,i}^{(1,0)} x_{p,i}) \tag{4}$$

- a entrada do *k-ésimo* nó da camada de saída é:

$$\nu_k^{(2)} = \sum_j \omega_{k,j}^{(2,1)} x_{p,j}^{(1)} \tag{5}$$

onde o peso $\omega_{k,j}^{(2,1)}$ representa a conexão entre o *j-ésimo* nó da camada oculta e o *k-ésimo* nó da camada de saída.

- a saída do *k-ésimo* nó na camada de saída é

$$o_{p,k} = \phi(\sum_j \omega_{k,j}^{(2,1)} x_{p,j}^{(1)}) \tag{6}$$

onde ϕ é uma sigmóide.

- a saída desejada do *k-ésimo* nó na camada de saída é $d_{p,k}$ e o erro quadrático correspondente é: $\ell_{p,j}^2 = |d_{p,j} - o_{p,j}|^2$.

Para minimizar a função erro para o *p-ésimo* padrão apresentado à entrada da rede, $E_p = \sum_k (\ell_{p,k})^2$, através do gradiente descendente, a alteração do peso deve ser $-\partial E/\partial\omega$, o que resulta nas equações:

$$\Delta\omega_{k,j}^{(2,1)} \propto \left(\frac{-\partial E}{\partial \omega_{k,j}^{(2,1)}} \right) \tag{7}$$

e

$$\Delta\omega_{j,i}^{(1,0)} \propto \left(\frac{-\partial E}{\partial \omega_{j,i}^{(1,0)}} \right) \tag{8}$$

Pela aplicação da *regra da cadeia*, chegam-se às equações para as correções dos pesos da rede,

$$\Delta\omega_{k,j}^{(2,1)} = \eta \cdot \delta_k \cdot x_j^{(1)} \tag{9}$$

para a camada de saída e

$$\Delta\omega_{j,i}^{(1,0)} = \eta \cdot \delta_j \cdot x_i^{(1)} \tag{10}$$

para a camada oculta. η é denominado *taxa de aprendizado*. δ_k e δ_j são denominados gradientes locais às camadas k e j, respectivamente, e são determinados a partir das equações:

$$\delta_k = (d_k - o_k) \cdot \phi'(\nu_k^{(2)}) \tag{11}$$

e

$$\delta_j = \left(\sum_k \delta_k \cdot \omega_{k,j}^{(2,1)} \right) \cdot \phi'(\nu_j^{(1)}) \tag{12}$$

Se a função de ativação é sigmoidal, tem-se

$$\phi'(\nu) = \phi(\nu)(1 - \phi(\nu)) \tag{13}$$

Substituindo (13) em (11) e (13) em (12), resulta:

$$\delta_k = (d_k - o_k) \cdot o_k(1 - o_k) \tag{14}$$

e

$$\delta_j = \sum_k \delta_k \cdot \omega^{(2,1)}_{k,j} \cdot x^{(1)}_j (1 - x^{(1)}_j). \tag{15}$$

Pode-se ainda aplicar um *momentum* para prevenir a rede de convergir em algum mínimo local. Se as alterações nos pesos dependerem da média do gradiente do erro quadrático em uma pequena região, ao invés do gradiente em um ponto, a regra do aprendizado (delta), modificada com a inclusão do *momentum* passa a ser:

$$\Delta\omega_{k,j}(t+1) = \eta \cdot \delta_k \cdot x_j + \alpha \cdot \Delta\omega_{k,j}(t) \tag{16}$$

O parâmetro α é denominado *momentum* e ajusta a contribuição do ajuste, do instante anterior, no instante atual. A alteração nos pesos não depende agora apenas do gradiente atual. Sua escolha depende da aplicação. Quando $\alpha \approx 0$ a história passada não tem muito efeito na alteração dos pesos, enquanto que se $\alpha \gg \eta$ o erro atual tem pouco efeito na alteração dos pesos.

2 Implementação das Classes

Como forma de exercitar o reuso de código e estudar a aplicação dos conceitos de herança e polimorfismo, este trabalho se baseou nas classes propostas por [Rogers, 97]. Outras implementações [Park, 98] e [Wax, 98] também foram analisadas.

O treinamento da rede é realizado antes da validação dos resultados. O erro de convergência deve ser escolhido para minimizar o tempo de treinamento sem afetar o resultado desejado. Em geral se procura a maior generalização permitida para uma determinada topologia ou então aquela aceita para uma determinada aplicação.

Para validação do algoritmo foi implementada a solução do problema do ou-exclusivo (um conjunto de pontos não separáveis linearmente. Os resultados obtidos foram comparados com aqueles fornecidos pelo programa Matlab [Neural Network Toolbox, 94], mostrados na Figura 2.1.

Pode-se observar que os resultados são equivalentes [1].

Um próximo objetivo deste trabalho será a realização de um treinamento por lote, no qual a entrada e a saída de dados se farão a partir de arquivos ASCII.

As implementações podem ser obtidas com o autor, a pedido.

A formalização básica da estrutura é realizada pelas classes `Base_Node` e `Base_Link`, a partir das quais o algoritmo *backprop* se desenvolve. Nelas estão a forma como os nós da rede são interligados e os apontadores para a estrutura criada (as conexões de cada nó).

A classe `Adaline_Node` é utilizada para formar cada um dos nós da rede, a partir das definições das classes `Base_Node` e `Base_Link`, implementando uma rede *feedforward*. Os valores de polarização dos nós são determinados a partir da classe `Bias_Node`. A classe `Input_Node` gerencia as entradas do nó.

A forma de entrada - os padrões, são formados a partir da classe Pattern, definindo as funções membros básicas para lidar com a entrada da rede: entrada e saída de dados, impressão, etc.

[1] As rotinas do Matlab implementam uma variação da taxa de aprendizado para acelerar a convergência. O programa utilizado utiliza uma taxa de aprendizado constante

```
Rede criada e nao treinada. Saidas obtidas para as entradas fornecidas:
     1     0
     1     1
     1     1
     1     0

TRAINBPX: 0/5000 epochs, lr = 1.35, SSE = 1.1158.

TRAINBPX: 100/5000 epochs, lr = 98.8535, SSE = 0.484195.

TRAINBPX: 107/5000 epochs, lr = 139.097, SSE = 0.0096684.

Rede criada e treinada. Saidas obtidas para as entradas fornecidas:
     0     0
     1     1
     1     1
     0     0

Rede criada e nao treinada. Saidas obtidas para as entradas fornecidas:
     1     0
     1     1
     1     1
     1     0

TRAINBPX: 0/5000 epochs, lr = 1.35, SSE = 1.1158. TRAINBPX: 100/5000 epochs,
lr = 98.8535, SSE = 0.484195. TRAINBPX: 107/5000 epochs, lr = 139.097, SSE =
0.0096684. Rede criada e treinada. Saidas obtidas para as entradas
fornecidas:
     0     0
     1     1
     1     1
     0     0


0.   0/4   Error: 0.456909775098387 1000.   0/4   Error: 0.243252886363585

2000.   2/4   Error: 0.223851091835278 3000.   2/4   Error: 0.22135941925437

4000.   0/4   Error: 0.15176765776989 5000.   0/4   Error: 0.062178381647769

6000.   3/4   Error: 0.0369859606163587 7000.   3/4   Error:
0.0259180852091161

Pattern:   0  Input: (0,0)   Backprop: (0)    Actual: (0) Pattern:   1 Input:
(0,1)   Backprop: (1)    Actual: (1) Pattern:   2 Input: (1,0) Backprop: (1)
Actual: (1) Pattern:   3  Input: (1,1) Backprop: (0) Actual: (0)
```

Figura 2.1: Comparação dos Resultados Obtidos com o Programa Matlab

A rede multicamadas é formada a partir das classes `Base_Node`, `Base_Link` e `Adaline_Node` utilizando as funções implementadas no arquivo *backprop.h* e o programa principal está implementado em *backpro2.cpp*.

3 Conclusões

A adaptabilidade e a robustez da rede neural artificial aplicada na modelagem de dispositivos elétricos pode resultar em uma ferramenta poderosa no desenvolvimento de programas para a simulação de sistemas de energia elétrica, principal problema encontrado no EMTP (*Electromagnetic Transients Program*) [Araújo, 93], considerado um padrão na área. A reestruturação do EMTP, utilizando algoritmos de modelagem baseados em RNA e programação orientada a objeto (*OOP*) [Ferreira, 99] pode originar uma nova geração de simuladores de sistemas de energia ainda mais flexíveis e genéricos.

A utilização das classes propostas por [Rogers, 97] ilustram o conceito de reuso de software, utilizando os conceitos de herança e polimorfismo. Podem assim serem adaptadas para atender aos objetivos propostos neste trabalho.

As duas maiores dificuldades encontradas no presente trabalho estiveram relacionadas com o tempo insuficiente para um perfeito domínio das classes e na escolha da melhor configuração da rede necessária para atender ao problema proposto.

Para a modelagem de dispositivos mais complexos, não lineares, deve-se utilizar uma rede com maior número de nós e, provavelmente, camadas intermediárias. Se o número for muito elevado será necessária a utilização das heurísticas e o desenvolvimento das correspondentes funções para utilização no programa.

Uma alternativa interessante pode ser o desenvolvimento de uma classe apenas de heurísticas que podem ser chamadas pela função de treinamento, de forma transparente ao usuário.

Referências

[Araújo, 93] Araújo, A. E. A.; Dommel, H. W. e Marti, J. R., *Simultaneous Solution Of Power And Control Systems Simulator IEEE Trans. On Power Systems, Vol. 8, N^o 4, November 1993.*

[Ferreira, 94] Ferreira, José Hissa, *Simulação e Estudo de um Condicionador Ativo de Potência. Dissertação de Mestrado, UFMG, 1994.*

[Ferreira, 99] Ferreira, José Hissa, *Projeto de Pesquisa submetido ao Programa de Pós-Graduação em Engenharia Elétrica da UFMG, 1999.*

[Haykin, 99] Haykin, Simon, *Neural Networks — A comprehensive Foundation, second edition. Prentice-Hall, 1999.*

[Mehrotra, 97] Mehrotra, K.; Mohan,C. K. e Ranka, S., *Elements of Artificial Neural Networks. MIT Press, 1997.*

[Neural Network Toolbox, 94] Demuth, H. e Beale, M., *Mathworks Inc., 1994.*

[Rogers, 97] Rogers, Joey, *Object - Oriented Neural Networks In C++. Academic Press, 1996.*

[Park, 98] Youngchoon, Park, The Class hierarchy and the Network Class.
`URL:www.eas.asu.edu/~mmis/bp`

[Wax, 98] Wax, Michael J., The Backpropagation Network Training Software.
`URL:www.michaelwax.com/neuralmail.html`

Um Exemplo de Utilização de Estatística Espacial na Previsão de Séries Temporais

Otaviano F. Neves[1]

[1]Departamento de Estatística - ICEx - UFMG,
Caixa Postal 702,
30123-970 - Belo Horizonte - MG
E-mail: otaviano@est.ufmg.br

Resumo

Apresentamos a utilização de medidas de variabilidade espacial na previsão de series temporais, bem como a implementação de um progama em C++ que calcule estas medidas.

1 Introdução

Dentre as várias metodologias existentes, no caso de análise no domínio do tempo, encontra-se o método de Box e Jenkins [1], que basicamente se resume na identificação de um modelo ARIMA (modelos autorregressivo e de médias móveis com diferenciação simples), que descreve o comportamento da série, o que é feito através da informação das funções de correlogramas, na estimação de parâmetros do modelo escolhido e na verificação de qualidade do ajuste.

Uma outra forma de se tratar uma série temporal é pela técnica de estatística espacial [2], que utiliza a informação do variograma ou de outras medidas de variabilidade como o madograma e o rodograma [3], como forma principal de identificação do modelo e predição de valores.

No caso especifico de series temporais, os estágios de identificação e estimação dos modelos de variograma, madograma e rodograma podem, na grande maioria dos casos de interesse práticos, ser suprimidos, uma vez que seus valores experimentais contêm toda informação necessária para a construção das equações de predição.

O objetivo deste projeto é implementar um programa em C++ que, dada uma serie temporal de entrada, possa gerar como saída medidas de variograma, rodograma e madograma experimentais.

Apresentaremos a seguir a metodologia de geoestatistica, a abordagem computacional do problema, bem como as considerações finais.

2 Metodologia de Estatística Espacial

Para se proceder a análise dos dados através de métodos clássicos de estatística espacial, ou mais precisamente a análise via métodos de geoestatística [2], é necessária a imposição de que o processo estocástico gerador dos dados amostrais seja intrinsicamente estacionário e isotrópico, ou seja,deve satisfazer às seguintes condições:

i)$E[Z(t)] = \mu, \forall t \in T$;

ii)$\mathrm{Var}[Z(t) - Z(s)] = 2\gamma(t-s), \forall t \neq s \in T$.

Assim sendo, o processo tem média constante ao longo do tempo, e para cada t e s a variância das diferenças $[Z(t) - Z(s)]$ é uma função que depende apenas da distância no tempo das observações

$Z(t)$ e $Z(s)$. A quantidade $2\gamma(.)$ é chamada de variograma do processo estocástico gerador da série e $\gamma(.)$ é chamado de semi-variograma. O conhecimento da função de variograma tem uma importância fundamental na construção do modelo de projeção obtido pelos métodos de geoestatística, uma vez que as equações de previsão são todas funções dos valores do variograma do processo estocástico gerador dos dados. Embora o variograma seja a função mais conhecida para descrever a variabilidade espacial dos dados, existem outras medidas. As mais conhecidas são o madograma e o rodograma, que, para um processo intrinsicamente estacionário e isotrópico, são definidas respectivamente por:

$$2\Gamma_w(t-s) = E[|Z(t) - Z(s)|^w], \forall t \neq s, \in T, \tag{1}$$

sendo que, para $w = 1$, tem-se o madograma e para $w = 1/2$, o rodograma. De acordo com esta notação, $w = 2$ representa o variograma. Estas medidas alternativas são mais robustas no que se refere a influência dos valores discrepantes e do tamanho da amostra na identificação de modelos de projeção espacial. Na prática, as funções de variograma, madograma e rodograma são estimadas a partir dos dados amostrais. Os estimadores clássicos não viciados usuais, para o caso de séries temporais, são definidos por:

$$2\hat{\Gamma}_w(S) = \frac{\sum^{n-s}[|Z(t) - Z(s)|]^w}{n-s} \tag{2}$$

onde, para $w = 1$, temos o estimador do madograma, para $w = 1/2$, o rodograma e para $w = 2$, o variograma. Em geral as estimativas obtidas são chamadas de madograma, rodograma e variograma experimentais. Através do cálculo destes estimadores na amostra coletada, identifica-se o modelo de variograma, madograma e rodograma do processo estocástico gerador dos dados amostrais. Identificado o modelo, prossegue-se com a estimação dos parâmetros do mesmo. O modelo assim estimado é utilizado nas equações de projeção. No nosso caso específico, apenas os medidas de variabilidades experimentais são suficientes para se proceder a construção das equações de predição

Na seção seguinte apresentaremos um esboço da implementação de um programa em C++ para resolver o problema do cálculo das medidas de variabilidade espacial.

3 Implementação do Programa para o Cálculo das Medidas de Variabilidade Espacial

O motivo de se implementar um programa em C++, que dada uma serie temporal de entrada, possa gerar como saída, medidas de variograma, rodograma e madograma experimentais, é pela falta de pacotes que resolvessem este determinado tipo de problema.

O programa principal, que calcula as medidas de variabilidade, e a biblioteca, que define a classe criada, são apresentados nas Figuras 3.1 e 3.2.

A título de exemplo, foram utilizadas 100 observações da série do número de manchas solares de Wolfer, no período de 1770 a 1869. Esta é uma série bastante conhecida na área de séries temporais, tendo sido tratada por muitos autores.

Os 15 primeiros valores da serie de entrada, bem como os resultados obtidos pelo programa, que deve ser chamado pelo comando `Medvar < sunspots.dat > medidas.dat`, são apresentados na Figura 3.3.

```
//**********************
//Arquivo mainMV.cpp
//**********************
#include <stdio.h>
#include "serie.cpp"

int main() {
   Serie MinhaSerie;
   MinhaSerie.Leia();
   MinhaSerie.GeraVariograma();
   fprintf(stdout, "Variograma:\n");
   MinhaSerie.ImprimeMedVar();
   MinhaSerie.GeraMadograma();
   fprintf(stdout, "Madograma:\n");
   MinhaSerie.ImprimeMedVar();
   MinhaSerie.GeraRodograma();
   fprintf(stdout, "Rodograma:\n");
   MinhaSerie.ImprimeMedVar();
}
```

Figura 3.1: Programa Principal

```
//*******************
//serie.hpp
//******************
//
//classe Serie
//
#ifndef _SERIE_HPP_
#define _SERIE_HPP_

const int SERIE_TAM_MAX = 10000;

class Serie{
private:
  int tamMax;
  float *vetor;
  int tamSerie;
  float *medVar;
public:
//construtores
  Serie();
//default
  Serie(int Tam);
//generico
  Serie(const Serie &serie1);
//copiador/inicializador
  ~Serie();

  void Leia();
// le^ a serie
  void GeraVariograma();
// gera variograma completo
  void GeraMadograma();
// gera madograma completo
  void GeraRodograma();
// gera rodograma completo
  void ImprimeMedVar();
// imprime as medidadas de
//variabilidade
};
#endif
```

Figura 3.2: Código em C++ para a Classe Serie

```
tempo  serie
    1  101
    2   82
    3   66
    4   35
    5     31
    6    7
    7   20
    8   92
    9    154
   10    125
   11     85
   12     68
   13     38
   14     23
   15     10

Variograma Madograma Rodograma
    504,09      3,70     17,14
  1.562,33      5,14     31,00
  2.579,91      6,02     41,43
  3.283,65      6,48     47,88
  3.593,99      6,69     50,75
  3.455,24      6,61     49,82
  2.984,98      6,45     46,44
  2.370,76      5,92     40,11
  1.754,08      5,11     31,86
  1.435,04      4,65     26,91
  1.459,34      4,39     25,40
  1.802,73      4,58     28,41
  2.277,09      5,26     34,86
```

Figura 3.3: Saída do Programa

4 Considerações Finais

Através da análise dos resultados obtidos, podemos considerar que o objetivo deste projeto foi alcançado, pois foram implementados os métodos da classe `Serie` e o programa que calcula as medidas de variabilidade espacial em séries temporais, ou seja, um programa eficiente, devidamente documentado e que pode ser entendido e estendido pelo usuário. Cumpre destacar que um trabalho futuro possível é implementar um método para a classe `Serie` que faça as previsões n passos à frente, bem como seus respectivos intervalos de confiança.

Referências

[1] Box, G. P. and Jenkins, G. M *Time series analysis: forecasting and control.* New York: Holden Day, 1976.

[2] Cressies, N. *Statistics for spacial datas.* New York: John Wiley e Sons, 1992.

[3] Mingoti, S. A. As funcões de madograma e rodograma como alternativas para descrever a variabilidade dos dados. *Revista Escola de Minas (REM).* Ouro Preto, 50, 2, 71-74, 1996.

Geração de Números Aleatórios

Maristela D. Oliveira[1]

[1]Departamento de Estatística - ICEx - UFMG,
Caixa Postal 702,
30123-970 - Belo Horizonte - MG
E-mail: maridias@est.ufmg.br

Resumo

Apresentamos aqui uma biblioteca para geração de números aleatórios, segundo determinadas distribuições contínuas, desenvolvida em linguagem C++.

1 Introdução

Para gerar números aleatórios, muitas linguagens têm internamente uma função, cuja saída é uma seqüência de números pseudo-aleatórios, isto é, uma seqüência de números que se assemelha a uma amostra aleatória. Muitos geradores de números aleatórios começam com um valor inicial X_0, chamado de semente, e então calculam recursivamente os valores seguintes, através de transformações matemáticas adequadas, gerando uma amostra de uma distribuição de interesse.

Como normalmente está disponível apenas a distribuição uniforme, é de interesse disponibilizar outras distribuições.

Para desenvolver uma biblioteca que disponibilize essas distribuições, utitilizaremos basicamente dois métodos, (i) o método da transformação inversa e (ii) o método da rejeição [1, 2, 3]. Esperamos gerar amostras aleatórias e compará-las às geradas por pacotes estatísticos amplamente utilizados (por exemplo, o MINITAB), através de testes de ajustes às respectivas distribuições. Esperamos concluir se tais geradores são realmente confiáveis.

A seguir apresentamos os métodos citados.

2 Os Métodos

2.1 Transformação Inversa

Proposição:

Seja U, uma variável aleatória com distribuição uniforme em $(0, 1)$. Para qualquer função de distribuição contínua F, se definirmos a variável aleatória $Y = F^{-1}(U)$, então Y tem função de distribuição F [1].

Apresentamos a seguir um algoritmo para gerar uma amostra de tamanho n, da variável aleatória Y.

Apesar de ser de fácil aplicação, este método apresenta o inconveniente de necessitar da expressão analítica da inversa da função de distribuição de Y [2].

algoritmo
  $i = 0$
  repita
    $i \leftarrow i + 1$
    gerar um valor u de U
    $y_i \leftarrow F^{-1}(u)$
  até $i = n$
fim algoritmo

2.2 Rejeição

Suponha que temos uma método para gerar uma variável aleatória X, com função densidade de probabilidade $g(x)$. Podemos usá-la como base para simulação da variável aleatória contínua Y tendo densidade $f(x)$, simulando Y a partir de g e então aceitando este valor simulado com uma probabilidade proporcional a $f(X)/g(X)$. Especificamente, seja c uma constante tal que:

$$\frac{f(y)}{g(y)} \leq c$$

para todo y.

Temos então os seguintes passos para gerar uma variável aleatória Y, com densidade f.

algoritmo
  $i = 0$
  repita
    simule X, tendo densidade g
    gere uma variável aleatória U, uniforme em $(0, 1)$
    se $U \leq \frac{f(y)}{g(y)}$ **então**
      $Y \leftarrow X$
      $i \leftarrow i + 1$
    fim se
  até $i = n$
fim algoritmo

Proposição:

A variável aleatória Y, gerada pelo método da rejeição, tem função densidade de probabilidade f [1, 3].

3 O programa

A seguir apresentamos os "objetos" necessários à implementação da biblioteca. Optamos por utilizar o método da rejeição para gerar a distribuição normal padrão e o método da transformação inversa para gerar a uniforme em (a,b) e a exponencial. A partir destas, chegaremos às demais distribuições, através de relações matemáticas adequadas [4]. Vale ressaltar aqui que o programa assumirá como desenvolvida uma função capaz de gerar números aleatórios que seguem a distribuição uniforme em (0,1). Este gerador se encontra no arquivo "rand.c".

3.1 A Classe

```
// arquivo numaleat.hpp

#ifndef _ALEATORIOS_
#define _ALEATORIOS_

class Aleatorios{
public:
  Aleatorios() {}
  ~Aleatorios() {};

float GereUniforme
 (int *semente, float liminf, float limsup);
ListaR GereUniforme(int *semente,
 int numElem, float liminf, float limsup);

float GereExponencial
 (int *semente, float media);
ListaR GereExponencial
 (int *semente, int numElem, float media);

float GereGamma
 (int *semente, int alfa, float beta);
ListaR GereGamma
 (int *semente, int numElem, int alfa, float beta);

float GereNormal(int *semente,
float media, float variancia);
ListaR GereNormal(int *semente,
 int numElem, float media, float variancia);

float GereBeta
 (int *semente, float alfa, float beta);
ListaR GereBeta(int *semente,
 int numElem, float alfa, float beta);

float GereWeibull
 (int *semente, float alfa, float beta);
ListaR GereWeibull(int *semente,
 int numElem, float alfa, float beta);

float GereLognormal
 (int *semente, float mi, float sigma);
ListaR GereLognormal(int *semente,
 int numElem, float mi, float sigma);
};
#endif

// fim do arquivo numAleat.hpp}
```

3.2 Implementações

```
//arquivo numAleat.cpp
#include "rand.c"
#include "listaR.cpp"
#include "numaleat.hpp"

float Aleatorios::GereUniforme
 (int *semente, float liminf, float limsup) {
  float unif =
  (urand(semente)*(limsup - liminf)) + liminf;
  return unif;
}

ListaR Aleatorios::GereUniforme
 (int *semente, int numElem,
 float liminf, float limsup) {
  ListaR amostraUniforme(numElem);
  float unif;
  for ( int i = 0; i < numElem; i++) {
    unif = GereUniforme(semente, liminf, limsup);
    amostraUniforme.Insere(unif,
       amostraUniforme.Final());
  }
  return amostraUniforme;
}

float Aleatorios::GereExponencial
 (int *semente, float media) {
  float exp = -(media)*log(urand(semente));
  return exp;
}

ListaR Aleatorios::GereExponencial
 (int *semente, int numElem, float media) {
  ListaR amostraExponencial(numElem);
  float exp;
  for ( int i = 0; i < numElem; i++) {
    exp = GereExponencial(semente, media);
    amostraExponencial.Insere(exp,
      amostraExponencial.Final());
  }
  return amostraExponencial;
}

float Aleatorios::GereGamma
 (int *semente, int alfa, float beta) {
  float amostraGamma;
  float *amostraExponencial = new float[alfa];
  int i;

  for (i = 0; i < alfa; i++) {
```

```
    amostraExponencial[i]=GereExponencial
                          (semente, beta);
  }
  amostraGamma = 0.0;
  for (i = 0; i < alfa; i++) {
    amostraGamma += amostraExponencial[i];
  }
  return amostraGamma;
}

ListaR Aleatorios::GereGamma
 (int *semente, int numElem, int alfa, float beta){
  ListaR amostraGamma(numElem);
  float gamma;
  for ( int i = 0; i < numElem; i++) {
    gamma = GereGamma(semente, alfa, beta);
    amostraGamma.Insere(gamma, amostraGamma.Final());
  }
  return amostraGamma;
}

float Aleatorios::GereNormal
 (int *semente, float media, float variancia) {
  float y1, y2;

  while (1) {
    y1 = -log(urand(semente));
    y2 = -log(urand(semente));
    if (y2 - (y1 - 1)*(y1 - 1)/2 >= 0) {
      if (urand(semente) > 0.5)
       return -y1;
      else
       return y1;
    }
  }
}

ListaR Aleatorios::GereNormal
 (int *semente, int numElem,
 float media, float variancia) {
  ListaR amostraNormal(numElem);
  float normal;
  for ( int i = 0; i < numElem; i++) {
    normal = GereNormal(semente, media, variancia);
    amostraNormal.Insere(normal,
         amostraNormal.Final());
  }
  return amostraNormal;
}

// fim do arquivo numAleat.cpp
```

3.3 Programa Principal

```
// arquivo main_num.cpp

#include <stdio.h>
#include "numaleat.cpp"

int main(){

  int semente = 0;
  float elemento;
  ListaR elementos;
  Aleatorios Amostra;

  fprintf (stdout, "Gerando uniformes\n");
  elemento = Amostra.GereUniforme
             (&semente, 0.0, 1.0);
  fprintf (stdout, "Elemento:\n %f\n", elemento);
  elementos = Amostra.GereUniforme
             (&semente, 500, 0.0, 1.0);
  fprintf (stdout, "Elementos:\n");
  elementos.Imprime();

  fprintf (stdout, "Gerando exponenciais\n");
  elemento = Amostra.GereExponencial
            (&semente, 1.0);
  fprintf (stdout, "Elemento:\n %f\n", elemento);
  elementos = Amostra.GereExponencial
            (&semente, 500, 1.0);
  fprintf (stdout, "Elementos:\n");
  elementos.Imprime();

  fprintf (stdout, "Gerando gamas\n");
  elemento = Amostra.GereGamma
            (&semente, 1.0, 10);
  fprintf (stdout, "Elemento:\n %f\n", elemento);
  elementos = Amostra.GereGamma
            (&semente, 500, 1.0, 10);
  fprintf (stdout, "Elementos:\n");
  elementos.Imprime();

  fprintf (stdout, "Gerando normais\n");
  elemento = Amostra.GereNormal
            (&semente, 0.0, 1.0);
  fprintf (stdout, "Elemento:\n %f\n", elemento);
  elementos = Amostra.GereNormal
            (&semente, 500, 0.0, 1.0);
  fprintf (stdout, "Elemento:\n");
  elementos.Imprime();

  return 0;
}
```

```
// fim do arquivo main.cpp
```

4 Conclusão

Estamos disponibilizando uma biblioteca em C++, capaz de gerar amostras pseudo-aleatórias de algumas distribuições contínuas. Chamamos a atenção para a limitação imposta pelo fato de que a distribuição Gama gerada está restrita ao caso em que o parâmetro α seja um número inteiro. O programa principal, arquivo “main.cpp”, disponibilizado neste relatório gera um número ou uma lista com 500 valores das distribuições Uniforme em $[0, 1]$, Exponencial, com média 1.0, Gama (1.0,10) e da Normal padronizada. O usuário que necessitar de distribuições com outros parâmetros deve fazer as devidas modificações neste arquivo. Observemos que ainda faltam implementar as classes capazes de gerar amostras das distribuições Beta, Weibull e Lognormal, o que faremos em tempo oportuno.

Referências

[1] Ross, S. M. (1997) *A First Course in Probability*. Rio de Janeiro: Prentice Hall do Brasil.

[2] Santos, M. A. C. (1998) Noções de Estatística Computacional. Relatório Técnico RTE 03/98, EST-ICEx-UFMG. Belo Horizonte, 1998.

[3] Johson, N.L., Kotz, S. (1994) *Continuos Univariate Distributions*. vol. 1-2, EUA: A Wiley-Interscience Publication,

[4] Cassela, G., Berger, R. L. (1994) *Statistical Inference*. Belmont: California.

Comparação Empírica Entre os Métodos de Integração Numérica Simpson e Monte Carlo

Daniella D. Viana[1]

[1]Departamento de Estatística - ICEx - UFMG,
Caixa Postal 702,
30123-970 - Belo Horizonte - MG
E-mail: dvianna@est.ufmg.br

Resumo

Apresentamos a descrição do projeto desenvolvido no curso de Computação Científica e Estatística I, além dos resultados alcançados.

1 Introdução

Para determinar valores numéricos de integrais exatas, os métodos clássicos , como, por exemplo, o de Simpson [1], são preferencialmente utilizados para dimensões baixas. Para o caso multidimensional, entretanto, estes são impraticáveis, cedendo lugar a um outro método muito eficiente neste contexto, o Monte Carlo [2].

Na prática, a regra de Simpson é adequada para funções bem comportadas, ou seja, que são bem representadas por um polinômio. Se $f(x)$ for uma função deste tipo, podemos avaliar a área para um dado número n de subintervalos. Para isso, pode-se utilizar aproximações como a trapezoidal e a interpolação quadrática. Então, dobra-se este número e avalia-se novamente. Se as duas avaliações estiverem suficientemente próximas, podemos parar. Isto dirá se $f(x)$ é bem comportada.

O método de Monte Carlo é um método totalmente diferente dos usuais.

A razão para se optar por um destes dois métodos quando se quer resolver integrais numericamente, é que o erro associado à estimativa do Simpson é aproximadamente $n^{-4/d}$, onde n corresponde ao número de subintervalos que dividem o intervalo de integração d é a dimensão. Ou seja, o erro aumenta proporcionalmente à dimensão. Já para o Monte Carlo, o erro é de $n^{-1/2}$, onde n é o número de pontos aleatórios dentro de uma região, independentemente da dimensão e da natureza do integrando. Isto mostra que cada um dos métodos tem melhor aplicabilidade em casos diferentes.

Este trabalho tem como objetivo implementar um programa que comprove empiricamente qual dos métodos em questão é mais eficiente para determinada situação. Ou seja, que o método de Simpson é melhor para os casos de dimensões baixas, enquanto que para dimensões altas, o método de Monte Carlo é o mais indicado.

Apresentamos a seguir a implementação proposta para o cálculo da integral de uma função normal padronizada, que é relevante na estatística.

2 Implementação

2.1 Regra de Simpson

A seguir, será exibido a classe proposta para o cálculo de uma integral através da 1ª regra de Simpson, com o respectivo programa principal. Este cálculo será feito para a função de densidade de uma normal padronizada. Os limites de integração serão zero e infinito.

2.1.a simpson.hpp

```
// file Simpson.hpp

#include <stdlib.h>
#include <math.h>

inline float Abs(float a)
   {return fabs(a);}
inline double Abs(double a)
   {return fabs(a);}
inline int Abs(int i)
   {return abs(i);}
inline int trunc (float x)
   {return ((int)x);};

class Simpson {
  private:
    float linf;        //limite inferior
    float lsup;        //limite superior
    int intervalo;     //numero de sub-intervalos
    int grau;          //grau do polinomio
  public:
    Simpson();
    ~Simpson() {};
    void Inicializa(void);
    void SetLinf(float li) {linf=li;}
    void SetLsup(float ls) {lsup=ls;}
    void SetIntervalos(int inter) {intervalo=inter;}
    float Calcula(float (*aptFuncao)(float x), int grau);
    float Regra1Simpson(float (*aptFuncao)(float x));
 };

Simpson::Simpson() {
    // Codigos
    d[1] = 2;     d[2] = 6;     d[3] = 8;
    d[4] = 90;    d[5] = 288;   d[6] = 840;
    d[7] = 17280; d[8] = 28350;
    c[1] = 1;     c[2] = 1;     c[3] = 4;
    c[4] = 1;     c[5] = 3;     c[6] = 7;
    c[7] = 32;    c[8] = 12;    c[9] = 19;
    c[10]= 75;    c[11]= 50;    c[12]= 41;
    c[13]= 216;   c[14]= 27;    c[15]= 272;
    c[16]= 751;   c[17]= 3577;  c[18]= 1323;
    c[19]= 2989;  c[20]= 989;   c[21]= 5888;
    c[22]=-928;   c[23]= 10496; c[24] =-4540;
};

void Simpson::Inicializa(void) {
```

```
    float d[9];  float c[25];   float x;
    float y;     float s;       float h;
    int p;       int ck;        int k;
    x = 0; y = 0; s = 0; h = 0;
    p = 0; ck= 0; k = 0;
};

float Simpson::Calcula(float
              (*aptFuncao)(float x), int grau){
    Inicializa();  // inicializa as variaveis
    p = ( grau *( grau + 2 ) + (grau % 2) )/4;
    x = linf;
    y = (*aptFuncao)(linf);
    ck = c[p];
    s = y * ck;
    h = (lsup - linf)/intervalo;
    for (int i=2; i <= (intervalo + 1); i++) {
      x = x + h;
      y = (*aptFuncao)(x);
      k = trunc((((i-2) % grau ) + 1)/grau)-
       trunc((i-1)/intervalo) + 1;
      ck = c[p + trunc(grau/2)-
       Abs(trunc(((i-2)%grau) +1 -(grau/2)))]*k;
      s = s + y * ck;
    };
    return (grau * h/d[grau] * s);
};

float Simpson::Regra1Simpson
            (float (*aptFuncao)(float x)) {
    grau = 2;
    return (Calcula((*aptFuncao),grau));
};

// end file Simpson.hpp
```

2.1.b main.cpp

```
// file main.cpp

#include <stdio.h>
#include <math.h>
#include "Simpson.hpp"

float Funcao(float x) {
  return exp(-(x*x)/2)/sqrt(2*3.1415);
}

int main(void) {
    Simpson Simpson;
    float li = 0.0;
    float ls = 500.0;
```

```
    int inter = 990;
    float  s1;

    Simpson.SetLinf(li);
    Simpson.SetLsup(ls);
    Simpson.SetIntervalos(inter);
    s1 = Simpson.Regra1Simpson(Funcao);
    fprintf(stdout,"Integracao via Simpson 1: %f\n",s1);
    return 0;
};

// end file main.cpp
```

2.2 Monte Carlo

A seguir, será exibido a classe proposta para o cálculo de uma integral através do método de Monte Carlo. Este cálculo será feito para a função de densidade de uma normal padronizada. Os limites de integração serão zero e infinito.

2.2.a montecarlo.hpp

```
// file MonteCarlo.hpp

#include <stdlib.h>
#include <math.h>

class MonteCarlo {
  private:

  public:
    float Entrada[10];
    float SaidaFCalculada[10];
    float SaidaFIntegrada[10];
    float SaidaFAproximada[10];
    loat IntegrDivAprox[10];
    MonteCarlo();
    float Calcula(float (*aptFCalculada)(float x),
                  float (*aptFIntegrada)(float x),
                  float (*aptFAproximada)(float x));
};

MonteCarlo::MonteCarlo() {
};

float MonteCarlo::Calcula(float (*aptFCalculada)(float x),
                  float (*aptFIntegrada)(float x),
                  float (*aptFAproximada)(float x)) {
  float nSomaIntegral;
  int i;

  for (i=0; i<10; i++) {
```

```
   SaidaFCalculada[i] =
     (*aptFCalculada)(Entrada[i]);
  }

  for (i=0; i<10; i++) {
   SaidaFIntegrada[i] =
     (*aptFIntegrada) (SaidaFCalculada[i]);
  }

  for (i=0; i<10; i++) {
   SaidaFAproximada[i] =
     (*aptFAproximada) (SaidaFCalculada[i]);
  }

  for (i=0; i<10; i++) {
   if (SaidaFAproximada[i] != 0) {
     IntegrDivAprox[i] =
       SaidaFIntegrada[i] / SaidaFAproximada[i];
   } else {
      IntegrDivAprox[i] = 0;
     }
  }

  nSomaIntegral = 0;
  for (i=0; i < 10; i++) {
   nSomaIntegral=nSomaIntegral+IntegrDivAprox[i];
  }
  nSomaIntegral = nSomaIntegral / 10;
  return (nSomaIntegral);
};

// end file MonteCarlo.hpp
```

2.2.b main.cpp

```
// file main.cpp

#include <stdio.h>
#include <math.h>
#include "MonteCarlo.hpp"

// Funcao "Calculada"
float fCalculada(float x) {
  return -(log(-x+1))/0.42;
}

// Funcao "Integrada"
float fIntegrada(float x) {
  return exp(-(x*x)/2)/sqrt(2*3.1415);
}

// Funcao "Aproximada"
```

```
float fAproximada(float x) {
  return 0.42*exp(-0.42*x);
}

void InicializaVetorEntrada(float *Entrada) {
// Valores do vetor de entrada
  Entrada[0] = 0.865;   Entrada[1] = 0.159;
  Entrada[2] = 0.079;   Entrada[3] = 0.566;
  Entrada[4] = 0.155;   Entrada[5] = 0.664;
  Entrada[6] = 0.345;   Entrada[7] = 0.655;
  Entrada[8] = 0.812;   Entrada[9] = 0.332;
}

int main(void) {
  MonteCarlo MonteCarlo;
  float ResultadoFinal;
  int i;

  InicializaVetorEntrada(MonteCarlo.Entrada);
  ResultadoFinal =
   MonteCarlo.Calcula(fCalculada,fIntegrada,fAproximada);
  printf("%s%f\n"," RESULTADO FINAL: ",ResultadoFinal);

};

// end file main.cpp
```

3 Conclusões

O desenvolvimento deste trabalho auxiliou na aprendizagem da linguagem C++, além de um maior aprofundamento sobre os métodos de integração Simpson e Monte Carlo. Infelizmente, não foi possível implementar em tempo hábil o cálculo de integrais multidimensionais. Tal fato, comprometeu a comparação final entre os dois métodos. Caso a implementação tivesse sido feita, esperava-se constatar a maior eficiência do método de Monte Carlo em integrais multidimensionais em detrimento ao método de Simpson. Já no cálculo de integrais de baixa dimensão, pôde ser constatado que o método de Simpson é bastante eficiente, sendo até mais preciso que o de Monte Carlo.

Referências

[1] Barroso, L.C.; Barroso, M. M. A.; Campos, F. F.; Carvalho, M. L. B.; Maia, M.L. *Cálculo Numérico (com aplicações).* São Paulo: Editora Harbra, 1987.

[2] Gould, H.; Tobochnik, J. *An Introduction to Computer Simulation Methods- Apllications to Physical Systems.* New York: Addison-Wesley, 1996.

Índice de Autores