Implementação do algoritmo Simultaneous Co-Clustering and Learning utilizando Spark Python API

Fabrício Costa Antoniasse

1 Introdução

Induzir modelos preditivos únicos que caracterizem a base de dados em estudo nem sempre é uma tarefa fácil. Nestes casos, frequentemente, as bases de dados são particionadas em grupos a fim de gerar modelos de predições sobre elas, conhecido na literatura como múltiplos modelos localizados. Neste trabalho será apresentado uma possível implementação, em Apache Spark, do algoritmo SCOAL (Simultaneous Co-Clustering and Learning), o qual realiza de forma iterativa o agrupamento dos dados e o treinamento dos modelos preditivos para cada grupo criado, simultaneamente.

2 SCOAL

Proposto por Deodhar e Ghosh, o algoritmo SCOAL intercala as tarefas de agrupamento e treinamento dos modelos de predição, melhorando assim, tanto a função de agrupar objetos semelhantes entre si como também de gerar modelos preditivos.

Em um contexto de sistema de recomendação, dada uma matriz de avaliação **Z** com dimensão mxn, onde m representa o número total de usuários e n a quantidade de produtos, cada célula $\mathbf{z_{ij}}$ corresponde a avaliação dada pelo usuário **i** ao produto **j**, no qual $\mathbf{z_{ij}} \in \mathbf{R}$. Todo par usuário-produto é descrito por um vetor covariante $\mathbf{x_{ij}}$, composto pela concatenação dos atributos dos usuários $\mathbf{u_i}$, como por exemplo idade, profissão, sexo, com os atributos dos produtos $\mathbf{p_j}$, os quais poderiam ser data de lançamento, gênero do filme, diretor etc. A matriz **Z** é particionada em l clusters de linha e c clusters de coluna, resultando em lxc co-clusters. Para cada co-cluster um modelo preditivo será induzido. Assume-se que o modelo de predição gerado para cada co-cluster seja construido a partir de uma regressão linear, dado por:

$$\mathbf{z}_{\mathrm{ij}} = \beta^T \mathbf{x}_{\mathrm{ij}} + \mathcal{N}(0, \sigma^2) \tag{1}$$

onde $\mathbf{x_{ij}^T} = [1, \mathbf{u_i^T}, \mathbf{p_j^T}]$ e $\beta^T = [\beta_0, \beta_u^T, \beta_p^T]$.

Uma vez estimado β, a partir dos dados históricos representados pela matriz **Z**, é possível fazer predições para qualquer par usuário-produto, através de:

$$\hat{\mathbf{z}}_{\mathrm{ij}} = \beta^T \mathbf{x}_{\mathrm{ij}} \tag{2}$$

A função objetivo a ser minimizada é:

$$\sum \mathbf{w}_{\mathrm{ij}} (\mathbf{z}_{\mathrm{ij}} - \hat{\mathbf{z}}_{\mathrm{ij}})^2 \tag{3}$$

onde $\mathbf{z}_{\mathrm{ij}}$ é o valor observado, $\hat{\mathbf{z}}_{\mathrm{ij}}$ o valor estimado e $\mathbf{w}_{\mathrm{ij}}$ assume o valor 1 caso $\mathbf{z}_{\mathrm{ij}}$ seja conhecido e 0 caso contrário. Neste trabalho, a fim de estimar os valores de β, será utilizada a técnica do gradiente descendente. Os valores de β serão ajustados através da equação 4.

$$\beta_{new} = \beta_{old} + \alpha \sum (\mathbf{z} - \hat{\mathbf{z}}) \mathbf{x} \tag{4}$$

onde α é a taxa de aprendizado, $\mathbf{z}$ é o valor observado, $\hat{\mathbf{z}}$ o valor estimado e **x** é o vetor contendo os atributos.

2.1 Algoritmo SCOAL

O SCOAL recebe uma matriz de avalização **Z** a qual é particionada em l grupos de usuários e c grupos de produtos, onde cada usuário i é mapeado por uma função ρ(i) a um cluster de linha e de forma análoga, cada produto será mapeado por uma função γ(j) a um cluster de coluna. Em seguida é efetuada de forma iterativa, até a convergência, os seguintes passos:

1° passo: ajusta-se os modelos para cada co-cluster, i.e., encontra-se os valores de β_{lc}.
2° passo: efetua-se o agrupamento dos usuários mais similares entre si de forma indireta, ou seja, o agrupamento é realizado através dos valores preditos dos modelos induzidos da seguinte maneira:
➪ deseja-se encontrar, para cada usuário, o cluster de linha l que minimize o erro global dada pela equação 5:

$$\rho(i) = arg_{\mathrm{g}} min \sum_{g=1}^{l} \sum_{j=1}^{n} \mathbf{w}_{\mathrm{ij}}(\mathbf{z}_{\mathrm{ij}} - \hat{\mathbf{z}}_{\mathrm{ij}})^2 \tag{5}$$

onde $\hat{\mathbf{z}}_{\mathrm{ij}} = \beta^T_{g\gamma(j)}\ \mathbf{x}_{\mathrm{ij}}$. Efetua-se o mesmo processo em relação aos produtos, como mostra a equação 6:

$$\gamma(j) = arg_{\mathrm{h}} min \sum_{h=1}^{c} \sum_{i=1}^{m} \mathbf{w}_{\mathrm{ij}}(\mathbf{z}_{\mathrm{ij}} - \hat{\mathbf{z}}_{\mathrm{ij}})^2 \tag{6}$$

onde $\hat{\mathbf{z}}_{\mathrm{ij}} = \beta^T_{\rho(i)h}\ \mathbf{x}_{\mathrm{ij}}$.

O algoritmo SCOAL, resumido, está representado por meio do Algoritmo 1.

Algoritmo 1: Algoritmo SCOAL (adaptado de (DEODHAR; GHOSH))

```
Entrada: Z , x_ij
Saída: Partição dos co-clusters com seus respectivos modelos, β_lc
Inicializar randomicamente os co-clusters.
repita
    Gerar, para cada co-cluster, os modelos preditivos.
    Atualize ρ(i) - declarar cada linha a um cluster de linha que minimize o erro.
    para i=1 até m faça
        ρ(i) = arg_g min Σ_{g=1}^{l} Σ_{j=1}^{n} w_ij (z_ij − ẑ_ij)^2
    fim
    Atualize γ(j) - declarar cada coluna a um cluster de coluna que minimize o erro.
    para j=1 até n faça
        γ(j) = arg_h min Σ_{h=1}^{c} Σ_{i=1}^{m} w_ij (z_ij − ẑ_ij)^2
    fim
até o critério de convergência ser atingido;
retorna (ρ(i), γ(j)) e β_lc
```

2.2 Implementação em Apache Spark

Neste trabalho será apresentada a implementação da geração dos modelos dos co-clusters e o rearranjo dos usuários aos clusters de linha. Vale salientar que o processo de atualização dos produtos aos clusters de coluna é realizado de forma semelhante ao dos ajustes dos clusters de linha. Foi utilizado o Spark Python API (PySpark) para a implementação do algoritmo SCOAL. O RDD "dataCluster" é composto por:

dataCluster.take(1) = [idUser, idItem, rating,[1,$atb^1_{user}, atb^2_{user}, .., atb^p_{user}, atb^1_{item}, atb^2_{item}, .., atb^q_{item}$],(($\rho(i), \gamma(j)$),$\rho(i), \gamma(j)$)]

2.2.1 Geração dos modelos preditivos

A estratégia consiste em dividir a equação 4 em 2 partes:
1 parte - Calcula-se o somatório $\sum(\mathbf{z} - \hat{\mathbf{z}})\mathbf{x}$
2 parte - Adiciona-se o β com o resultado de $\alpha \sum(\mathbf{z} - \hat{\mathbf{z}})\mathbf{x}$ a fim de atualizar os valores de β.

.
def nabla(vet_x, n_beta, y, pesos):
    produto = np.dot(vet_x,pesos[n_beta-1][1])
    return np.dot((y- produto),vet_x)
	A função "nabla" recebe um vetor covariante $\mathbf{x_{ij}}$, o id do co-cluster associado ao vetor $\mathbf{x_{ij}}$ e a matriz de pesos β.
	
	Tem como retorno o cálculo de $(\mathbf{z} - \hat{\mathbf{z}})\mathbf{x}$

. **def** addBeta(n_beta,vet_xB,pesos): **return** np.array(pesos[n_beta-1][1])+vet_xB	A função "addBeta" recebe como parâmetros o id do co-cluster, o produto do resultado da função "nabla" e α e a matriz de pesos β. Tem como retorno o valor dos pesos β atualizados.

```
.
def modelos(pesos,dataCluster,alpha):
    bNovo = dataCluster.map(lambda x : (x[-1][0],nabla(x[3],x[-1][0],x[2],pesos)))
                       .reduceByKey(lambda x,y : np.array(x)+np.array(y))
                       .map(lambda x : (x[0],addBeta(x[0],x[1]*(alpha),pesos)))
                       .sortByKey()
    return bNovo.collect()
```

2.2.2 Atualização de $\rho(i)$ - rearranjo dos usuários aos clusters de linha

```
.
def mse(vet_x, n_beta,y,peso):
    produto = np.dot(vet_x,peso[n_beta-1][1])
    return pow((y - produto),2)
```

A função "mse" recebe um vetor covariante $\mathbf{x_{ij}}$, o id do co-cluster associado ao $\mathbf{x_{ij}}$ e a matriz de pesos β.

Tem como retorno o valor do erro médio quadrático (MSE).

```
.
def mseClustersLinha (user,rating,n_clus,clusterLinha,cl_L,cl_C,peso):
    quantidadeClusterLinha = len(cl_L)
    quantidadeClusterColuna = len(cl_C)
    aux = n_clus - clusterLinha*(quantidadeClusterColuna)
    erros = []
    for linha in range(quantidadeClusterLinha):
        erros.append(mse(user,aux,rating,peso))
        aux += quantidadeClusterColuna
    return erros
```

A função "mseClustersLinha" recebe como parâmetros o id do usuário, $\mathbf{z_{ij}}$, id do co-cluster $(\rho(i), \gamma(j))$, $\rho(i)$, listas contendo os limites dos clusters de linha e de coluna e a matriz de pesos β.
Tem como retorno uma lista contendo os erros globais de cada cluster de linha de forma ordenada.

```
.
def def mudanca(x_erros):
    return x_erros.index(min(x_erros))
```

A função "mudanca" recebe como parâmetro uma lista contendo os erros dos clusters de linha.

Ela retorna a posição do menor valor encontrado na lista.

```
.
mudancaL = dataCluster.map(lambda x : [x[0], mseClustersLinha(x[3],x[2],x[-1][0],x[-1][1],cl_L,cl_C,beta)])
                      .reduceByKey(lambda x ,y : np.array(x)+np.array(y))
                      .mapValues(lambda x : mudanca(list(x)))
dsAuxiliar = dataCluster.map(lambda x : [x[0], x[1:]]).leftOuterJoin(mudancaL)
                      .map(lambda x : [x[0]]+ x[1][0][:-1] +[(x[1][0][-1][0],x[1][-1],x[1][0][-1][2])])
```

Referências

DEODHAR, Meghana; GHOSH, Joydeep. A framework for simultaneous co-clustering and learning from complex data. In: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2007. p. 250-259.

KARAU, Holden et al. Learning spark: lightning-fast big data analysis. " O'Reilly Media, Inc.", 2015.

PEREIRA, Andre Luiz Vizine; HRUSCHKA, Eduardo Raul. Simultaneous co-clustering and learning to address the cold start problem in recommender systems. Knowledge-Based Systems, v. 82, p. 11-19, 2015.