DSC 102: Systems for Scalable Analytics

Programming Assignment 2

1 Introduction

This assignment has two parts. In the first part we will conduct feature engineering for the Amazon dataset. In the second part we will train ML models using the extracted features. We will use Apache Spark on a SDSC[1] cluster. You will need to login to the **login node**[2] of the SDSC cluster and create a Spark cluster using the scripts that we provide. You will then connect to the cluster using SSH tunnels and finish all the developments and tests there. You are not expected to code anything locally.

2 Dev-kit

A dev-kit consisting skeletons and other necessary files has been provided to you along with this document. You first need to clone the dev-kit to your home directory in the login node of the SDSC cluster. When you spawn the cluster following the instructions in Section **??**, this dev-kit should be prepared on your cluster's master node automatically, you do not need to manually download it.

Within the dev-kit there are several files:

```
assignment2.ipynb -- the task descriptions and a playground for your development
assignment2.py -- the deliverable of this assignment, your final file to submit
------------------------------
cluster-manager.sh --
spark-cluster.yaml.template --
pa2_main.py --
utilities.py -- above files are necessary for your code to run. Do not modify any of them
```

3 Dataset Description

For the first part you are expected to extract features from three tables, their schemas and descriptions are listed below:

```
1. product
    |-- asin: string, the product id, e.g., 'B00I8HVV6E'
    |-- salesRank: map, a map between category and sales rank, e.g., {'Home & Kitchen': 796318}
    |    |-- key: string, category, e.g., 'Home & Kitchen'
    |    |-- value: integer, rank, e.g., 796318
    |-- categories: array, list of list of categories, e.g., [['Home & Kitchen', 'Artwork']]
    |    |-- element: array, list of categories, e.g., ['Home & Kitchen', 'Artwork']
    |    |    |-- element: string, category, e.g., 'Home & Kitchen'
    |-- title: string, title of product, e.g., 'Intelligent Design Cotton Canvas'
    |-- price: float, price of product, e.g., 27.9
    |-- related: map, related information, e.g., {'also_viewed': ['B00I8HW0UK']}
    |    |-- key: string, the attribute name of the information, e.g., 'also_viewed'
    |    |-- value: array, array of product ids, e.g., ['B00I8HW0UK']
    |    |    |-- element: string product id , e.g., 'B00I8HW0UK'
2. product_processed
```

[1] San Diego Supercomputing Center

[2] dsmlp-login.ucsd.edu

```
    |-- asin: string, same as above
    |-- title: string, title column after imputation, e.g., 'Intelligent Design Cotton Canvas'
    |-- category: string, category column after extraction, e.g., 'Home & Kitchen'
3. review
    |-- asin: string, same as above
    |-- reviewerID: string, the reviewer id, e.g., 'A1MIP8H7G33SHC'
    |-- overall: float, the rating associated with the review, e.g., 5.0
```

The `review` table will be useful for extracting the rating information for each product in Task 1. We will be working primarily with `product` table throughout Task 1-4. `product_processed` is used for Task 5-6.

For the second part you are expected to train ML models on the extracted features for predicting the user rating for a product. For your convenience we have created ML-ready data which contain user and product features and also the rating. We will provide two tables one for training and one for testing. The schemas of the two tables is shown below:

```
1. ml_features_train
    |-- features: SparseVector(float), SparseVector of concatenated
    features from user and product data (all features are continuous
    features)
    |-- overall: int, review rating
2. ml_features_test
    |-- features: Vector(float), same as above
    |-- overall: int, , same as above
```

All the datasets required for this assignment can be found in a NFS directory[3]. We will be reading from this NFS directly which gets mounted with the Spark cluster. You do not need to download any of them.

4 Tasks

Part 1 and 2 two has 6 and 2 tasks respectively. Altogether, you are required to complete eight tasks in total. In each task you will need to implement a function `task_i()`. The function signatures and return types are fixed and provided to you in the dev-kit. Each function will take in several inputs and conduct the desired transformations. At the end of each task you will be asked to extract several statistical properties (mean, variance, RMSE etc.) from the transformed data. You will need to programmatically put these properties in a python dictionary named `res`, the schema of which is also given. Each of the tasks will be tested in unit. It means each function you write will be tested in isolation from the rest. We will award partial points even if some tasks failed.

For the tasks you can use any combinations of the Spark APIs available in the environment. However, you can select (by setting a global variable called `INPUT_FORMAT`) one of the three APIs for inputs: DataFrame, RDD, Koalas. Inside your function body, you have the freedom to switch between them.

Important: Task 2, 3, 5, 6, 7, 8 **cannot** be solved solely with Koalas. Currently Koalas does not support nested types so Task 2, 3 are not doable with it, also it does not have the required ML support for Task 5, 6, 7, 8. You will need to switch to other APIs for these tasks.

4.1 Conventions

These rules apply to all the tasks.

4.1.1 Results format

Each task comes with a pre-defined schema for the output results. The result must be stored as python native dictionary and must contain all the keys and nested structures. You must only use python built-in datatypes. For instance, if your value is of datatype `np.float64()`, you must first cast it into python `float`.

For the following schema:

[3] `</dsc102-pa2-public/dataset>`

```
res
 | -- single_value: int -- an integer number
 | -- list_of_values: list -- a list of values
     | -- element: float -- a float number
```

A desired python code snippet to compose up the dictionary would be similar to:

```python
...
data = ... # Your transformed data
res = {
    'single_value': None,
    'list_of_values': [None]
} # Skeleton given for the result
res['single_value'] = int(data.some_op())
res['list_of_values'] = [float(data.some_op()), float(data.some_op()), float(data.some_op())]
...
```

4.1.2 Dealing with `null`, `None` and `NaN`

The input tables contain `null` (or `None` as in RDD/python, or `NaN` as in Koalas/pandas, we will be using these notations interchangeably) and dangling references. You do not need to deal with dangling reference unless instructed. For `null` values we will follow the common practice in SQL world: unless instructed otherwise, you need to ignore all `nulls` when calculating statistics such as count, mean and variance. Of course, do not ignore `null` when you are explicitly asked to count the number of `null` entries.

4.2 Task1: mean and count of ratings

First you will aggregate and extract some information from the user review table. We want to know for each product, what are the mean rating and the number of ratings it received. Implement a function `task_1` that does the following:

1. For each product ID `asin` in `product_data`, calculate the average rating it received. The ratings are stored in column `overall` of `review_data`.
2. Similarly, put the count of ratings for each product in a new column named `countRating`.
3. You need to conduct the above operations, then extract some statistics out of the generated columns. You need to put the statistics in a python dictionary named `res`. The description and schema of it are as follows:

```
res
 | -- count_total: int -- count of total rows of the entire table after your operations
 | -- mean_meanRating: float -- mean value of column meanRating
 | -- variance_meanRating: float -- variance of meanRating
 | -- numNulls_meanRating: int -- count of nulls of meanRating
 | -- mean_countRating: float -- mean value of countRating
 | -- variance_countRating: float -- variance of countRating
 | -- numNulls_countRating: int -- count of nulls of countRating
```

If for a product ID, there is not a single reference in `review`, meaning it was never reviewed, you should put `null` in both `meanRating` and `countRating`.

4.3 Task 2: flatten `categories` and `salesRank`

Implement a function `task_2()` to conduct the following operations:

1. For the `product` table, each item in column `categories` contains an array of arrays of hierarchical categories. The schema is `ArrayType(ArrayType(StringType))`. We are only going to use the most general category, which is the first element of the nested array: `array[0][0]`. For each row, put the the first element of `categories` in a new column `category`. If `categories` is `null` or empty, put a `null` in your new column.

2. On the other hand, each entry in column `salesRank` is a key-value pair: `(bestSalesCategory, rank)`. Your task is to flatten it into two columns. Put the key in a new column named `bestSalesCategory` and the value in `bestSalesRank`. Put `null` if the original entry was `null` or empty.

3. The schema of output is as follows:

```
res
 | -- count_total: int -- count of total rows of the entire table
 | -- mean_bestSalesRank: float -- mean value of bestSalesRank
 | -- variance_bestSalesRank: float -- variance of bestSalesRank
 | -- numNulls_category: int -- count of nulls of category
 | -- countDistinct_category: int -- count of all distinct values of category
 | -- numNulls_bestSalesCategory: int -- count of nulls of bestSalesCategory
 | -- countDistinct_bestSalesCategory: int -- count of distinct values of bestSalesCategory,
                                              excluding nulls
```

4.4 Task 3: flatten `related`

Values of `related` column are maps with four keys/attributes: `also_bought`, `also_viewed`, `bought_together`, and `buy_after_viewing`. Each value of these maps contains an array of product IDs. We call them attribute arrays. You need to calculate the length of the arrays and find out the average prices of the products in these arrays.

The logic for all four attributes are identical. For the sake of simplicity, you are only required to flatten the `also_viewed` attribute. Your task is to implement function `task_3()` that does the following:

1. For each row of `related`, you need to :
   1. Calculate the mean price of all products from the `also_viewed` attribute array. Put it in a new column `meanPriceAlsoViewed`. Remember to ignore the products if they do not match any record in `product`, or if they have `null` in price. Do **not** ignore the product if it has `price=0`
   2. Similarly, put the length of that array in a new column `countAlsoViewed`. You do not need to check if the product IDs in that array are dangling references or not. Put `null` (instead of zero) in the new column, if the attribute array is `null` or empty

2. The schema of output is as follows:

```
res
 | -- count_total: int -- number of rows of the entire processed table
 | -- mean_meanPriceAlsoViewed: float -- mean value of meanPriceAlsoViewed
 | -- variance_meanPriceAlsoViewed: float -- variance of meanPriceAlsoViewed
 | -- numNulls_meanPriceAlsoViewed: int -- count of null-value entries of meanPriceAlsoViewed
 | -- mean_countAlsoViewed: float -- mean value of countAlsoViewed
 | -- variance_countAlsoViewed: float -- variance of countAlsoViewed
 | -- numNulls_countAlsoViewed: int -- count of null-value entries of countAlsoViewed
```

4.5 Task 4: data imputation

You may have noticed that there are lots of `nulls` in the table. Now your task is to impute them with meaningful values that can be used for ML.

Since the schema is already flattened, now we only have two datatypes in our table: numerical (including integer and floating numbers) and string. Now you need to impute a numerical column `price`, as well as a string column `title`.

1. Please implement a function `task_4()`. For column `price`, first cast it to float type. Then impute the `nulls` with the mean value of all the rest values. Store the outputs in a new column `meanImputedPrice`.

2. Same as above, but this time impute with the **median** value. Store the outputs in a new column `medianImputedPrice`.

3. As for the string-typed columns, we want to simply impute `nulls` **and empty strings** with a special string '`unknown`'. Store the outputs in a new column `unknownImputedTitle`.

4. The schema of output is as follows:

```
res
 | -- count_total: int -- count of total rows of the entire table after above operations
 | -- mean_meanImputedPrice: float or None -- mean
 | -- variance_meanImputedPrice: float -- variance
 | -- numNulls_meanImputedPrice: int -- count of null-value entries
 | -- mean_medianImputedPrice: float or None -- mean
 | -- variance_medianImputedPrice: float -- variance
 | -- numNulls_medianImputedPrice: int -- count of null-value entries
 | -- numUnknowns_unknownImputedTitle: float -- count of 'unknown' value entries
```

4.6 Task 5: embed `title` with `word2vec`

This task assumes the `title` *column is already imputed with* `unknown`*. We have provided the imputed data table* `product_processed_data`*.*

In this task we want to transform `title` into a fixed-length vector via word2vec.

1. You need to implement function `task_5()`. For each row, convert `title` to lowercase, then split it by whitespace (`' '`) to an array of strings, store the array in a new column `titleArray`
2. Train a word2vec model out of column `titleArray`. Do not try to implement word2vec yourself. Instead, use `M.feature.Word2Vec`. See instructions below.
3. For each of the three words inputed as `<word_0>`, `<word_1>`, and `<word_2>`, use your obtained word2vec model to get the 10 closest synonyms along with similarity scores (cosine similarity of word vectors). `M.feature.Word2Vec` also has built-in method for this task.
4. The schema of output is as follows:

```
res
 | -- count_total: int -- count of total rows of the entire table
 | -- size_vocabulary: int -- the size of the vocabulary of your word2vec model
 | -- word_0_synonyms: list -- synonyms tuples of word_0
 |    | -- element: tuple -- tuple of format (synonym, score)
 |    |    | -- element: string -- synonym
 |    |    | -- element: float -- score
 | -- word_1_synonyms: list
 |    | -- element: tuple
 |    |    | -- element: string
 |    |    | -- element: float
 | -- word_2_synonyms: list
 |    | -- element: tuple
 |    |    | -- element: string
 |    |    | -- element: float
```

word2vec instructions:

1. Set `minCount`, the minimum number of times a token must appear to be included in the word2vec model's vocabulary, to `100`.
2. Set the dimension of output word embedding to `16`.
3. You need to set the random seed as `SEED`, this is a global variable defined to be 102.
4. Set `numPartitions` to 4.
5. You should keep all other settings as default.
6. `M.feature.Word2Vec` is not fully reproducible (although we have set the seed here). We are aware of the issue and your score will not be affected by its internal randomness.

4.7 Task 6: one-hot encoding `category` and PCA

Assume `categories` is already flattened and `unknown` imputed for the input data. We have provided you with the preprocessed table

Now you need to one-hot encode the categorical features. Meanwhile, they may be correlated. So as a practice, we would like to run PCA on these categories.

1. Implement function `task_6()`. First one-hot encode `category` and put the resulted vectors in a new column `categoryOneHot`. Ensure the dimension of generated vectors equals to the size of domain. For example, if we have three categories in total: `V = {'Electronics', 'Books', 'Appliances'}`. Then the encoding for 'Electronics' can be `[1, 0, 0] or [0, 1, 0] or [0, 0, 1]`, but the dimension of this vector must be 3.

   Hint: For DataFrame, before encoding a string-typed column, you may have to first convert it to a column of numerical indices with `M.feature.StringIndexer`. Then use `M.feature.OneHotEncoderEstimator` to do the encoding. Set `dropLast` argument to false.

   For RDD, you may need to implement the one-hot-encoding logic yourself. Consider to build the one-hot mapping locally and broadcast it.

2. Apply PCA on the one-hot-encoded column. Reduce the dimension of each one-hot vector to `15`, put the transformed vectors in a new column `categoryPCA`. On DataFrame, use `M.feature.PCA`. On RDD, see instructions[4].

3. Column `categoryOneHot` and `categoryPCA` will be of VectorType. You do not need to worry if the vectors are sparsely or densely represented.

4. The schema of output is as follows::

```
res
 | -- count_total: int -- count of total rows of the entire transformed table
 | -- meanVector_categoryOneHot: list -- mean vector of transformed one-hot-encoding vectors
 |    | -- element: float -- element of the mean vector, from first to last dimension
 | -- meanVector_categoryPCA: list -- mean vector of the PCA-transformed vectors
 |    | -- element: float
```

4.8 Task 7: Train a Decision Tree Regression model

Assume we have extracted all the product features and combined it with user features generated in PA1. We are providing you with two processed ML-ready feature tables for training and testing ML models.

Now you need to train a Decision Tree Regression model using this data to predict the user rating for a product.

1. Implement function `task_7()`. Train a Decision Tree Regression model using the training data. The **max tree depth** parameter of the model **must be set to 5**. All other parameters of the model should be left to **default** values.
2. Use the trained model to generate predictions on test data. Calculate the root mean square error (RMSE) of the test predictions and report it in the output.
3. The schema of output is as follows::

```
res
 | -- test_rmse: float -- RMSE of the test predictions
```

4.9 Task 8: Hyperparameter tuning for the Decision Tree Regression model

In `task_7()` we fixed the max tree depth parameter of the model and trained a single model. Now we perform hyperparameter tuning to select the best max tree depth.

1. First create new training and validation data from the original training data. Use a random split of 75/25.

2. Train Decision Tree Regression models with max tree depth values of 5, 7, 9, and 12. Also, Calculate the RMSE of validation data predictions and report it in the output.

[4] <https://spark.apache.org/docs/2.4.4/mllib-dimensionality-reduction>

3. Based on the validation RMSE values, pick the best model and use it to generate prediction on test data. Report test RMSE in the output.
4. The schema of output is as follows::

```
res
 | -- test_rmse: float -- RMSE of the test predictions generated by the best model based on
 validation RMSEs
 | -- valid_rmse_depth_5: float -- RMSE of the validation set predictions
 generate by max tree depth of 5
 | -- valid_rmse_depth_7: float -- same as above but w/ depth 7
 | -- valid_rmse_depth_9: float -- same as above but w/ depth 9
 | -- valid_rmse_depth_12: float -- same as above but w/ depth 12
```

5 Deliverables

Code up all the tasks in the designated places in `assignment2.py`. Then rename the file to `assignment2_<your team id>.py`. For instance, if your team id is 18, then your filename would be `assignment2_18.py`. Submit this file on Canvas, only one team member needs to do so.

6 Getting started

6.1 How to read and copy commands in this section

1. In this section, we have three different hosts where you can type commands: your own computer (`local`), the front-end node (`dsmlp-login`), and Spark master node (`spark-master`). All shell commands will be given you in the format of:

   ```
   @<host>: <commands>
   ```

   For instance, if we would like you to list the directory on your own computer, the command would be:

   ```
   @local: ls
   ```

   In this scenario, what you need to do is open a terminal (Linux and OS X users) or a PowerShell (Windows users), copy paste `ls`, and execute it.
2. On the other hand, if you are given a command like:

   ```
   @dsmlp-login: ls
   ```

   This means the command `ls` needs to be executed on the front-end node. You need to first SSH into it and then execute the command. We will show you how to do the SSH.
3. Sometimes you may encounter angular brackets `<XXX>`, in this situation you will need to substitute it with the desired value. Do **not** leave the brackets. For example, the following command

   ```
   @local: echo <pid>
   ```

   You need to put your pid in the command and the command you actually run would become (assuming your pid is a10000000):

   ```
   @local: echo a10000000
   ```

6.2 SSH into the front-end node

First, use your ETS account and password to sign into the front-end node via SSH from your own machine:

```
@local: ssh <ETS account>@dsmlp-login.ucsd.edu
```

Your ETS account name is usually the same as your UCSD email name. If you have trouble finding it or you forgot the password, use ETS Account Lookup[5].

[5] `<https://sdacs.ucsd.edu/~icc/index.php>`

6.3 Prepare the dev-kit

You only need to do this once. In the front-end's shell, clone the repo prepared to you by:

```
@dsmlp-login: git clone --single-branch --branch its https://github.com/makemebitter/dsc102-ucsd-public.git
```

This should create a folder named `dsc102-ucsd-public` in your home directory.

6.4 Launch the cluster

1. In the front-end's shell, enter the home directory of dev-kit:

   ```
   @dsmlp-login: cd dsc102-ucsd-public
   ```

2. Create the cluster via:

   ```
   @dsmlp-login: ./cluster-manager.sh create
   ```

   Wait until the cluster is up and it will output instructions similar to below:

   ```
   => Successfully initiated the Spark cluster
   => Next create a SSH tunnel from your personal computer using the following command:
      ssh -N -L 127.0.0.1:8888:127.0.0.1:XXX -L 127.0.0.1:8080:127.0.0.1:XXX -L 127.0.0.1:4040:127.0.0.1:XXX XXX@dsmlp-login.ucsd.edu
   => Link to PySpark/Jupyter UI: http://127.0.0.1:8888?token=XXXXXXXXXX
   => Link to Spark cluster manager UI: http://127.0.0.1:8080
   => Link to Spark job UI: http://127.0.0.1:4040
   ```

3. Copy paste the port-forwarding command to a new shell on **your computer**:

   ```
   @local: ssh -N -L 127.0.0.1:8888: ... ... ... ... @dsmlp-login.ucsd.edu
   ```

4. In your browser, connect to the following. Jupyter notebook

   ```
   http://127.0.0.1:8888?token=...
   ```

   Spark cluster manager UI

   ```
   http://127.0.0.1:8080
   ```

   Spark job UI

   ```
   http://127.0.0.1:4040
   ```

5. The working directory of this Jupyter notebook is the home directory of your front-end node. So all your modifications to the assignment files will be saved and no files are stored in the cluster. In Jupyter notebook, go to directory `dsc102-ucsd-public/src`, rename `assignment2.ipynb` to `assignment2_<your pid>.ipynb` and continue the assignment there.

6.5 Test and submit

You will **not** submit the notebook. Instead, you need to put your implementations of `task_1` to `task_8`, along with all the dependencies you imported and helper functions you defined, in the file co-located with the notebook: `assignment2.py`.

If you are collaborating in team, please combine your work into one single file. Only **one** person needs to submit the final file. Do **not** modify the filename yet.

6.5.1 Test your file

Before submitting the file, you need to make sure your script runs under the deployment environment, otherwise you may lose points.

1. From the shell on front-end node, query the master node's pod name:

   ```
   @dsmlp-login: kubectl get pods
   ```

   The name would be in the format of `spark-master-XXX-XXX`.

2. SSH into the master node via

```
@dsmlp-login: kubectl exec -it <spark-master-XXX-XXX> bash
```

3. On the master node shell, go to your root directory of scripts

```
@spark-master: cd /home/dsc102-ucsd-public/src
```

4. Run PA2 with the following command, do not modify anything except `<your pid>`:

```
@spark-master: spark-submit \
--py-files utilities.py,assignment2.py \
--files log4j-spark.properties \
--deploy-mode client \
--driver-java-options "-Dlog4j.configuration=file:log4j-spark.properties" \
--conf "spark.executor.extraJavaOptions=-Dlog4j.configuration=file:log4j-spark.properties" \
pa2_main.py --pid <your pid>
```

Make sure your script can execute and try to pass as many tests as you can.

6.5.2 Submit your file

Go to your AWS console, navigate to S3 buckets and find the bucket named `<your pid>-pa2`. Download the `assignment2.py` file to your own machine.

Then rename the file to `assignment2_<your team id>.py`. For instance, if your team id is 18, then your filename would be `assignment2_18.py`.

Upload this file to Canvas, only one of the team members needs to do so.

6.6 Delete your cluster

Don't forget to delete the cluster. No data will be lost so you should do this whenever you are not using it.

1. Go to the dev-kit directory from your front-end node's shell

```
@dsmlp-login: cd ~/dsc102-ucsd-public
```

2. Delete the cluster via

```
@dsmlp-login: ./cluster-manager.sh delete
```