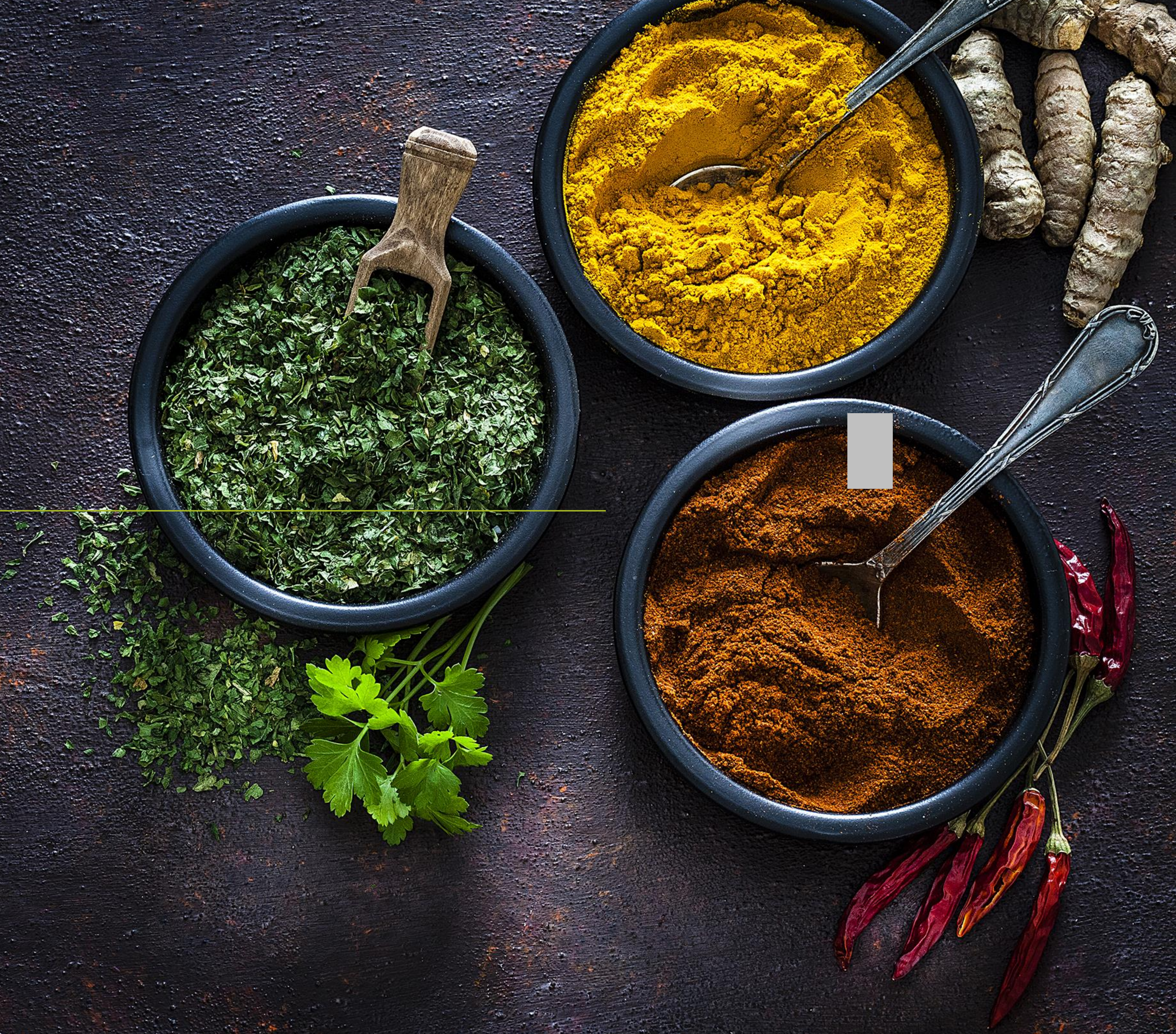
RECIPE RECOMMENDER ASSIGNMENT EDA

SUBMITTED BY –

RASHMI PANDEY, RUPALI PAL,

PHAM THIEU QUAN



Objective

In the role of a Machine Learning engineer at food.com, our primary mission is to develop a sophisticated recipe recommendation engine. This system aims to personalize recipe suggestions according to user preferences and the recipes they are presently exploring. The key to success lies in engaging users more deeply, potentially opening doors to increased commercial prospects. The effectiveness of our recommender system is crucial, as it will have a direct effect on the site's revenue streams. Although constructing such a system from the ground up demands considerable effort, this project focuses on the critical tasks of data exploration and feature development essential for crafting an efficient recommender.

Steps to approach the problem

- Create and launch an EMR Cluster on Amazon AWS
- Create and launch a Jupyter Notebook on top of this cluster
- Perform all the necessary tasks provided in task list

Task-List

01

Task 1: Read the data: Read RAW_recipes.csv from S3 bucket. Ensure each field has the correct data type.

02

Task 2: Extract individual features from the nutrition column: Separate the array into seven individual columns to create new columns named calories, total_fat_PDV, sugar_PDV, sodium_PDV, protein_PDV, saturated_fat_PDV, and carbohydrates_PDV.

03

Task 3: Standardize the nutrition values: Convert the nutritional values to per 100 calories.

Task-1

Task 1:

Read the data: Read RAW_recipes.csv from S3 bucket.

Ensure each field has the correct data type.

```
# Task 01 Cell 1 out of 1

raw_recipes_df = (spark.read.csv("s3://finalnewbucket1rush/e-26E90VMNWUJI3Y0S5OM5FAD8G/RAW_recipes_cleaned.csv",
                                 inferSchema = True, header = True))

# Please forward the exact name of data frames and columns as suggested in the code.
# It will ensure that the assert commands function correctly.
```

Task -2 and Task -3

Task 2:

- Extract individual features from the nutrition column: Separate the array into seven individual columns to create new columns named calories, total_fat_PDV, sugar_PDV, sodium_PDV, protein_PDV, saturated_fat_PDV, and carbohydrates_PDV.

Task 3:

- Standardize the nutrition values: Convert the nutritional values to per 100 calories.

```
# Task 02 Cell 2 out of 3
# STEP 2.2 - split the neutrition string into seven individial values.
# Create an object to split the nutrition column
import pyspark

nutrition_cols_split = pyspark.sql.functions.split(raw_recipes_df['nutrition'],',')# pyspark function to split value

# Write a loop to extract individual values from the nutrition column

for col_index, col_name in enumerate(nutrition_column_names):
    # col_index holds the index number of each column, e.g., calories will be 0
    # col_name holds the name of each column

    raw_recipes_df = (raw_recipes_df.withColumn(col_name,nutrition_cols_split.getItem(col_index).cast("float"))
                                        # pyspark function to extract individual values from the nutrition_cols_spli
                                        # You can also cast the extracted value to floats in the same code.
                                               )
```

Solution to Task 2

complete the code in the following cell

```
In [13]: # Task 02 Cell 1 out of 2
# 2.1 - string operations to remove square brackets
import pyspark
from pyspark.sql import functions as F
raw_recipes_df = (raw_recipes_df
                  .withColumn('nutrition', (F.regexp_replace("nutrition","[\[\]]",""))))
                            # add code to remove square brackets
                            # pyspark function to replace string characters
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
```

```
In [14]: # Task 02 Cell 2 out of 3
# STEP 2.2 - split the neutrition string into seven individual values.
# Create an object to split the nutrition column


nutrition_cols_split = pyspark.sql.functions.split(raw_recipes_df['nutrition'],',') # pyspark function to split values based on c

# Write a loop to extract individual values from the nutrition column

for col_index, col_name in enumerate(nutrition_column_names):

    # col_index holds the index number of each column, e.g., calories will be 0
    # col_name holds the name of each column

    raw_recipes_df = (raw_recipes_df.withColumn(col_name, nutrition_cols_split.getItem(col_index).cast("float")))
                                # pyspark function to extract individual values from the nutrition_cols_split object
                           # You can also cast the extracted value to floats in the same code.
```

Solution to Task 3

Complete the code in the following cell

```
In [19]: # Task 03 Cell 1 out of 1

for nutrition_col in nutrition_column_names: # loop over each of the newly created nutrition columns
    if nutrition_col != "calories": # the calories column should not be a part of the transformation exercise
        # following code will name the new columns
        nutrition_per_100_cal_col = (nutrition_col
                                     .replace('_PDV','')
                                     +'_per_100_cal')
        raw_recipes_df = raw_recipes_df.withColumn(nutrition_per_100_cal_col,
                                        raw_recipes_df[nutrition_col]*100/ raw_recipes_df["calories"]
                                             # pyspark code to recreate the intended transformation
                                               )

        # You might end up adding nulls to the data because of our intended transformation.
        # Perform a fill na operation to fill all the nulls with 0s.
        # You must limit the scope of the fill na to the current column only.

        raw_recipes_df = raw_recipes_df.fillna(value=0,subset=[nutrition_per_100_cal_col])
        # pyspark code to fill nulls with 0 in only the current nutrition_per_100_cal_col
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
```

Task-List

Task 4:

Convert the tags column from a string to an array of strings:
Convert the tags column from a string to an array of strings.

Task 5:

Read the second data file: Read the RAW_interaction.csv and join this interaction level file with the recipe level data frame. The resulting data frame should have all the interactions.

Task 6:

Create time based features:
Create features that capture the time passed between one review and the date on which the recipe was submitted.
Use the review_date and the submitted columns after you join the two data files

Solution to Task 4

Complete the code in the following cell

```
In [25]: # Task 04 Cell 1 out of 1


raw_recipes_df = (raw_recipes_df
                 .withColumn('tags',  F.regexp_replace("tags","[\\[\\]\\']",""))
                          # pyspark function to remove symbols like '[' ']' "'" from the strings in the tags column.
                          )
                 .withColumn('tags',  F.split("tags",", ")
                          # pyspark function to split the column using the comma delimiter.
                          ))
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
```

Solution to Task 5

Complete the code in the following cell

```
In [32]: # Task 05 Cell 1 out of 1

interaction_level_df = raw_ratings_df.join(raw_recipes_df,raw_ratings_df.recipe_id == raw_recipes_df.id,"inner")
                                            # add the key on which the join should happen
                                            # mention the type of join expected.
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
```

Test cases for Task 05

```
In [33]: # Code check cell
         # Do not edit cells with assert commands
         # If an error is shown after running this cell, please recheck your code.

         assert (interaction_level_df.count() ,len(interaction_level_df.columns)) == (1132367, 38), "The type of join is incorrect"

         list1 = raw_ratings_df.select('recipe_id').collect()
         list2 = raw_recipes_df.select('id').collect()
         exclusive_set = set(list1)-set(list2)

         assert len(exclusive_set) == 0, "There is a mistake in reading one of the two data files."
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
```

Solution to Task 6

Complete the code in the following cell

In [34]:
```
# Task 06 Cell 1 out of 2

interaction_level_df = (interaction_level_df
                        .withColumn('submitted',F.col("submitted").cast("date") # pyspark function to cast a column to DateType(
                                   )
                        .withColumn('review_date', F.col("review_date").cast("date")# pyspark function to cast a column to DateT
                                   )


                        )
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
```

In [35]:
```
interaction_level_df = (interaction_level_df
                        .withColumn('days_since_submission_on_review_date',F.datediff("review_date","submitted")
                                    # Pyspark function to find the number of days between two dates
                                   )
                        .withColumn('months_since_submission_on_review_date',F.months_between("review_date","submitted")
                                    # Pyspark function to find the number of months between two dates
                                   )
                        .withColumn('years_since_submission_on_review_date',F.months_between("review_date","submitted")/12
                                    # Pyspark function to find the number of months between two dates / 12
                                   )
                         )
```

Task 7:

Processing Numerical Columns (Optional): Convert all numerical columns to categorical columns using the percentile approach to decide the category boundaries. After creating buckets, study the variation of the average rating for each bucket and decide whether or not a particular bucketed column should be kept in the analysis.

Task 8:

Create user-level features (Optional):
1.Create user-level features to capture intrinsic feedback. 2.Create columns such as user_avg_rating, user_avg_n_ratings, user_avg_years_betwn_review_and_submission, user_avg_prep_time_recipes_reviewed, user_avg_n_steps_recipes_reviewed, user_avg_n_ingredients_recipes_reviewed, user_avg_years_betwn_review_and_submission_high_ratings, user_avg_calories_recipes_reviewed, user_avg_total_fat_per_100_cal_recipes_reviewed, user_avg_sugar_per_100_cal_recipes_reviewed, user_avg_sodium_per_100_cal_recipes_reviewed, user_avg_protein_per_100_cal_recipes_reviewed, user_avg_saturated_fat_per_100_cal_recipes_reviewed, user_avg_carbohydrates_per_100_cal_recipes_reviewed, user_avg_prep_time_recipes_reviewed_high_ratings, and user_avg_n_steps_recipes_reviewed_high_ratings. 3.After these columns are created, do a thorough data check. You might have introduced null values to the data during your transformations. You can also do the bucketing exercise on user- level features.

Defining Custom Functions

In [7]:
```python
def get_quantiles(df, col_name, quantiles_list = [0.01, 0.25, 0.5, 0.75, 0.99]):
    """
    Takes a numerical column and returns column values at requested quantiles

    Inputs
    Argument 1: Dataframe
    Argument 2: Name of the column
    Argument 3: A list of quantiles you want to find. Default value [0.01, 0.25, 0.5, 0.75, 0.99]

    Output
    Returns a dictionary with quantiles as keys and column quantile values as values
    """

    # Get min, max and quantile values for given column
    min_val = df.agg(F.min(col_name)).first()[0]
    max_val = df.agg(F.max(col_name)).first()[0]
    quantiles_vals = df.approxQuantile(col_name,
                                       quantiles_list,
                                       0)

    # Store min, quantiles and max in output dict, sequentially
    quantiles_dict = {0.0:min_val}
    quantiles_dict.update(dict(zip(quantiles_list, quantiles_vals)))
    quantiles_dict.update({1.0:max_val})
    return(quantiles_dict)
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
```

```
{0.0: 0, 0.01: 2.0, 0.05: 5.0, 0.25: 20.0, 0.5: 40.0, 0.75: 70.0, 0.95: 310.0, 0.99: 930.0, 1.0: 2147483647}
```

In [19]:
```python
# Capping prep time at 930 minutes

interaction_level_df = (interaction_level_df
                        .withColumn("minutes",
                                    F.when(interaction_level_df["minutes"] > 930, 930)
                                     .otherwise(interaction_level_df["minutes"])))
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
```

In [20]:
```python
# investigating recipes with minutes = 0 -> Look at n_steps for such recipes.

get_column_distribution_summary(df = (interaction_level_df
                                      .filter('minutes == 0')
                                      .withColumn('n_steps_modified', (F.when(interaction_level_df['n_steps'] >= 10, ">= 10")
                                                                        .otherwise(F.lpad(interaction_level_df['n_steps'],2,"0"))
                                col_name = 'n_steps_modified')
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
```

n_steps_modified	avg_rating	stddev_rating	n_ratings	n_recipes
01	4.24	1.0908712114635715	25	12
02	4.4423076923076925	1.0867866867358402	104	28
03	3.989130434782609	1.5750414356065068	184	44
04	4.306358381502889	1.3867374413641125	173	57
05	4.231788079470198	1.356386192466306	302	90
06	4.470703125	1.1463893346523668	512	102
07	4.3447432762836184	1.2875641979464005	409	92
08	4.381995133819951	1.2774085466671234	411	92
09	4.076190476190476	1.5270088873176948	315	86
>= 10	4.240963855421687	1.3768155493871745	2075	491

Task 9:

Create tag-level features (Optional): Extract tags-level features by exploring all the available tags. Create new columns to capture the unique tags and their frequency in the dataset.

In [33]:
```
tags_ratings_summary.sort(F.col("n_user_ratings").asc()).show(5)
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…

+--------------------+---------------+--------------+---------+---------------------+---------------------+
|      individual_tag|avg_user_rating|n_user_ratings|n_recipes| in_percent_recipies|in_percent_interactions|
+--------------------+---------------+--------------+---------+---------------------+---------------------+
|     cranberry-sauce|            5.0|             1|        1|4.340164752654011E-6|   8.876193959039915E-7|
|           pot-roast|            0.0|             1|        1|4.340164752654011E-6|   8.876193959039915E-7|
|   main-dish-seafood|            0.0|             1|        1|4.340164752654011E-6|   8.876193959039915E-7|
|   ham-and-bean-soup|            4.0|             1|        1|4.340164752654011E-6|   8.876193959039915E-7|
|lamb-sheep-main-dish|            0.0|             1|        1|4.340164752654011E-6|   8.876193959039915E-7|
+--------------------+---------------+--------------+---------+---------------------+---------------------+
only showing top 5 rows
```

The above tags are present in 1 recipe in over two hundred thousand. The features we create based on these tags will not teach the model new information. If these tags were one hot encoded, the entire column would be filled with zeros, and only a few rows will have 1s. One hot encoding of these tags is not a good idea. If you come up with an encoding that captures the rarity of these tags, only then can you add these tags to the analysis.

3. Top `n` rated tags

In [34]:
```
tags_ratings_summary.sort(F.col("avg_user_rating").desc()).show(5)
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…

+--------------------+---------------+--------------+---------+---------------------+---------------------+
|      individual_tag|avg_user_rating|n_user_ratings|n_recipes| in_percent_recipies|in_percent_interactions|
+--------------------+---------------+--------------+---------+---------------------+---------------------+
|   side-dishes-beans|            5.0|             2|        2|8.680329505308021E-6|   1.775238791807983E-6|
|             cabbage|            5.0|             1|        1|4.340164752654011E-6|   8.876193959039915E-7|
|heirloom-historic...|            5.0|             3|        2|8.680329505308021E-6|   2.662858187711975E-6|
|middle-eastern-ma...|            5.0|             2|        1|4.340164752654011E-6|   1.775238791807983E-6|
```

Newness Feature Extraction:

Determine the 'newness' of a recipe at the time of review by subtracting the submitted date from the review date. This feature can capture the immediate appeal of new recipes.

User Preferences Analysis:

Identify user preferences by analyzing their past ratings. For example, if a user consistently rates dessert recipes highly, this preference can be used to tailor future recommendations.

Recipe Specifics Characterization:

Develop features that encapsulate specific attributes of a recipe, like its category (e.g., dessert, appetizer), to match it with user preferences.

Tag Processing Priority: Give high priority to processing the 'tags' field, as it contains valuable information on user preferences and recipe characteristics, which are crucial for the recommender system.

Description Column Analysis: Evaluate the potential value of processing the 'description' column while considering the overlap of information with the 'tags' field. Prioritize based on the uniqueness and value of information it adds.

Strategic Documentation: Use a structured document to track and organize the EDA and feature extraction process. Document each field, intended processing, and the features to be extracted, along with their prioritization.

Template Utilization: Leverage template notebooks for EDA and feature extraction, which contain prewritten code and guidelines. Customize or create your features as needed, ensuring the data passes assert checks for consistency and accuracy.