ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO PROJECT 1

RECOMMENDER SYSTEM WITH SPARKS

- Sinh viên thực hiện:
  - NGUYỄN HOÀNG LINH MSSV: 1712559
  - HUỲNH NGỌC QUÂN MSSV: 1712689
  - LÊ THANH HIẾU MSSV: 1712434

- Bộ môn: Ứng dụng Dữ Liệu Lớn
- Lớp: 17_21

TP. Hồ Chí Minh, ngày 02 tháng 05 năm 2021

1. Thông tin nhóm

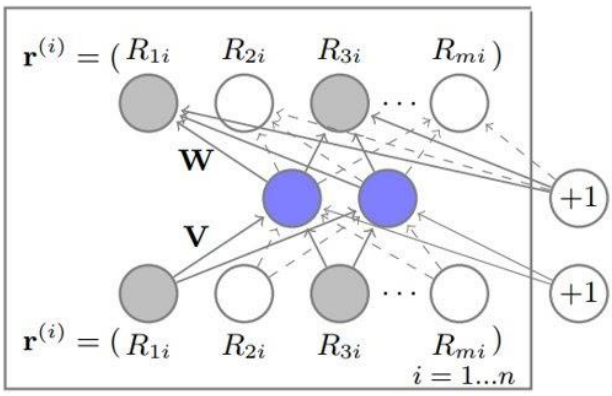
STT	MSSV	Họ tên	Công việc
1	1712434	Lê Thanh Hiếu	• Tìm hiểu bài báo • Trình bày bài báo • Cài đặt SparseFC trên Colab • Cài đặt ALS trên Spark • Viết báo cáo
2	1712559	Nguyễn Hoàng Linh	• Tìm hiểu bài báo • Trình bày chính • Cài đặt SparseFC trên Colab • Cài đặt thuật toán i-AutoRec • Viết báo cáo
3	1712689	Huỳnh Ngọc Quân	• Tìm hiểu bài báo • Viết báo cáo • Đánh giá performance

2. Nội dung

- **Bài báo mà nhóm nghiên cứu:** Lorenz K. Muller, Julien N.P. Martel, Giacomo Indiveri. Kernelized Synaptic Weight Matrices

a. Tổng quan về Kernelized Synaptic Weight Matrices (hay gọi là Sparse FC)

- Nhóm tác giả đã giới thiệu một kiến trúc neural network mới mà trong đó ma trận trọng số được tham số hóa lại dưới dạng các vector low-dimensional thông qua 1 hàm kernel. Kết quả đạt được hiệu năng state-of-the-art trong 1 tác vụ collaborative filtering trên tập dataset MovieLens
- Tổng quan về tập dataset MovieLens 1M
- Gồm: 1 triệu đánh giá của 3706 bộ phim bởi 16000 người dùng, với mật độ (density) dữ liệu là 0.045 trong năm 2000.
- **Tổng quan về thuật toán:**



- Ban đầu, danh sách đánh giá của mỗi người dùng đối với (n) vật phẩm sẽ là đầu vào của mô hình Sparse FC.
- Sau đó, dữ liệu đầu vào sẽ đi qua 2 lớp V và W, với V và W đã được thay đổi tham số hoá để có thể giảm độ phức tạp tính toán cho mô hình.
- Cuối cùng, mô hình sẽ cho ra kết quả h(r(i); delta) là điểm số đánh giá của người dùng đối với toàn bộ vật phẩm đã đưa vào ban đầu.
- Hàm mất mát của thuật toán:

$$L = \min_{\theta} \sum_{i=1}^{n} \left\| r^{(i)} - h(r^{(i)}; \theta) \right\|_2^2 + R$$

$$R = \lambda_2 \left(\sum_{ij} \alpha_{ij} + \sum_{ij} \beta_{ij} \right)$$

$$+\lambda_0 \left(\sum_{ij} K(\vec{v}_i, \vec{u}_j) + \sum_{ij} K(\vec{s}_i, \vec{t}_j) \right)$$

- **Ưu điểm của thuật toán:**
- Mô hình đơn giản và ít tham số, vì tác giả đã sử dụng thủ thuật thay đổi tham số hoá của ma trận W, V và mô hình chỉ có 2 lớp ẩn.
- Độ chính xác cao trên tập Movielens nói chung và xếp đầu tập Movielens-1M
- **Nhược điểm thuật toán:**
- Mặc dù đã giải quyết việc giảm thiểu tham số cho thuật toán, nhưng vẫn chưa thể triển khai tốt trên tập dữ liệu lớn hơn (ví dụ Movielens 20M, Netflix...) do độ chính xác và hiệu suất thấp hơn

b. Cài đặt thuật toán đã chọn trên Google Colab và môi trường Spark

- Do gặp khó khăn trong quá trình cài đặt lại thuật toán trên môi trường Spark nên nhóm đã cài đặt lại source code của tác giả trên Google Colab, Tensorflow 1.x
- Đầu tiên, nhóm tiến hành load tập dữ liệu ml-1m và lấy ra ngẫu nhiên 10% làm tập validation. Kết quả trả về gồm 2 ma trận trainRatings, validRatings có kích thước (n_u, n_m) trong đó n_u là số users và n_m là số movies

- **Xây dựng các hàm network:**

```python
# define network functions
def kernel(u, v):
    """
    Sparsifying kernel function

    :param u: input vectors [n_in, 1, n_dim]
    :param v: output vectors [1, n_hid, n_dim]
    :return: input to output connection matrix
    """
    dist = tf.norm(u - v, ord=2, axis=2)
    hat = tf.maximum(0., 1. - dist**2)
    return hat
```

```python
def kernel_layer(x, n_hid, n_dim, activation, lambda_s,
                 lambda_2, name):
    """
    a kernel sparsified layer

    :param x: input [batch, channels]
    :param n_hid: number of hidden units
    :param n_dim: number of dimensions to embed for kernelization
    :param activation: output activation
    :param name: layer name for scoping
    :return: layer output, regularization term
    """

    # define variables
    with tf.variable_scope(name):
        W = tf.get_variable('W', [x.shape[1], n_hid])
        n_in = x.get_shape().as_list()[1]
        u = tf.get_variable('u', initializer=tf.random.truncated_normal([n_in, 1, n_dim], 0., 1e-3))
        v = tf.get_variable('v', initializer=tf.random.truncated_normal([1, n_hid, n_dim], 0., 1e-3))
        b = tf.get_variable('b', [n_hid])

    # compute sparsifying kernel
    # as u and v move further from each other for some given pair of neurons, their connection
    # decreases in strength and eventually goes to zero.
    w_hat = kernel(u, v)

    # compute regularization terms
    sparse_reg = tf.contrib.layers.l2_regularizer(lambda_s)
    sparse_reg_term = tf.contrib.layers.apply_regularization(sparse_reg, [w_hat])

    l2_reg = tf.contrib.layers.l2_regularizer(lambda_2)
    l2_reg_term = tf.contrib.layers.apply_regularization(l2_reg, [W])

    # compute output
    W_eff = W * w_hat
    y = tf.matmul(x, W_eff) + b
    y = activation(y)
    return y, sparse_reg_term + l2_reg_term
```

- **Chỉnh các hyper-parameters**

```
# Set hyper-parameters
n_hid = 500
lambda_2 = 60.0
lambda_s = 0.013
n_layers = 2
output_every = 50  # evaluate performance on test set; breaks l-bfgs loop
n_epoch = n_layers * 10 * output_every
verbose_bfgs = True
use_gpu = True
if not use_gpu:
    os.environ['CUDA_VISIBLE_DEVICES'] = ''
```

- **Khởi tạo network**

```
# Instantiate network
y = R
reg_losses = None
for i in range(n_layers):
    y, reg_loss = kernel_layer(x=y, n_hid=n_hid, n_dim=5, activation=tf.nn.sigmoid, lambda_s=lambda_s,
                               lambda_2=lambda_2, name=str(i))
    reg_losses = reg_loss if reg_losses is None else reg_losses + reg_loss
prediction, reg_loss = kernel_layer(x=y, n_hid=n_u, n_dim=5, activation=tf.identity, lambda_s=lambda_s,
                               lambda_2=lambda_2, name='out')
reg_losses = reg_losses + reg_loss
```

- **Tính độ lỗi**

```
# Compute loss (symbolic)
diff = tm*(R - prediction)
sqE = tf.nn.l2_loss(diff)
loss = sqE + reg_losses
```

- **Khởi tạo thuật toán L-BFGS Optimizer**

```
# Instantiate L-BFGS Optimizer
optimizer = tf.contrib.opt.ScipyOptimizerInterface(loss, options={'maxiter': output_every,
                                                                   'disp': verbose_bfgs,
                                                                   'maxcor': 10},
                                                   method='L-BFGS-B')
```

- **Khởi tạo tensorflow session, training model và tiến hành validation**

```python
# Training and validation loop
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    for i in range(int(n_epoch / output_every)):
        optimizer.minimize(sess, feed_dict={R: tr}) #do maxiter optimization steps
        pre = sess.run(prediction, feed_dict={R: tr}) #predict ratings

        error = (vm * (np.clip(pre, 1., 5.) - vr) ** 2).sum() / vm.sum() #compute validation error
        error_train = (tm * (np.clip(pre, 1., 5.) - tr) ** 2).sum() / tm.sum() #compute train error

        print('.-^-._' * 12)
        print('epoch:', i, 'validation rmse:', np.sqrt(error), 'train rmse:', np.sqrt(error_train))
        print('.-^-._' * 12)

    with open(summary_file, 'a') as file:
        for a in sys.argv[1:]:
            file.write(a + ' ')
        file.write(str(np.sqrt(error)) + ' ' + str(np.sqrt(error_train))
                   + ' ' + str(seed) + '\n')
        file.close()
```

c. Benchmark thuật toán đã cài đặt trên tập đánh giá MovieLens 1M

Kết quả: với các thông số mặc định, thuật toán cho kết quả RMSE = 0.82712615 ở epoch thứ 20

```
INFO:tensorflow:Optimization terminated with:
  Message: b'STOP: TOTAL NO. of ITERATIONS REACHED LIMIT'
  Objective function value: 310487.500000
  Number of iterations: 50
  Number of functions evaluations: 53
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
epoch: 17 validation rmse: 0.82686025 train rmse: 0.6854515
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
INFO:tensorflow:Optimization terminated with:
  Message: b'STOP: TOTAL NO. of ITERATIONS REACHED LIMIT'
  Objective function value: 309490.468750
  Number of iterations: 50
  Number of functions evaluations: 55
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
epoch: 18 validation rmse: 0.8267706 train rmse: 0.6840771
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
INFO:tensorflow:Optimization terminated with:
  Message: b'STOP: TOTAL NO. of ITERATIONS REACHED LIMIT'
  Objective function value: 308503.062500
  Number of iterations: 50
  Number of functions evaluations: 54
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
epoch: 19 validation rmse: 0.82712615 train rmse: 0.6825071
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
```

d. Cài đặt thuật toán ALS, thuật toán I-AutoRec, và so sánh kết quả

Cài đặt thuật toán ALS trên PySpark

- **Nhóm đọc dữ liệu từ file .dat bằng các đọc ra RDD và chuyển đổi sang spark data frame để có thể fit được model ALS**

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DoubleType, Row

def dataLoaderALS(path='./ml-1m/ratings.dat', delimiter='::',
                  seed=seed, valfrac=0.1):
  schema = StructType([
        StructField("userId",IntegerType(),True),
        StructField("movieId",IntegerType(),True),
        StructField("rating",DoubleType(),True),
        StructField("timestamp", IntegerType(), True),
        ])
  rdd = spark.sparkContext.textFile(path)\
        .map(lambda x: x.split('::'))\
        .map(lambda x: [int(x[0]), int(x[1]), float(x[2]), int(x[3])])
  data = spark.createDataFrame(data=rdd, schema=schema)
  return data

```

- **Sau đó chia dữ liêu đọc được thành 2 tập train, test theo tỉ lệ 90:10 tương tự như với SparseFC**

```
# split data into train and test sets with 90:10 proportions
train, test = data.randomSplit([0.9, 0.1], seed=seed)
# cache to reduce time taken
train.cache()
```

- **Khởi tạo model ALS**

```
# import the ALS algorithm we will be using
from pyspark.ml.recommendation import ALS

#instantiate model with the "drop" cold start strategy
model = ALS(coldStartStrategy="drop")
```

- **Fitting model**

```
# import the ALS algorithm we will be using
from pyspark.ml.recommendation import ALS

#instantiate model with the "drop" cold start strategy
model = ALS(coldStartStrategy="drop")
```

```
# set the column names for the required data
model.setItemCol("movieId")\
    .setUserCol("userId")\
    .setRatingCol("rating")
```

```
ALS_687af8489039
```

```
model = model.fit(train)
```

- Dự đoán rating bằng cách sử dụng model để dự đoán trên tập test predictions = model.transform(test)
- **Đánh giá kết quả của model**

```
# import the regression evaluator
from pyspark.ml.evaluation import RegressionEvaluator

# instantiate evaluator, specifying the desired metric "mae" and the columns
# that contain the predictions and the actual values
evaluator = RegressionEvaluator(metricName="rmse", predictionCol="prediction", labelCol="rating")
```

```
# evaluate the output of our model
rmse = evaluator.evaluate(predictions)
print('The ALS RMSE is ' + str(rmse))
```

```
The ALS RMSE is 0.8665453923948111
```

Cài đặt thuật toán I-AutoRec

- **Ban đầu, mô hình sẽ tạo ra tham số delta (W, V, mu, b) để chuẩn bị cho việc huấn luyện, sau đó sẽ tính toán hàm mất mát cho mô hình.**

```
def prepare_model(self):
    self.input_R = tf.placeholder(dtype=tf.float32, shape=[None, self.num_items], name="input_R")
    self.input_mask_R = tf.placeholder(dtype=tf.float32, shape=[None, self.num_items], name="input_mask_R")

    V = tf.get_variable(name="V", initializer=tf.truncated_normal(shape=[self.num_items, self.hidden_neuron],
                                    mean=0, stddev=0.03),dtype=tf.float32)
    W = tf.get_variable(name="W", initializer=tf.truncated_normal(shape=[self.hidden_neuron, self.num_items],
                                    mean=0, stddev=0.03),dtype=tf.float32)
    mu = tf.get_variable(name="mu", initializer=tf.zeros(shape=self.hidden_neuron),dtype=tf.float32)
    b = tf.get_variable(name="b", initializer=tf.zeros(shape=self.num_items), dtype=tf.float32)

    pre_Encoder = tf.matmul(self.input_R,V) + mu
    self.Encoder = tf.nn.sigmoid(pre_Encoder)
    pre_Decoder = tf.matmul(self.Encoder,W) + b
    self.Decoder = tf.identity(pre_Decoder)

    pre_rec_cost = tf.multiply((self.input_R - self.Decoder) , self.input_mask_R)
    rec_cost = tf.square(self.l2_norm(pre_rec_cost))
    pre_reg_cost = tf.square(self.l2_norm(W)) + tf.square(self.l2_norm(V))
    reg_cost = self.lambda_value * 0.5 * pre_reg_cost

    self.cost = rec_cost + reg_cost
```

- **Tiếp theo, ta sẽ sử dụng hàm tối ưu Adam để tìm ra tham số phù hợp với mô hình**

```
if self.optimizer_method == "Adam":
    optimizer = tf.train.AdamOptimizer(self.lr)
elif self.optimizer_method == "RMSProp":
    optimizer = tf.train.RMSPropOptimizer(self.lr)
else:
    raise ValueError("Optimizer Key ERROR")

if self.grad_clip:
    gvs = optimizer.compute_gradients(self.cost)
    capped_gvs = [(tf.clip_by_value(grad, -5., 5.), var) for grad, var in gvs]
    self.optimizer = optimizer.apply_gradients(capped_gvs, global_step=self.global_step)
else:
    self.optimizer = optimizer.minimize(self.cost, global_step=self.global_step)
```

- **Kết quả:**

```
================================================================================================
Training // Epoch 1990 //  Total cost = 207077.54 Elapsed time : 0 sec
Testing // Epoch 1990 //  Total cost = 74444.45  RMSE = 0.84732 Elapsed time : 0 sec
================================================================================================
Training // Epoch 1991 //  Total cost = 207062.07 Elapsed time : 0 sec
Testing // Epoch 1991 //  Total cost = 74685.96  RMSE = 0.84875 Elapsed time : 0 sec
================================================================================================
Training // Epoch 1992 //  Total cost = 207206.84 Elapsed time : 0 sec
Testing // Epoch 1992 //  Total cost = 74755.32  RMSE = 0.84915 Elapsed time : 0 sec
================================================================================================
Training // Epoch 1993 //  Total cost = 207075.65 Elapsed time : 0 sec
Testing // Epoch 1993 //  Total cost = 74731.09  RMSE = 0.84901 Elapsed time : 0 sec
================================================================================================
Training // Epoch 1994 //  Total cost = 207093.38 Elapsed time : 0 sec
Testing // Epoch 1994 //  Total cost = 74919.70  RMSE = 0.85012 Elapsed time : 0 sec
================================================================================================
Training // Epoch 1995 //  Total cost = 207204.83 Elapsed time : 0 sec
Testing // Epoch 1995 //  Total cost = 74708.45  RMSE = 0.84887 Elapsed time : 0 sec
================================================================================================
Training // Epoch 1996 //  Total cost = 207168.13 Elapsed time : 0 sec
Testing // Epoch 1996 //  Total cost = 74535.48  RMSE = 0.84786 Elapsed time : 0 sec
================================================================================================
Training // Epoch 1997 //  Total cost = 207124.73 Elapsed time : 0 sec
Testing // Epoch 1997 //  Total cost = 74762.47  RMSE = 0.84920 Elapsed time : 0 sec
================================================================================================
Training // Epoch 1998 //  Total cost = 207140.86 Elapsed time : 0 sec
Testing // Epoch 1998 //  Total cost = 74648.93  RMSE = 0.84853 Elapsed time : 0 sec
================================================================================================
Training // Epoch 1999 //  Total cost = 207128.38 Elapsed time : 0 sec
Testing // Epoch 1999 //  Total cost = 74700.43  RMSE = 0.84884 Elapsed time : 0 sec
================================================================================================
```

So sánh hiệu năng giữa ALS và I-AutoRec

- Ta có I-AutoRec với RMSE = 0.849 tốt hơn ALS với RMSE = 0.867

e. **Thảo luận về hiệu năng của SparseFC, ALS by Spark và I-AutoRec trên tập dataset MovieLens 10M**
    - Chọn dataset movielens 10M: 10 triệu ratings, 10,680 movies, 71,000 users, density = 0.013
    - Do giới hạn của RAM nên không thể chạy các thuật toán trên với tập dataset này, nhóm thu thập kết quả RMSE từ các bài báo để so sánh hiệu năng của các thuật toán
    - Ngoài ra Spark ALS recommender là một thuật toán matrix factorization sử dụng Alternating Least Squares với Weighted-Lambda-Regularization để tối ưu (ALS-WR) nên nhóm sẽ lấy kết quả của thuật toán ALS-WR làm kết quả cho thuật toán baseline được cung cấp bởi Spark

Model	RMSE	Parameters	MACS
I-AutoRec	0.782	6.30M	3.28M
ALS-WR	0.7949	-	-
Sparse FC	0.769	7.00M	2.23M

Nhận xét: Ta có thể thấy SparseFC cho kết quả RSME vượt trội hơn so với ALS-WR và I-AutoRec trên tập dataset có cả kích thước lớn hơn cũng như mật độ dữ liệu thấp hơn movielens 1M và độ phức tạp của thuật toán Sparse FC thấp hơn so với I-Autorec nên cho hiệu suất tốt hơn.

3. Tài Liệu Tham khảo

- Cài đặt PySpark trên colab: file notebook seminar PySpark Dataframe
- Source code SparseFC: https://github.com/lorenzMuller/kernelNet_MovieLens
- Cài đặt thuật toán ALS: Spark Collaborative Filtering document
- Source code I-AutoRec: https://github.com/gtshs2/Autorec
- Kết quả RSME của các thuật toán: Steffen Rendle, Li Zhang, Yehuda Koren. On the Difficulty of Evaluating Baselines