Adoção de Apache Spark na implementação do algoritmo Knn

Clovis Henrique da Silva Chedid

[1]Coppe – Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – RJ – Brazil

cchedid@cos.ufrj.br

Abstract. *This paper presents an implementation of KNN algorithm with landmarks, first presented by [Lima et al. 2018]. This implementations adopt Apache Spark platform as its main technology. This algorithm has a large set of use cases with recomendation systems and the training step involves huge volumes of data. At training stage, it is necessary to do a cartesian product of items and landmarks matrices, king of thing hard to be implemented with Spark. This paper shows how to do that with a technique of data propagation to all nodes of the cluster.*

Resumo. *Este artigo traz uma sugestão de implementação do algoritmo KNN (K-Nearest Neighbors) com Landmarks, apresentado por [Lima et al. 2018], utilizando a plataforma Apache Spark. Este algoritmo é largamente utilizado em sistemas de recomendação e seu treinamento envolve grandes volumes de dados. A execução do produto cartesiano entre a matriz de itens e a de landmarks é o maior desafio da implementação e o artigo traz uma abordagem que utiliza a propagação da matriz de landmarks para todo o cluster Spark.*

1. Introdução

Técnicas de Big Data e Aprendizado de Máquina vêm habilitando um conjunto de novas soluções onde sua principal característica é a análise de dados para predição ou interpretação de situações observadas.

Algoritmos de Aprendizado de Máquina que foram propostos há décadas atrás estão sendo revistos em um novo contexto, o de grande de oferta de dados e poder computacional, tanto de recursos de infra estrutura como de ferramentas e plataformas de processamento distribuído.

Um desses algoritmos é o K Nearest Neighbors (Knn), proposto inicialmente por [Cover and Hart 1967] e depois estendido para acomodar não só classificações mas também regressões por [Stone 1977]. As características desse algoritmo foi pesquisado por [Jingwen Sun 2018] e os seus principais conceitos foram explorados em exemplos didáticos.

Pesquisas vem sendo conduzidas para a adoção do Knn em sistemas de recomendações e [Lima et al. 2018] propôs um novo modelo de predição de avaliação de itens com o emprego de Knn e itens mais expressivos, a saber, *landmarks*. Este artigo foi fruto da pesquisa da extensão de duas outras técnicas, [Braida et al. 2015] e [Lima et al. 2017], onde o emprego de *landmarks* é utilizado para filtragem colaborativa de itens.

O grande desafio computacional da abordagem proposta por [Lima et al. 2018] é a computação da matriz de similaridades entre usuários e *landmarks*, o custo algorítmico é $O(U \times L)$. Com um *dataset* razoável, de um milhão de registros de avaliações e duzentos e cinquenta *landmarks* escolhidos, por exemplo, serão 250×10^9 interações para que se obtenha a matriz de similaridade.

Existem implementações bastante eficientes de multiplicação itens dois a dois ao longo de linha de um conjunto (multiplicação *pairwise*) mas estas exigem a disponibilização de recursos computacionais abundantes. Um exemplo é a implementação do pacote [skl] que consegue computar matrizes com valores de distancia entre vetores com diferentes métricas.

Como dito, estas implementações fazem uso intenso de recursos computacionais e, em geral, não possuem implementação de computação distribuída. O presente artigo traz um estudo de viabilidade da utilização da plataforma [spa] para a implementação do algoritmo Knn com *Landmarks*.

A sessão 2 mostra o algoritmo Knn com *Landmarks* e resumidamente alguns conceitos do [spa] utilizados na implementação.

A sessão 3 descreve como o algoritmo foi implementado utilizando os recursos de processamento distribuído dos [spa].

A sessão 4 apresenta os resultados obtidos com o processamento de um dataset de um milhão de registros obtidos em [mov].

Finalmente, a sessão 5 traz o resumo do experimento e algumas lições que puderam ser aprendidas ao longo da implementação.

2. Algoritmo e conceitos do Apache Spark

O algoritmo Knn com *Landmarks* proposto por [Lima et al. 2018] consiste no emprego de filtragem colaborativa para prever avaliações de itens, dado histórico de avaliações de usuários para outros itens.

Este é um processo de aprendizado supervisionado, onde um grupo de itens ou usuários é pre selecionado, baseado em sua relevância para o conjunto de dados. Este grupo relevante é conhecido como *landmarks* e utilizado como plano de referência para os outros itens.

O fluxo consiste em computar a similaridade entre itens e cada um dos *landmarks*, através do cálculo da distância entre as *features* de um item e de um *landmark*. Esta nova matriz, $Usuarios \times Landmarks$ é então utilizada como matriz de similaridade no algoritmo convencional do Knn.

A listagem abaixo representa o fluxo de processamento do algoritmo Knn com *Landmarks*.

Dados: Matriz Mu de avaliacoes $[IdUsuario, IdFilme, Nota]$, n_landmarks, Lista de Usuarios U, metrica_similaridade

Resultado: Lista M de modelos para cada usuario

Knn com *Landmarks*

```
1
2   # Seleciona os landmarks a partir
3   # de sua relevancia,
4   # quanto mais avaliacoes melhor.
5
6   L = Seleciona os Landmarks(n_landmarks)
7   S = Matriz UxL em branco
8   Para cada u em U faca
9     u_v = avaliacoes por um usuario em Mu
10    Para cada l em L faca
11      l_v = avaliacoes por um landmark em Mu
12      S[u, l] = metrica_similaridade(u_v, l_v)
13    Fim
14  Fim
15
16  Normaliza linhas de S
17  M = Vetor de tamanho U
18  Para cada u em U faca
19    v = landmarks com menores distancias
20    M[u] = v
21  Fim
22  Returna M[u]
```

Tabela 1. Algoritmo1: Pseudo código do algoritmo Knn com *Landmarks*

O primeiro desafio é o cálculo da matriz de similaridade. Esta computação envolve um produto cartesiano. Cada linha da matriz de itens deve ser utilizada no cálculo da distância para cada uma das linhas da matriz de *landmarks*. Além disso, esta é uma operação *row wise*, pois envolve todas as *features* de uma linha ao mesmo tempo.

Implementar o produto cartesiano, necessário para o cálculo da matriz de similaridades é implementar de um algoritmo de força bruta. O Apache Spark não oferece bom desempenho para este tipo de operação. As opções de implementação são laços aninhados ou programação dinâmica. Em qualquer uma das duas alternativas, não é possível utilizar os recursos de processamento distribuído do Apache Spark, pois as matrizes estarão residentes na memória do *driver* do nó principal do *cluster spark*.

Associado a este desafio, está uma característica de otimização do processamento no spark. Quando operações de transformação são executadas sequencialmente, o spark otimiza a execução tentando reduzir a quantidade de troca dados entre os nós do cluster. Um produto cartesiano é um mapeamento das linhas de uma matriz para todas as linhas de outra matriz. A implementação literal desse fluxo impede a otimização da execução, e a execução acontecerá em recortes homogêneos dos dados, em paralelo nos nós do *cluster*.

Para que o spark consiga otimizar o plano de execução do produto cartesiano, será necessário utilizar álgebra relacional ao invés de mapeamento direto das linhas. Uma operação de *cross join*, onde duas tabelas são relacionadas sem cláusulas de relacionamento, o resultado obtido é um produto cartesiano.

Quando o spark processa uma transformação de *cross join*, ele otimiza sua execução da mesma maneira que faria em uma operação de *inner join* ou *outer join*. Esta escolha é mais eficiente que o mapeamento direto dos dados.

O dataset utilizado contém um milhão de avaliações de 4.000 filmes, com 6.000 usuários. A parametrização para escolha de *landmarks* escolhida foi 250 usuários relevantes. Isso gera um produto cartesiano de $6.000 \times 250 = 1.500.000$ interações e um dataset com aproximadamente 6Gb.

Este volume será tratado em uma única submição de processamento do spark. É necessário observar o gerenciamento de memória do cluster, principalmente se ele possuir recursos reduzidos, e adotar técnicas de *chackpoint*, ou seja, a cada operação de manipulação, persistir os dados no sistema distribuído de arquivos para evitar a troca de dados em memória pelos nós. Essa técnica privilegia a troca de dados através do sistema distribuído.

3. Implementação

A implementação do algoritmo Knn com *Landmarks* foi concentrada nos desafios computacionais inerentes à medição de distâncias e projeção de pontos em planos. Ambos os casos envolvem produtos cartesianos.

É importante entender como é a execução de um processamento no Spark. Diferente de um algoritmo em execução em um único nó de cluster, que tem acesso a todos os dados para interações, consultas e transformações, uma instância de algoritmo em execução paralela e distribuída tem acesso aos dados que foram designados para seu nó e só processará os dados disponíveis.

Dessa forma, diversas operações forçam a distribuição dos dados pelos nós do cluster, que aumenta o tempo de processamento. Uma técnica para reduzir esse tipo de fenômeno é, previamente, distribuir os dados entre os nós. Este método pode ser feito utilizando apenas a memória dos nós, o que tem um custo a longo prazo maior, ou utilizando o sistema de arquivos distribuído, que, a curto prazo tem um custo maior.

Para esta implementação, a forma mais adequada é a distribuição dos dados pelo sistema de arquivos, uma vez que o próximo passo será projetar o produto cartesiano dos dados.

Para a projeção do cartesiano, existem várias técnicas, como laços aninhados ou mapeamento linha-linha dos dados. Em ambos casos, o processamento distribuído do spark não é utilizado, já que o algoritmo já determina o fluxo de execução.

Considerando que o cartesiano é formado de duas partes, uma delas pode ser distribuída previamente pelos nós do cluster. Com isso, o cartesiano pode ser obtido através de uma operação de *join* sim cláusulas de ligação, ou seja, o *join* vai relacionar todos os dados entre si. Essa técnica pode ser observada na listagem abaixo.

Knn com *Landmarks*

```
1  def save_cross_join(X, Y, base_hdfs, name):
2   """Cria um dataframe com o produto
3     cartesiano de duas matrizes.
4
5   :param DataFrame X: Matriz 1.
6   :param DataFrame Y: Matriz 2.
7   :param str base_hdfs: Caminho base para
8     salvar o novo dataframe.
9   :param str name: Nome do arquivo parquet.
10  :return:
11  :rtype: None
12
13  """
14  X_columns = ['x_{}'.format(c) for c in X.columns]
15  Y_columns = ['y_{}'.format(c) for c in Y.columns]
16  X_ren_ = X.toDF(*X_columns)
17  X_ren = X_ren_.repartition(12)
18  Y_ren_ = Y.toDF(*Y_columns)
19  Y_ren = Y_ren_.repartition(12).cache()
20  Y_ren.take(1)
21  spark.sql('set spark.sql.autoBroadcastJoinThreshold=0')
22
23  cross = X_ren.join(Y_ren)
24  cross.repartition(12).write \
25    .mode('append') \
26    .parquet('{}{}.parquet'.format(base_hdfs, name))
```

Tabela 2. Algoritmo2: Implementação do cartesiano em PySpark

Como é possível observar, a linha 19 executa a transformação de *cache* em um dos datasets. Esta transformação é efetivada na linha 20, quando a ação *take* é executada. O *chache* força a distribuição dos dados através do sistema distribuído de arquivos dos nós do cluster.

A linha 23 executa a transformação de *join* entre o dataset principal e o "cacheado", gerando o produto cartesiano entre os dos datasets. Para aliviar a pressão de memória da operação de *join*, o dataset resultante do cartesiano é salvo no sistema distribuído de arquivos, como é executado nas linhas $[24, 25, 26]$.

Com o cartesiano calculado, a matriz resultante será

$[UsuarioX, UsuarioY, ItemsdeX[], Itemsde : Y[]]$. Cada linha dessa matriz representa as avaliações de um usuário e as avaliações de um dos *landmarks*. É necessário iterar por todas as linhas dessa matriz, calculando as distâncias entre o usuário e o *landmark* em questão.

Definindo-se uma função de cálculo de distância, uma simples projeção na matriz do cartesiano percorrerá todas as linhas calculando as distâncias. Esta operação será otimizada pelo Spark e distribuída pelos nós do cluster. A próxima listagem mostra como a função de distância e a projeção foram implementadas.

Knn com *Landmarks*

```
1   def get_distance(X_features, Y_features, metric):
2     """Calcula a distancia entre duas listas com a metrica fornecida.
3
4     :param list X_features: Lista 1.
5     :param list Y_features: Lista 2.
6     :param func metric: Funcao de calculo da distancia.
7     :return: Valor da distancia.
8     :rtype: DoubleType ou None
9
10    """
11    if (X_features is not None) \
12     and (Y_features is not None) \
13     and (metric is not None):
14       distance = metric(X_features, Y_features)
15    return distance.item()
16  metric = dis.correlation
17  partial_get_distance = partial(get_distance, metric=metric)
18  get_distance_udf = f.udf(partial_get_distance, t.DoubleType())
19
20  similarities = cross2.select(\
21   f.col('x_user').alias('user'),\
22   f.col('y_user').alias('y_label'),\
23   get_distance_udf('x_features',\
24    'y_features').alias('distance'))
```

Tabela 3. Algoritmo2: Implementação do cartesiano em PySpark

As linhas $[20-24]$ executam a projeção que calcula a distância de um usuário para um dos *landmarks*. Ao final dessa operação, a matriz resultante é Usuário X Landmark. Isso representa todos os usuários projetados no espaço de *landmarks*. Além da projeção, é necessário também normalizar todas as distâncias, para que elas sejam distâncias relativas. Este processo pode ser obtido com um mapeamento das linhas da matriz resultante, aplicando operações *row wide*.

Para completar o algoritmo Knn, ainda é necessário projetar os usuários contra o próprio plano, obtendo uma matriz Usuário X Usuário. O processo descrito acima, da aplicação do *cross join* e projeção para obtenção das distâncias é novamente executado, agora utilizando somente a matriz Usuário X Landmarks como parâmetro. O resultado será a matriz Usuário X Usuário.

Finalmente, a matriz Usuário X Usuário é percorrida e para cada usuário (linha), são escolhidos os usuários (coluna) mais próximos, ou seja, com menor distância. São se-

lecionados N vizinhos mais próximos, o que será utilizado nas estimativas de avaliações de um filme por um usuário. A listagem abaixo demonstra como os vizinhos são escolhidos.

Knn com *Landmarks*

```
1  def get_neighbors(row, k_nn):
2   """Para cada linha, busca os vizinhos mais proximos.
3
4   :param Row row: Uma linha de usuario.
5   :param int k_nn: Quantidade de vizinhos.
6   :param double lim_similarity: Limite de similaridade.
7   :return: Linha [item, num_vizinhos, vizinhos[]]
8   :rtype: Row
9
10  """
11
12  d = row.asDict()
13  user = d['user']
14  other_users = [d[k] for d.keys() if k != 'user']
15  # other_users = [(i, other_users[i]) for i in np.arange(len(
        other_users))]
16  neighbors = sorted(other_users, reverse = True)
17  neighbors = neighbors[1:k_nn+1]
18
19  new_row = {}
20  new_row['user'] = user
21  new_row['count'] = len(neighbors)
22  new_row['neighbors'] = neighbors
23  return Row(**new_row)
24
25 partial_get_neighbors = partial(get_neighbors, k_nn = n_knn)
26 neighbors = similarities_u_u.rdd.map(partial_get_neighbors).toDF()
```

Tabela 4. Algoritmo2: Implementação do cartesiano em PySpark

Após todo o processamento dos dados e escolha dos vizinhos próximos projetados no plano de *landmarks*, é necessário materializar todos os dados e apresentar aos nós do cluster, pois os próximos passos do Knn envolvem buscas de dados constantes.

Um aspecto relevante, que permeia toda a implementação do algoritmo é a distribuição dos dados, técnica de particionamento, que induz a distribuição do trabalho nos nós do cluster.

Em vários trechos de código existe a instrução $.repartition(12)$. Esta instrução faz com que os dados do dataframe em questão sejam distribuídos em 12 partições antes da próxima transformação. Assim, a próxima transformação será distribuída pelos executores do cluster em até 12 vezes.

A quantidade 12 vem da quantidade de executores existentes do cluster e também da avaliação do tamanho do dataset resultante após o particionamento. Um dataset pequeno resultante ocasionará em muitas operações de redistribuição de dados, enquanto poucas partições resultará em executores sem trabalho.

É necessário balancear o número de partições e o tamanho do dataset resultante

para se obter o máximo consumo de recursos computacionais. Como o volume de dados é grante, na maioria das vezes, o particionamento é feito diretamente para o número de executores, pois os dataset resultantes terão ainda tamanho suficiente para o trabalho de um executor.

Em casos específicos, o dataset original é menor e o particionamento escolhido privilegiou o tamanho do dataset resultante, como é na situação em que a matriz de simularidade Usuário X Usuário que é uma matriz de forma aproximada de 6000 X 6000 itens, foi particionada em apenas 4 partes, pois o tamanho de cada dataset resultante, para um particionamento maior, era muito pequeno (200Kb).

4. Resultados

As técnicas de força bruta, como laços aninhados ou mapeamento direto das linhas de uma matriz podem ser implmentados na forma de comandos de algebra relacional, sem perda funcional. O Spark atende corretamente ao planejamento da execução do volume de operações independentes envolvidas em um produto cartesiano e distribui corretamente a carga de processamento entre os nós.

Para este ensaio, foi utilizado o dataset de 1 milhão de avaliações, que pode ser obtido em [mov], a técnica de Knn com $k = 250$ e a métrica de distância Pearson (correlação).

Além disso, foi utilizado um cluster Spark, hospedado no serviço Microsoft Azure [azu] HDInsights [hdi]. O cluster consiste em:

- Nó Master
  Quantidade:2
  CPU: 4 cores/nó
  Memóeria: 28Gb/nó
- Nó Worker
  Quantidade:6
  CPU: 8 cores/nó
  Memóeria: 56Gb/nó

Como *framework* de execução foi utilizado o Apache Spark na versão 2.4.0 e o Zeppelin Notebook para desenvolvimento e execução dos *scripts* de processamento.

Em geral, o processamento não se mostrou com alto desempenho, mas o objetivo principal do trabalho era apenas mostrar as técnicas de processamento que viabilizassem a implementação do Knn com *Landmarks*. Os tempos de resposta de um cartesiano com 1.5M de iterações foram, em média, de 15 minutos. Este tempo é muito maior que o obtido com o framework SKLearn do Python.

Por outro lado, a execução foi robusta e sem esgotamento de recursos de processamento, mantendo-se com um consumo de recursos gerenciável. Quando compara-se com o SKLearn, o último busca a alocação completa dos recursos disponível, assumindo uma execução irresponsável.

Ao se escalar o volume de dados, o Spark mantem o processamento constante mas o SKLearn sofre com as limitações de memória, principalmente.

O algoritmo Knn se resume a apresentar os vizinhos mais próximos de um determinado item. A aplicação dessa informação é a predição de novas avaliações de um item.

Esta aplicação envolve, no mínimo, 6 consultas independentes nos dados de treino e também nos dados dos vizinhos mais próximos. O processamento Spark, baseado no sistema distribuído de arquivos, não apresenta bom tempo de resposta para essa situação. Para este tipo de processamento é interessante materializar os dados em memória e apresentar diretamente a cada nó do cluster, para que possam ser acessados rapidamente.

O principal resultado com este trabalho foi a referência para um algorítimo de produto cartesiano para o cálculo da matriz de similaridade do algoritmo Knn. Como o espaço landmark precisa ser usado como plano de projeção dos items da matriz de dados, este produto cartesiano é grande e o algoritmo se foca na distribuição do processamento entre os nós.

A implementação da operação *row wise* para identificação dos vizinhos próximos também se mostra bastante eficiente com o Spark. O uso de RDDs e mapeamento linha a linha garantiu paralelismo de processamento e acesso único a todos os dados de um usuário.

Estas duas técnicas são robustas e com consumo de recursos previsível (não exponencial). Quando projetado o volume de dados e o tamanho do cluster, as técnicas se mostram sustentáveis e eficientes pois mantém o consumo de recursos em patamares constantes ao longo do tempo.

O último resultado, pouco mencionado na sessão de Implementação, está relacionado a como a redistribuição dos dados, técnica conhecida como particionamento, pode garantir o consumo eficiente dos recursos disponíveis em operações longas de transformação.

Em vários trechos do código do trabalho, houve o reparticionamento do dataset em 12 itens. Esta quantidade representa o número de executores no cluster. São 48 núcleos disponíveis no cluster. Assumindo-se 3 núcleos por executor, o cluster pode acomodar 12 executores, ainda reservando 2 núcleos em cada nó para processamento de sistema operacional e filas.

Com essa técnica, cada novo processamento, considera 12 partições de dados a serem processadas e distribui corretamente os dados nos 12 executores existentes.

5. Conclusão

Após analisar o algoritmo Knn com *Landmarks* e identificar alguns produtos cartesianos necessários, o desafio do trabalho foi buscar uma implementação desses produtos cartesianos que fizesse bom uso dos recursos de processamento distribuído e do cluster Spark.

Atualmente, há uma implmentação experimental de transformação *row wise* no framework Spark, mas não é possível ainda definir a métrica aplicada ou as matrizes da operação. Esta implementação ainda é muito restritiva.

A situação com o algoritmo Knn não é diferente. Não existe uma implementação desse algoritmo no framework Spark e os comentários encontrados em fóruns técnicos é que o Knn, pela necessidade do produto cartesiano, é muito agressivo em consumo de

recursos para sem implementado com Spark ou qualquer outro processamento distribuído.

Neste trabalho a abordagem foi garantir que o produto cartesiano possa ser executado, penalizando o desempenho mas garantindo consumo de recursos eficiente. A maior caracteristica do processamento com Spark é garantir sua execução e não o melhor desempenho, simplesmente. É o melhor desempenho, desde que a execução esteja assegurada.

Esta filosofia pode ser aplicada ao produto cartesiano e fará sentido quando os volumes de dados que participam do cartesiano aumentam e tornam a computação exponencialmente grande. Para isso, é necessário trabalhar, não no algoritmo do cartesiano, mas como esse será executado.

A técnica utilizada foi transformar o cartesiano em um *Cross Join*, deixando previsível todo o trabalho a ser realizado e distribuindo esse trabalho através do sistema distribuído de arquivos, evitando ao máximo redistribuição de dados em memória, o que é mais custoso e impede o fluxo de execução.

A técnica se mostrou adequada e foi possível realizar os dois cartesianos propostos no algoritmo Knn com *Landmarks* em um cluster mínimo - sem considerar questões de otimização de desempenho, apenas de robustez.

Todo o código deste trabalho é de domínio público e pode ser encontrado no repositório GitHub do autor [coi].

Referências

Apache spark. `https://spark.apache.org/`. Accessed: 2019-12-13.

Azure cloud services. `https://azure.com`. Accessed: 2019-12-13.

Hdinsights azure service. `https://docs.microsoft.com/en-us/azure/hdinsight/`. Accessed: 2019-12-13.

Knn landmarks spark implementation. `https://github.com/coichedid/knn_landmarks_spark`. Accessed: 2019-12-13.

Movielens 1m dataset. `https://grouplens.org/datasets/movielens/1m/`. Accessed: 2019-12-13.

sklearn.metrics.pairwise distances. `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise_distances.html`. Accessed: 2019-12-13.

Braida, F., Mello, C. E., Pasinato, M. B., and Zimbrão, G. (2015). Transforming collaborative filtering into supervised learning. *Expert Systems with Applications*, 42(10):4733 – 4742.

Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27.

Jingwen Sun, Weixing Du, N. S. (2018). A survey of knn algorithm. *Information Engineering and Applied Computing*.

Lima, G., Mello, C., and Silva, G. (2017). Speeding up memory-based collaborative filtering with landmarks.

Lima, G., Mello, C., and Silva, G. (2018). A new modeling for item ratings using landmarks. pages 1–8.

Stone, C. J. (1977). Consistent nonparametric regression. *Ann. Statist.*, 5(4):595–620.