COMP 4651 PROJECT ---- *Recommendation System* with Cloud Computing Implement

ZHANG Yuqi | LI Man Fung | Kevin LUK

Background & Objective

In the business world, especially e-commerce area, sellers see sales revenues as the most valuable assets. It is obvious that the tech-giants are knowing our preference better and better. Amazon & Taobao always know what you are planning to buy recently, YouTube and Tok-tok seems keep showing you the videos that perfectly fit your interests, and those annoying Google ads that show up every day are also selected according to your daily browsing histories. Thus, recommending the right thing that a certain individual may wish is of vital importance. If the company can understand and anticipate every person's preference, the marketing teams can make appropriate investments in their advertising.

Our objective is to use the friends network data from social media to train a Random Walk model and forecast who could be your next friends, just like what Facebook does, and evaluate this model by calculating the AUC Score. Then use this recommendation system to boost the business intelligence of today's social network.

Cloud Computing Platforms & Tools

1. Amazon S3 - Storage Platform

Amazon S3 is useful for storing large amounts of unstructured object data. We create an S3 bucket (comp4651-project-yzhangfe) to store the training and testing files. In Databricks, we can access AWS S3 buckets by mounting buckets using DBFS, with the provided ACCESS_KEY, SECRET_LEY, and AWS_BUCKET_NAME.

2. Databricks (Spark) - Visualization and Model Training Tool

To have better data visualization and more efficient model training, we use Databricks (Spark) as our visualization and model training tool, as it supports a lot of useful functions, like Map-Reduce, SQL context, etc. Also, a cloud computing platform like Databricks is powerful for dealing with large size data since we have 2,796,742 data in our training set and 1,920,014 data in our test set.

Data Analysis and Preprocessing

Firstly, we did a very detailed explorative data analysis to fully comprehend the dataset provided. Besides the mechanisms mentioned in the lecture and tutorial, we have tried many different special mechanisms to try to achieve a higher accuracy score. Finally, we used a GraphSAGE model through neighborhood aggregation and trained the embedding of edges with XGBoost, the model has attained an AUC of 0.965. The parameters, algorithm, the mechanism, codes, and final results will be discussed in detail.

1. **Feature Engineering**
   By mounting the Amazon S3 using DBFS, we can have a briefly preview for the raw data. The

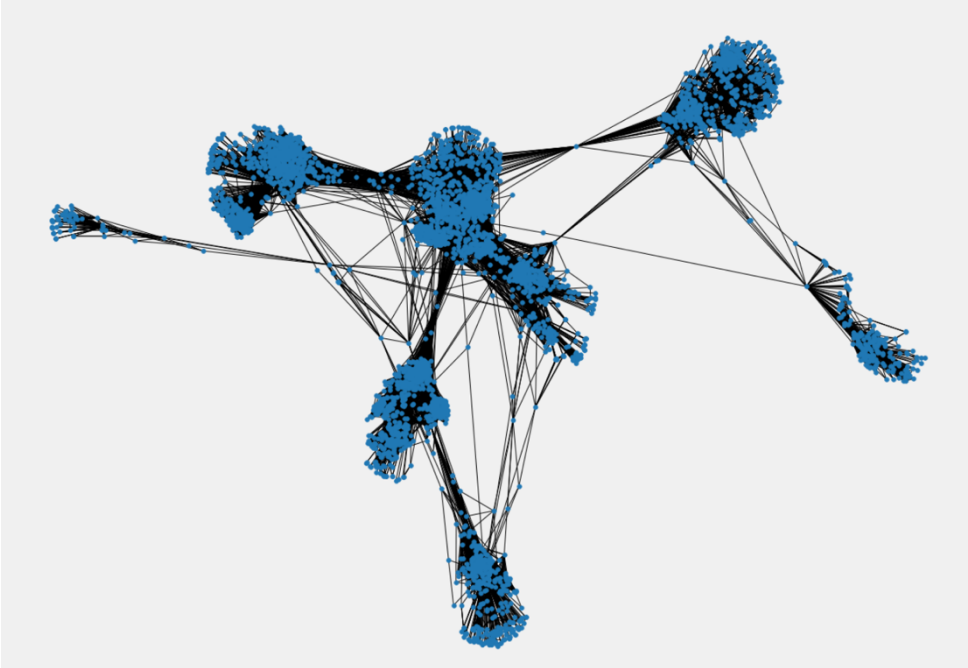
given features are user_id and friends with an extremely large size. We first need to preprocess the data and extract the features that we need.

```
import urllib
ACCESS_KEY = "AKIAVWMBO7IYZT4QM56I"
SECRET_KEY = "W5iaJpZ3GPbGrEJC4rIX2H/LYTadEHZI+baN/Vya"
encoded_secret_key = SECRET_KEY.replace("/", "%2F")
AWS_BUCKET_NAME = "comp4651-project-yzhangfe"
MOUNT_NAME = "s3"
#dbutils.fs.mount("s3a://%s:%s@%s" % (ACCESS_KEY, encoded_secret_key, AWS_BUCKET_NAME), "/mnt/%s" % MOUNT_NAME)
#display(dbutils.fs.ls("/mnt/%s" % MOUNT_NAME))
display(dbutils.fs.ls("/mnt/s3/data"))
```

path	name	size
dbfs:/mnt/s3/data/test.csv	test.csv	1920014
dbfs:/mnt/s3/data/train.csv	train.csv	2796742
dbfs:/mnt/s3/data/valid.csv	valid.csv	607552

2. **Visualization**

By visualizing the data using networkx, we could find the potential relationships among each raw data (users and their friends), thus build an embedding model with different mechanism to extract the adequate features and cut down the sample size.



3. **Data Preprocessing**

Continue from the visualized graph, we built a load_data user-defined function to add each friend as an edge and use each edge to build different nodes for the whole network. The total number of edges and nodes for the training sets are as follows:

```
def load_data(file_name):
    """
    read edges from an edge file
    """
    edges = list()
    df = pd.read_csv(file_name)
    #df = sqlContext.read.format('com.databricks.spark.csv').options(delimiter=',',
le_name)

    for idx, row in df.iterrows():
        user_id, friends = row["user_id"], eval(row["friends"])
        for friend in friends:
            # add each friend relation as an edge
            edges.append((user_id, friend))
    edges = sorted(edges)
    print("The number of edges is", len(edges))
    nodes = list(set(chain.from_iterable(edges)))
    print("The number of nodes is", len(nodes))
    return edges, nodes
```

We have built 100,000 edges and 8328 nodes and greatly decreased the data sizes to boost the following model training process. We will train this embedding model through different mechanism with XGBoost and calculate the AUC to evaluate this model and visualize it by a heatmap to select the best set of parameters.

```
train_file = "data/train.csv"
valid_file = "data/valid.csv"
test_file = "data/test.csv"

np.random.seed(0)
print("Load train edges...")
train_edges, train_nodes = load_data(train_file)
print("\nGenerate graph...")
graph = construct_graph_from_edges(train_edges)
```

```
Load train edges...
The number of edges is 100000
The number of nodes is 8328

Generate graph...
number of nodes: 8328
number of edges: 100000
```

Machine Learning Pipeline

After the explorative data analysis and data preprocessing. We have used many embedding methods includes Deepwalk, node2vec and GraphSAGE with different classifiers to try to increase the accuracy score. The best result we could attain is by GraphSAGE + XGBoost, which attain 0.965 AUC in the validation set.

1. **Create a 2-layer GraphSAEG Model**

```
class GraphSAGE(nn.Module):
    def __init__(self, in_feats, h_feats):
        super(GraphSAGE, self).__init__()
        self.conv1 = SAGEConv(in_feats, h_feats, 'mean')
        self.conv2 = SAGEConv(h_feats, h_feats, 'mean')
        self.conv3 = SAGEConv(h_feats, h_feats, 'mean')

    def forward(self, g, in_feat):
        h = self.conv1(g, in_feat)
        h = F.relu(h)
        h = self.conv2(g, h)
        h = F.relu(h)
        h= self.conv3(g, h)
        return h
```

2. **Set up Loss and Optimizer**

   In this case, we used Adam as the optimizer and loss will be in the training loop.

3. **Training**

   We first trained 150 epochs in total.

```
# ----------- 4. training -------------------------------- #
all_logits = []
for e in range(150):
    # forward
    h = model(g, g.ndata['feat'])
    pos_score = pred(train_pos_g, h)
    neg_score = pred(train_neg_g, h)
    loss = compute_loss(pos_score, neg_score)

    # backward
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if e % 5 == 0:
        print('In epoch {}, loss: {}'.format(e, loss))
h_list.append(h)
```

We then generated 100,000 false edges and define the edge vectors by the sum of two connecting nodes' vectors.

```
em_data_list = [np.zeros((200000,embed_length[i])) for i in range(4)]
for n in range(4):
    i = -1
    for (a,b) in train_edges:
        i += 1
        add_list = get_embedding(a, h_list[n])+get_embedding(b, h_list[n])
        em_data_list[n][i] = add_list.detach().numpy()

    i = 99999
    for (a,b) in train_false_edges:
        i += 1
        add_list = get_embedding(a, h_list[n])+get_embedding(b, h_list[n])
        em_data_list[n][i] = add_list.detach().numpy()
```

4. XGBoost

Then we train 100,000 true edges and 100,000 false edges in XGBoost classifier. Through our trials and errors, we have found that using XGBoost can greatly improve the model performance.

```
import xgboost as xgb
xg_reg_list = []
for i in range(4):
    data = pd.DataFrame(em_data_list[i])
    lable = pd.DataFrame(ytrain)
    dtrain = xgb.DMatrix(data, label=lable)
    xg_list = []
    for lr in lr_list:
        xg_reg = xgb.XGBRegressor(objective ='binary:logistic', colsample_bytree = 0.3, learning_rate = lr,
                        max_depth = 20, alpha = 10, n_estimators = 100)
        xg_reg.fit(data,lable)
        xg_list.append(xg_reg)
    xg_reg_list.append(xg_list)
```

Results and Evaluations

1. Evaluations

To evaluate the model performance, we processed the validation sets in the same way, used the XGBoost regressor to make the prediction, and calculated the final AUC of the model. The final AUC is 0.96, which proves it is a good model with high confidence level.

```
preds = xg_reg.predict(em_valid_data)
preds
```

```
array([0.8161394 , 0.07216774, 0.70945174, ..., 0.00217352, 0.11858444,
       0.00255821], dtype=float32)
```

```
[ ] auc_scores=roc_auc_score(yvalid, preds)
    print (auc_scores)
```

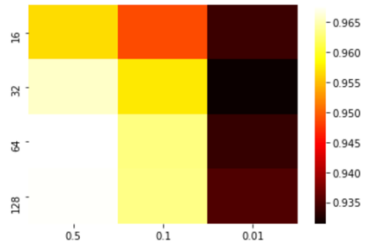
```
0.9599688721523806
```

2. Parameter Selections and Further Improvements

To select the best sets of parameters, we first trained the model for 12 times, each with a different parameter set, and selected the best model with the best sets of parameters. We visualized the model performance in a heatmap. The results are as follows (the left graph):

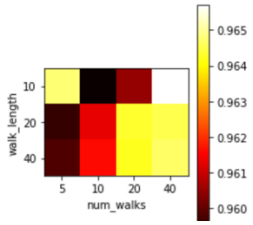
```
import seaborn
result.columns = [0.5, 0.1, 0.01]
result.rename(index={0: 16, 1:32, 2:64, 3:128}, inplace=True)

seaborn.heatmap(result, cmap='hot')
```

```
<AxesSubplot:>
```




```
plt.imshow(a, cmap="hot", interpolation="nearest")
plt.colorbar()
plt.yticks(ticks=[0,1,2], labels=[10, 20, 40])
plt.ylabel("walk_length")
plt.xticks(ticks=[0,1,2,3], labels=[5, 10, 20, 40])
plt.xlabel("num_walks")
```

```
Text(0.5, 0, 'num_walks')
```



Since we are using the GraphSAGE method, we firstly used the embedding size as the y-axis and the learning rate as the x-axis, and an embedding size of 128 along with a learning rate of 0.5 could generate the best performance.

Since this is a model with the random_walk algorithm, we have also slightly modified our model and built another heatmap (right graph). We used num_walks as the x-axis and the walk_length as the y-axis, and a num_walks of 40 along with a walk length of 10 could generate the best performance.

Discussion and Finding

1. Databricks and Spark

The memory of a Databricks cluster is only 6G which is not enough for a large dataset and our training result could be better if a cluster with better performance is provided. In Spark, several continuous join executions will lead to duplicate columns with the same name and it's not always user-friendly when pass parameters to our user-defined functions, so it is better to preprocess the data before joining. As the number of features grows, the execution time of every line of code will grow fast. It is necessary to cache the result every several steps to decrease the total running time.

2. Spark DataFrame and Pandas DataFrame

Comparing two kinds of dataframe, we find that the function provided by Spark is much fewer than Pandas. However, Spark supports the use of SQL, which could greatly compensate for those missing functions. For example, when using Pandas, we could directly do the division between two columns, but Spark does not support this action. Instead, we need to preprocess the datasets and change the dataframe into a table in SQLcontext.

Data Source and Source Code

Raw Data: https://github.com/hkust-comp4651-21S/project-comp4651_project_the3abandoned.git
Dataset on Amazon S3: https://s3.amazonaws.com/comp4651-project-yzhangfe/data_
Report and Implementation: https://github.com/zdxdsw/HKUST-COMP4651-Course-Project.git