BDM - Assignment 1

Christian Bank Lauridsen (chbl@itu.dk)
GitHub Repository

1 Specific DataFrame Queries

In this section, I am using Spark dataframes to construct the following queries.

1.1 Task 3.1.1

Listing 1: Find the total number of reviews for all businesses

```
1 reviews.count()
```

This returned the number **6990280**, which represents the total number of reviews for all businesses. This is a simple query since all business reviews exist in the `reviews` dataframe. As a result, I just need to count the number of rows in the dataset with the `count()` operation (line 1).

1.2 Task 3.1.2

Listing 2: Find all businesses that have received 5 stars and have been reviewed by 750 or more users. The output should be in the form of a DataFrame of (name, stars, review count)

```
1 five_stars_business = business.filter((business.stars==5) & (business.
      review_count >= 750)) \
2    .select("name", "stars", "review_count")
3
4 five_stars_business.show()
```

For this query, I create a new dataframe called `five_stars_business` (line 1) from the `business` dataframe. I use the `filter` operation to get businesses that fulfill the two requirements (have 5 stars and have more than 750 reviews) using the `and` operator. I only need to use the `business` dataframe as it contains both the `stars` and `review_count` columns. To get the required dataframe output, I use the `select` operation with the columns `name`, `stars`, `review_count` from the `business` dataframe. The resulting dataframe of the query is shown in Table 1, and a following barplot can be seen in Figure 1

Table 1: Top-rated businesses

Name	Stars	Review Count
Blues City Deli	5.0	991
Free Tours By Foot	5.0	769
Carlillos Cocina	5.0	799

1.3 Task 3.1.3

Listing 3: Find influencers who have written more than 750 reviews. And have an average rating higher than 4 stars. The output should be in the form of a Spark DataFrame of user id.

```
1 influencers = users.filter((users.average_stars > 4) & (users.review_count >
      750)).select("user_id")
2
3 influencers.show()
```

For this query I create a new dataframe called `influencers` from the `users` dataframe (line 1). I do this by using the `filter` operator to only capture users with an average rating higher than 4 stars from the `average_stars` column and have more than 750 reviews from the `review_count` column. I then use the `select` operation to only output the `user_id` for the `influencers` dataframe.

The query results in **1028** rows, which is the number of infuencers who have written more than 750 reviews. The resulting dataframe is shown in Table 2, which only shows the top 20 rows.

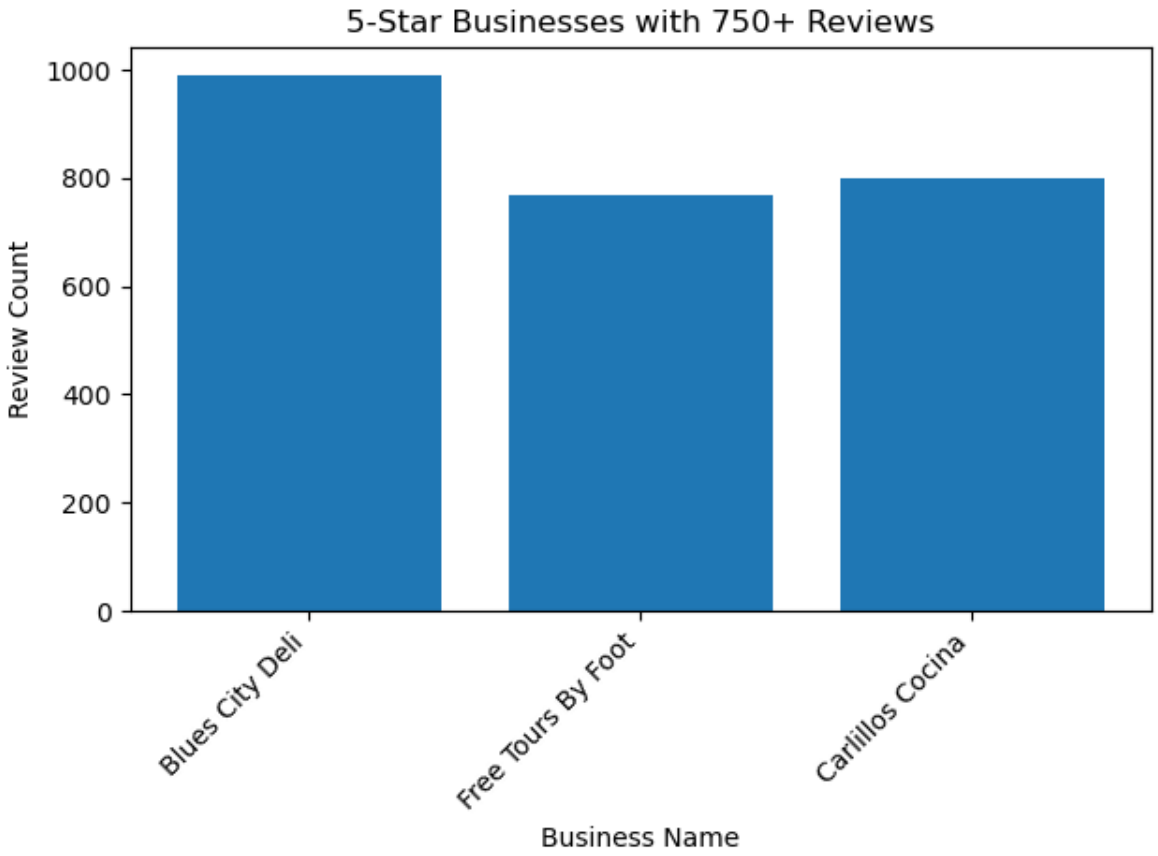

Figure 1: Barplot of businesses with 5 stars and have more than 750 reviews

Table 2: List of User IDs

User ID
MGPQVLsODMm9ZtYQW-g_OA
RgDVC3ZUBqpEe6Y1kPhIpw
VHdY6oG2JPVNjihWhOooAQ
UQFE3BT1rsIYrcDvu_XVow
0G-QF457q_0Z_jKqh6xWiA
ITa3vh5ERI90G_WP4SmGUQ
oW2bCSJ5bIHyRnoZvkHwDg
13f_vtUZEmlzweL91bmVng
UVUIi0q38pAvsfuEEppe0Q
zDBOdWtl2PsNY38IeoE5cQ
7Na1pUcEv3oF_QTRwZ-2iw
MUzkXfPS9JaMgJ907orz0g
YWFbTcVnun8i03XnEU7kVg
Ve0LUwcrzxL7w0RYgY4Aaw
i5Cm53q0pmklQsX8soi3tw
IzJ36jX6V6ky5BtoN-Agng
eTvp_hYnsrI5-ow_sQ31_g
7PHHxsjyk7I7oM6ENxwZZQ
WWnhqRnWWjutMqh-2SzEuQ
0QdwQLVxZpgy9Qb2Qakflw

1.4 Task 3.1.4

Listing 4: Find the businesses names and their average ratings for ones that have been reviewed by more than 5 influencer users.

```
1  review_influencers = reviews.join(influencers, on="user_id", how="inner").
       withColumnRenamed("stars", "review_stars")
2
3  business_review_influencers = business.join(review_influencers, on="
       business_id", how="inner")
4
5  influencer_counts = business_review_influencers.groupBy("business_id", "name")
       .agg(F.countDistinct("user_id").alias("influencer_count"))
6
7  good_businesses = influencer_counts.filter(influencer_counts.influencer_count
       > 5)
8
9  avg_rating = business_review_influencers.groupBy("business_id", "name").agg(F.
       avg("review_stars").alias("avg_rating"))
10
11 result = good_businesses.join(avg_rating, on=["business_id", "name"], how="
       inner")
12
13 result.select("name", "avg_rating").show()
```

This query is a bit more challenging as it requires using the `join` operation between different dataframes. I first join the `influencers` dataframe from the previous task together with the `reviews` dataframe (line 1). This comes with a problem as both dataframes contain the same column name `stars`, but they describe different ratings. To fix this problem, I use the `withColumnRenamed` to rename the `stars` column to `review_stars` from the `reviews` dataframe, to make a distinction between the two columns from the datasets.

This gives me a dataframe of reviews made by only influencers called `review_influencers`.

To find the businesses, I also use the `join` operation on `business` and `review_influencers` (line 3). This gives me the `business_review_influencers` dataframe, which only contains businesses that have been reviewed by influencers.

To get the number of reviews by distinct influencers for each business, I create another dataframe called `influencer_counts` (line 5) from the `business_review_influencers` dataframe. Here I use the `groupBy` operation to group businesses together by their `business_id` and their `name`. For each business, I create a new column called `influencer_count` by making use of the `agg` operation to aggregate all distinct influencers together by using the function `countDistinct` on `user_id`. To give this new column a name, I use the `alias` function.

Now that I have all the businesses together with their number of reviews by influencers, I can continue to keep the businesses with more than 5 influencers. I do this by creating a new dataframe called `good_businesses`, by using the `filter` operation (line 7).

Now I need to compute the average rating per business from influencer reviews only. To do this, I create a new dataframe called `avg_rating`, by using the `business_review_influencers` dataframe (line 9). I again use the `groupBy` operation on the `business_id` and their `name`. I then create a new column `avg_rating` which is the average rating from the `review_stars` by using the `avg` function.

Finally, to combine everything together in one dataframe called `result`, I do a `join` operation on the two dataframes `good_businesses` and `avg_rating` (line 11). To get the required dataframe output, I use the `select` operation (line 13), and only show the `name` and `avg_rating` columns.

This results in **2298** rows, which is the total number of businesses which have been reviewed by more than 5 influencers. And the dataframe showcasing the top 20 rows is shown in Table 3.

Table 3: Average ratings by businesses that have been reviewed by more than 5 influencers

Name	Average Rating
St. James Cheese Company — Uptown	4.5
American Sardine Bar	4.6667
Saba	5.0
Grille 54	4.1111
Dressel's Public House	4.1429
Drunken Fish – Central West End	3.3333
IKEA	3.8571
Buffa's Bar & Restaurant	3.6250
Audubon Zoo	4.8182
Tropical Isle	4.0
Listening Room Cafe	4.5714
Lucky Beaver Bar & Burger	4.1111
Westfield Brandon	3.1667
Ho Ho Choy	4.2222
St Pete Pier	4.6
Roy's Restaurant	4.5455
Charley's Steak House	4.0
Victory Field	4.5
Park Avenue Coffee – Lafayette Square	4.5833
The Jazz Playhouse	4.4286

1.5 Task 3.1.5

Listing 5: Find an ordered list of users based on the average star counts they have given in all their reviews

```
1 from pyspark.sql.functions import col
2 avg_star_users = users.select("average_stars").orderBy(col("average_stars").
      desc())
3
4 avg_star_users.show()
```

For this query I create a new dataframe called `avg_star_users` from the `users` dataframe (line 2). Since `users` already contains a column called `average_stars`, I only need to use the `orderBy` operation to get a sorted list of users based on the average star counts in all their reviews. I sort the users in descending order by using the `desc` function. To use the `orderBy` operation on a column, I import the `col` library from `pyspark.sql.functions` (line 1).

This results in **1987897** rows, which is the number of users and their average star counts. The ordered dataframe is shown in Table 4, which only show the top 20 rows.

Table 4: Average star ratings

Average Stars
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0
5.0

2 Authenticity Study

2.1 Data Exploration 3.2.1

2.1.1 Reviews with "authentic" and not "inauthentic" words

Listing 6: What is the percentage of reviews that contain a variant of the word "authentic" and not "inauthentic"

```
1  AUTHENTIC_WORDS = ["authentic", "original", "real", "legitimate", "legit", "
       credible", "genuine", "valid"]
2  BAD_WORDS = ["inauthentic", "illegitimate", "ungenuine", "unreal", "invalid",
       unoriginal"]
3
4  authentic_pattern = "(?i)(" + "|".join(AUTHENTIC_WORDS) + ")"
5  bad_pattern = "(?i)(" + "|".join(BAD_WORDS) + ")"
6
7  reviews_with_authentic = reviews.filter(col("text").rlike(authentic_pattern) &
       (~col("text").rlike(bad_pattern)))
8
9  total_reviews = reviews.count()
10 authentic_reviews = reviews_with_authentic.count()
11
12 percentage = (authentic_reviews / total_reviews) * 100
13 print(f"Percentage of reviews that contain a variant of the word \"authentic
       \": {percentage:.2f}%")
```

The following code returns **24.03%**, which is the percentage of reviews that contain a variant of the word "authentic" and not "inauthentic". To get the percentage of authenticity, I start by defining two lists called `AUTHENTIC_WORDS` and `BAD_WORDS` (lines 1-2), which describe words that are associated with the words "authentic" and "inauthentic".

I then create two regex patterns for each list for searching for authentic and inauthentic words called `authentic_pattern` and `bad_pattern` (lines 4-5). The patterns first make all words case insensitive by using the `(?i) pattern`. The following `"|".join()` pattern creates an `OR` pattern to search for the words from the authenticity lists (example checking for either authentic, original, or real).

Now I create a new dataframe called `reviews_with_authentic` (line 7). Here I use the `filter` operation on the `text` column from the `reviews` dataframe. I use the `rlike` function (like with regex) to do filtering with a regex by keeping only reviews with authentic words from the `AUTHENTIC_WORDS` list by using the `authentic_pattern` and ignoring words from the `BAD_WORDS` list by using the `bad_pattern`, removing reviews with inauthentic words

Finally, to get the percentage, I count the total number of reviews (`total_reviews`) by getting the number of rows from the `reviews` dataframe (line 9). I then count the number of authentic reviews (`authentic_reviews`) from the number of rows in the `reviews_with_authentic` dataframe (line 10). I then calculate and print the percentage (lines 12-13).

2.1.2 Count the number of reviews containing the word "genuine" grouped by business type

Listing 7: How many reviews contain the string "genuine" grouped by businesses type?

```
1  review_with_genuine = reviews.filter(col("text").rlike("(?i)genuine"))\
2      .select("business_id")
3
4  reduced_business = business.select("business_id", "categories")\
5      .filter(F.col("categories").isNotNull())\
6      .withColumn("business_type", F.explode(F.split(F.col("categories"), ", "))
           )
```

```
7
8  business_review_with_genuine = review_with_genuine.join(reduced_business, on="
       business_id", how="inner")
9
10 genuine_result = business_review_with_genuine.groupBy("business_type").count()
       .cache()
11
12 total_review_with_genuine = business_review_with_genuine.count()
13 print(f"Total reviews with 'genuine': {total_review_with_genuine}")
```

This results in **181107** rows, which is the total number of reviews containing the string "genuine" that is grouped by business type. The resulting dataframe can be seen in Table 5, only showing the top 20 rows.

To get the reviews only containing the word "genuine", I create a new dataframe called `review_with_genuine` from the `review` dataframe. I again use the `filter` operation on the `text` column by filtering with a regex pattern that filters for the word "genuine"(line 1).

Next, I am using the `business` dataframe to use the `categories` column to get business types (line 3). I use the `select` operation to reduce the size of the dataframe, to only use the columns `business_id` and `categories`. The `categories` column contains a single string containing multiple categories (example "Chinese, Restaurant, Japanese"), which makes grouping problematic.

To fix this problem, I use the `withColumn` operation together with the `explode` and `split` functions. These functions split the `categories` string into an array of individual category entries and expand them into multiple rows (one per category). For example, the row "Chinese, Restaurant, Japanese" becomes three separate rows, each with one category. After this transformation, the column `categories` is renamed to `business_type`.

Next, I do a join operation between the two dataframes to combine them to do the `groupBy` operation on the `business_type` and get the total count of reviews containing the word "genuine" (lines 9-12).

Table 5: The number of businesses containing the string "genuine" grouped by business type

Business Type	Count
Historical Tours	121
Paddleboarding	15
Dermatologists	82
Pet Photography	10
Day Spas	616
Hobby Shops	100
Reiki	73
Bubble Tea	142
Child Care & Day Care	107
Tanning	141
Bed & Breakfast	97
Boxing	31
Handyman	30
Salad	1391
Beauty & Spas	2978
Beer Bar	473
Education	360
Arcades	56
Art Classes	28
Bookstores	85
Falafel	41
Sports Clubs	55

2.1.3 Count the number of reviews containing the word "legitimate" or "illegitimate" grouped by cuisines

Listing 8: How many reviews contain the string "legitimate" grouped by type of cuisine? What about "illegitimate"?

```
resturants = business.filter(business.categories.rlike("(?i)restaurants"))

ASIAN_CUSINE_LIST = ["Chinese", "Japanese", "Korean", "Thai", "Vietnamese", "Indian", "Filipino", "Asian"]
asian_pattern = "(?i)(" + "|".join(ASIAN_CUSINE_LIST) + ")"
asian_cuisines = resturants.filter(col("categories").rlike(asian_pattern))

EUROPEAN_CUISINE_LIST = ["Italian", "French", "Spanish", "Greek", "German", "British", "Portuguese"]
european_pattern = "(?i)(" + "|".join(EUROPEAN_CUISINE_LIST) + ")"
european_cuisines = resturants.filter(col("categories").rlike(european_pattern))

AMERICAN_CUISINE_LIST = ["American", "Mexican", "Brazilian", "Argentinian", "Colombian"]
american_pattern = "(?i)(" + "|".join(AMERICAN_CUISINE_LIST) + ")"
american_cuisines = resturants.filter(col("categories").rlike(american_pattern))

all_cuisines = asian_cuisines.union(european_cuisines).union(american_cuisines)

reviews_with_legitimate = reviews.filter(col("text").rlike("(?i)legitimate"))
reduced_reviews_with_legitimate = reviews_with_legitimate.select("business_id", "text")

```

```
20 reviews_with_illegitimate = reviews.filter(col("text").rlike("(?i)illegitimate
       "))
21 reduced_reviews_with_illegitimate = reviews_with_illegitimate.select("
       business_id", "text")
22
23 business_review_with_legitimate = reduced_reviews_with_legitimate.join(
       all_cuisines, on="business_id", how="inner").select("business_id", "
       cuisine_type")
24 business_review_with_illegitimate = reduced_reviews_with_illegitimate.join(
       all_cuisines, on="business_id", how="inner").select("business_id", "
       cuisine_type")
25
26 business_review_with_legitimate = business_review_with_legitimate.groupBy("
       cuisine_type").agg(F.count("*").alias("count")).orderBy(F.desc("count"))
27 business_review_with_legitimate.show()
28
29 business_review_with_illegitimate = business_review_with_illegitimate.groupBy(
       "cuisine_type").agg(F.count("*").alias("count")).orderBy(F.desc("count"))
30 business_review_with_illegitimate.show()
```

The total number of reviews containing the string "legitimate" is **2471**, and its dataframe is shown in Table 6, and for "illegitimate" is **23** and its dataframe is shown in Table 7.

This task requires a grouping of cuisines. For this, I start by creating a dataframe called `restaurants` only containing businesses with the category restaurant (line 1).

Next, from lines 3-15, I use the same technique from Section 2.1.1, which finds businesses associated to different types of cuisines: Asian, European, and American. On line 15, I combine all the cuisines in one dataframe called `all_cuisines`. The reason I use multiple cuisines is that it is something I will use for the hypothesis testing task in Section 2.2.

Next, I create two dataframes from the `reviews` dataframe: `reviews_with_legitimate` and `reviews_with_illegitimate`. Each dataframe is generated by applying the `filter` operation on the `text` column using the `rlike` function to match reviews containing the words "legitimate" and "illegitimate", respectively (lines 17-21). To optimise the performance, I reduce each dataframe to only use `business_id` and `text` columns before further processing.

Then I perform a `join` operation between the filtered review dataframes and the `all_cusines` dataframe on the `business_id` column (lines 23-24). The join associates each review with its corresponding cuisine type.

Finally, I group the resulting data by `cuisine_type`, count the total number of reviews for each cuisine, and order the results in descending order of the review count (lines 26-30). This is done separately for both legitimate and illegitimate reviews.

Table 6: Cuisine types and their total number of reviews containing the string "legitimate"

Cuisine Type	Count
American (New)	560
American (Traditional)	544
Mexican	295
Italian	271
Chinese	159
Japanese	151
Asian Fusion	123
Thai	86
Vietnamese	65
Latin American	61
French	60
Korean	50
Greek	42
Indian	35
Spanish	19
British	13
German	12
New Mexican Cuisine	11
Brazilian	9
Filipino	7
Pan Asian	4
Colombian	4
Japanese Curry	1

Table 7: Cuisine types and their total number of reviews containing the string "illegitimate"

Cuisine Type	Count
American (Traditional)	6
Mexican	5
American (New)	4
Italian	3
Vietnamese	2
British	1
Korean	1
Latin American	1

2.1.4 Explore the difference in the amount of authentic language used in the different areas

Listing 9: Is there a difference in the amount of authenticity language used in the different areas?

```
AUTHENTIC_WORDS = ["authentic", "original", "real", "legitimate", "legit", "
    credible", "genuine", "valid"]
authentic_pattern = "(?i)(" + "|".join(AUTHENTIC_WORDS) + ")"

reviews_flagged = reviews.join(business.select("business_id", "state", "city")
    , "business_id").withColumn("has_authenticity", F.col("text").rlike(
    authentic_pattern).cast("int"))

cube_results = reviews_flagged.cube("state", "city").agg(F.sum("
    has_authenticity").alias("authenticity_count"), F.count("*").alias("
    total_count")).withColumn("percentage", (F.col("authenticity_count") / F.
    col("total_count") * 100)).orderBy(F.desc("percentage"))
```

```
7
8 cube_filtered = cube_results.filter(F.col("state").isNotNull() & F.col("city")
      .isNotNull())
9 cube_filtered.show()
```

The resulting dataframe can be seen in Table 8, which shows the top 20 rows. It shows that in some areas the percentage of used authentic language is higher than in other areas. But for the majority, the percentage of authentic language used is around 50-60%, which means that around 50% of reviews use authentic language.

For this query, I create a new dataframe called `reviews_flagged` which flags reviews containing authentic words. This is done by joining each `review` with its corresponding `business` to get location data (`state` and `city`). Then I add a new column `has_authenticity`, which is set to 1 if the review contains any authenticity words, otherwise 0 (line 4).

I then create a new dataframe `cube_results`, which does a cube aggregation over `state` and `city` (line 6). The `cube` computes all possible combinations of the grouping, which is four. For example, an individual city within a state (state, city), the total reviews for each state (state, null), totals for each city (null, city), or the overall total for all reviews (null, null). For each group, the total sum of authentic language (`authenticity_count`) and the total number of reviews (`total_count`) is counted in that group. I use these counts to create a new column to calculate `percentage` of reviews in each area mentioning authenticity words.

Because the cube operation also produces groupings with null values, I use a final filter away null values, and only get data having both `state` and `city` (line 8).

Table 8: Authenticity counts and percentages by city and state

State	City	Authenticity Count	Total Count	Percentage (%)
PA	East Greenville	4	5	80.0
NV	Sparks	4	6	66.67
PA	New Britian	4	6	66.67
FL	Bradenton Beach	5	8	62.5
NJ	Pittsgrove Township	3	5	60.0
NV	RENO AP	3	5	60.0
PA	MIDDLE CITY WEST	3	5	60.0
NJ	Vineland	3	5	60.0
FL	HUdson	10	17	58.82
LA	St Rose	7	12	58.33
IN	Franklin	4	7	57.14
FL	Lutz fl	5	9	55.56
HI	Lula Lula	10	19	52.63
PA	Oakford	11	21	52.38
IL	Fairmont City	70	138	50.72
MT	Kalispell	3	6	50.0
PA	W. Chester	3	6	50.0
PA	West Philadelphia	3	6	50.0
AB	Downtown	3	6	50.0
AB	NW Edmonton	5	10	50.0
PA	Tylersport	4	8	50.0
PA	Havertown, PA	3	6	50.0
PA	PLYMOUTH MTNG	5	12	41.67

2.2 Hypothesis Testing 3.2.2

For hypothesis testing, I use MLlib with the methods provided by `Statistics` to run a Pearson's chi-squared test. I use this test to determine whether there is statistical evidence that the use of authenticity language or negative words differs by cuisine groups (South American, Asian, European) in user reviews.

I define two hypotheses:

- H_0: The use of authenticity language is independent of the cuisine type. In other words, statistically, users are equally likely to mention "authenticity" regardless of whether they are reviewing a South American, Asian, or European food.
- H_1: The use of negative language is independent of the cuisine type. The frequency of negative reviews is statistically consistent across all three cuisine types. No specific cuisine is more likely to receive negative reviews than the others.

If the p-value of the test is less than 0.05, we can reject the null hypothesis, meaning there is a significant difference between the cuisines. If the p-values are greater than 0.05, we keep these null hypotheses.

Listing 10: Preprocessing of data

```
from pyspark.mllib.linalg import Matrices
from pyspark.mllib.stat import Statistics

resturants_exploded = resturants.withColumn("cuisine_type", F.explode(F.split(F.col("categories"), ", ")))\
    .select("business_id", "name", "cuisine_type")

# Cuisines (South American, Asian, European)
south_american_cuisine = ["Brazilian", "Argentinian", "Peruvian", "Colombian", "Chilean", "Cuban", "Venezuelan", "Guatemalan", "Costa Rican"]
asian_cuisine = ["Chinese", "Japanese", "Korean", "Thai", "Vietnamese", "Indian", "Filipino", "Asian"]
european_cuisine = ["Italian", "French", "Spanish", "Greek", "German", "British", "Portuguese"]

south_american_pattern = "(?i)(" + "|".join(south_american_cuisine) + ")"
asian_pattern = "(?i)(" + "|".join(asian_cuisine) + ")"
european_pattern = "(?i)(" + "|".join(european_cuisine) + ")"

# Join reviews to get only relevant cuisine types
reviews_with_cuisine = reviews.join(resturants_exploded.select("business_id", "cuisine_type"), on="business_id", how="inner") \
    .withColumn("cuisine_group",
                F.when(F.col("cuisine_type").rlike(south_american_pattern), "South American")
                 .when(F.col("cuisine_type").rlike(asian_pattern), "Asian")
                 .when(F.col("cuisine_type").rlike(european_pattern), "European")
                 .otherwise("Other")) \
                 .filter(F.col("cuisine_group") != "Other")

reviews_with_cuisine_reduces = reviews_with_cuisine.select("business_id", "text", "cuisine_group")

# Authenticity and negative word flags
reviews_contain_authenticity_negativity = reviews_with_cuisine_reduces \
    .withColumn("has_authenticity", F.col("text").rlike(authentic_pattern)) \
    .withColumn("has_negative", F.col("text").rlike(bad_pattern))

# Count authnticity and negative words by cuisine group and total number of reviews
authenticity_negativity_counts = reviews_contain_authenticity_negativity.groupBy("cuisine_group") \
    .agg(F.sum(F.col("has_authenticity").cast("int")).alias("authenticity_count"),
         F.sum(F.col("has_negative").cast("int")).alias("negative_count"),
         F.count("*").alias("total_count")) \
```

```
37      .orderBy("cuisine_group")
38
39  authenticity_negativity_counts.cache()
40  authenticity_negativity_counts.show()
```

I start by processing the data to get cuisines, with a similar approach in Section 2.1.3. On lines 17-25, I filter away cuisines that are not South American, Asian or European, to only have reviews from these cuisine types.

Next on lines 28-30, I flag each review to check if they mention authenticity or negative language with true or false values.

Then on lines 33-37, I group cuisines and convert the boolean values to numerical values (1 or 0) and aggregate the sums rof eviews mentioning authenticity, negative words, and the total reviews per cuisine type.

Listing 11: Hypothesis testing with chi-square test

```
1   # Authenticity by cuisine
2   counts = authenticity_negativity_counts.collect()
3
4   values = []
5   for row in counts:
6       # Column-major order for Matrices.dense: [authenticity1, authenticity2,
            authenticity3, no_auth1, no_auth2, no_auth3]
7       values.append(row['authenticity_count'])
8   values_no_auth = [row['total_count'] - row['authenticity_count'] for row in
        counts]
9
10  # Combine for column-major
11  matrix_values = values + values_no_auth
12
13  # Create contingency matrix: 3 cuisines x 2 columns (authenticity / no
        authenticity)
14  contingency_matrix = Matrices.dense(3, 2, matrix_values)
15
16  # Perform chi-square test
17  chi_result = Statistics.chiSqTest(contingency_matrix)
18  print("Authenticity by cuisine:\n")
19  print(chi_result)
20  print("-----------------------------------")
21
22  # Negativity by cuisine
23  neg_values = [row['negative_count'] for row in counts]
24  neg_no_values = [row['total_count'] - row['negative_count'] for row in counts]
25
26  matrix_values_neg = neg_values + neg_no_values
27  neg_matrix = Matrices.dense(3, 2, matrix_values_neg)
28
29  chi_result_neg = Statistics.chiSqTest(neg_matrix)
30  print("Negativity by cuisine:\n")
31  print(chi_result_neg)
32  print("-----------------------------------")
```

From lines 2 to 14, I build the contingency matrix (3x2), which is used for the chi-square test that checks whether the number of authenticity words differs significantly across cuisines. The matrix contains the 3 cuisines and the 2 counts (authenticity and no authenticity). I use the same logic for checking the frequency of negative words differs across cuisines (23-27).

Next on lines 17 and 29, the chi-square tests are performed to examine whether there is a relationship between cuisine groups and the use of authenticity or negative language in reviews.

When running the tests, both resulted in very low p-values (close to 0.0), indicating that both null hypotheses H_0 and H_1 can be rejected.

Therefore, there is statistically significant dependence between cuisine type and the usage of both authentic and negative language. Meaning that the use of authentic language and negative words in reviews differs significantly across cuisine groups.

Listing 12: Percentages of reviews with authenticity and negative words

```
authenticity_negativity_percent = authenticity_negativity_counts \
    .withColumn("authenticity_pct", F.round(F.col("authenticity_count") / F.
        col("total_count") * 100, 2)) \
    .withColumn("negative_pct", F.round(F.col("negative_count") / F.col("
        total_count") * 100, 4)) \
    .orderBy("cuisine_group")

authenticity_negativity_percent.show()
```

However, if we look at the practical significance in percentages, which is calculated in the code above, gives the result shown in Table 9. Which shows that there is no huge difference in the percentages in authenticity and negative language. We see that the percentage of authentic language is between 26-28%, and does not showcase a significant difference by the cuisine groups, which we also see for negative language, which is around 0.1%. Although the statistical test indicates a significant difference in the relationship between cuisine type and language, the percentages show that the difference is not necessarily large. By looking at the percentages, the results show that South American cuisines use 28.52% authenticity language, Asian cuisines use 27.19%, and European cuisines use 26.12%.

For percentages of negative words, the results show that European and South American cuisines use 0.13% and Asian cuisines use 0.11%.

Something we should also be aware of is that there are a lot more reviews for Asian and European cuisines, that has 993.635 and 683.667 reviews compared to South American cuisines, which have 68.490.

Finally, while the difference is statistically significant (likely due to the large sample size), the practical difference is minimal. Authenticity language appears to be a consistent feature across all three cuisine types, rather than a unique identifier for one specific cuisine type.

Table 9: Percentages of authentic and negative language for different cuisine types

Cuisine Type	Authenticity Count	Negative Count	Total Count	Authenticity (%)	Negativity (%)
Asian	270,157	1,100	993,635	27.19	0.1107
European	178,603	931	683,667	26.12	0.1362
South American	19,534	91	68,490	28.52	0.1329

3 Rating Prediction

For this task, I have to create an ML model that predicts the rating of a certain restaurant given a user review.

3.1 Classification vs. Regression

Predicting a star rating (1-5) can be approached as either a regression or a classification problem.

A regression approach would treat the rating as a continuous variable. The **pro** of this method is that it preserves the ordinality oof data (e.g., predicting 3.8 is closer to the true value of 4 than predicting 1). However, a **con** is that it assumes aa linear distance between ratings. Statistically, the psychological difference between a 1 star and a 2 star rating may not be the same as the difference between a 4 star and a 5 star rating.

I have chosen a classification approach to treat the ratings as discrete categoories. The **pro** of this method is that it simplifies the decision boundary, focusing on the distinct groups (positive, neutral, and negative), rather than exact numerical precisin. The **con** is the loss of granularity, and the model will treat errors equally, failing to recognize that predicting "High" for a "Medium" review is better than predicting "Low".

I approach this as a multiclass classification problem by grouping ratings into three categories:

- High rating (4-5 stars)
- Medium rating (2-3 stars)
- Low rating (1 star)

Since all ratings are integers from 1 to 5, this simplifies the prediction task. A challenge is that different users may express or use similar words associated with different ratings. For example, users may use the word "great" differently and give either 4 or 5 stars. Using the grouping of rating types (high, medium, low), the model needs to distinguish between three classes instead of five. As a **con** of this approach my model lose the ability to distinguish between a "good" and a "perfect" experience. But with this approach, I reduce the complexity and avoid the difficulty of predicting the difference between a 2 vs. 3-star rating or a 4 vs. 5-star rating.

3.2 Loading the data

Listing 13: Step 1 - Load data of reviews from resturants only

```
resturants = business.filter(business.categories.rlike("(?i)restaurant"))
resturants = resturants.select("business_id")

reviews_resturants = reviews.join(resturants, on="business_id", how="inner")\
    .select("review_id", "text", "stars")
```

I start by loading the `business` and `review` data and joining them together. I use the `filter` operation to only get businesses that are restaurants. Then I reduce the size of the `business` dataframe to only contain the `business_id` column. Finally I create the dataframe `reviews_resturants` that only contain reviews for restaurants,

3.3 Create groupings of rating types

Listing 14: Step 2 - Create new column "rating_type" with the three classes based on the stars column

```
reviews_resturants = reviews_resturants.withColumn(
    "rating_type",
    when(reviews_resturants.stars >= 4, 2)\
```

```
    .when((reviews_resturants.stars >= 2) & (reviews_resturants.stars <= 3),
        1) \
    .otherwise(0)
)
reviews_resturants = reviews_resturants.withColumn("rating_type", F.col("
    rating_type").cast(DoubleType())) # Cast to double for MLlib

reviews_resturants.show()
```

In this step I categorizes a review into the three rating types represented as numerical values (high: 2, medium: 1, low: 0), and adds it as a new column `rating_type`. I cast these value to doubles, because MLlib needs this format.

3.4 Data cleaning

Listing 15: Step 3 - Clean data

```
reviews_resturants = reviews_resturants.withColumn("text", F.lower(F.
    regexp_replace("text", r"[^a-zA-Z0-9\s]", "")))
```

In this step I make sure to clean the `text` column data such that all words are lowercase, remove punctuation and special characters. This ensures that to semantically identical words are represented consistently. Example "Great" or "GREAT" becomes two different tokens, and lowercasing them will merge them into one token "great".

3.5 Setup training and test data

Listing 16: Step 4 - Split data into training and testing datasets

```
train_data, test_data = reviews_resturants.randomSplit([0.8, 0.2], seed=42)
```

I split my data into a training set and a testing set. The training set represents 80% and the test set represents 20% of the data.

3.6 Balance dataset

Listing 17: Step 5 - Balance train data

```
target_per_class = 30000
seed = 42

# Get class counts
class_counts = dict(train_data.groupBy("rating_type").count().collect())
print("Original train class distribution:", class_counts)

balanced_parts = []
for label, cnt in class_counts.items():
    df_label = train_data.filter(F.col("rating_type") == label)
    # Downsample
    frac = target_per_class / float(cnt)
    sampled = df_label.sample(withReplacement=False, fraction=frac, seed=seed)
    balanced_parts.append(sampled)

train_balanced = reduce(lambda a, b: a.union(b), balanced_parts).cache()
train_balanced.groupBy("rating_type").count().orderBy("rating_type").show()
```

One problem is overfitting, and an example of this is the original training set, which can be seen in Figure 2, which has a distribution of rating types **0.0**: 453586, **1.0**: 757791, **2.0**: 2566563. This means that training my model on this training set will become biased towards higher rated reviews.

Figure 2: Original distribution of the training set

To fix this, I balance my training set to have a similar amount of reviews representing each rating type. I sample the reviews into rating type groups to a target size of 30000. I do downsampling if the group count is larger than the target size, it randomly selects a fraction of rows. This result in a balanced dataset **0.0**: 30050, **1.0**: 30171, **2.0**: 30227, and is shown in Figure 3

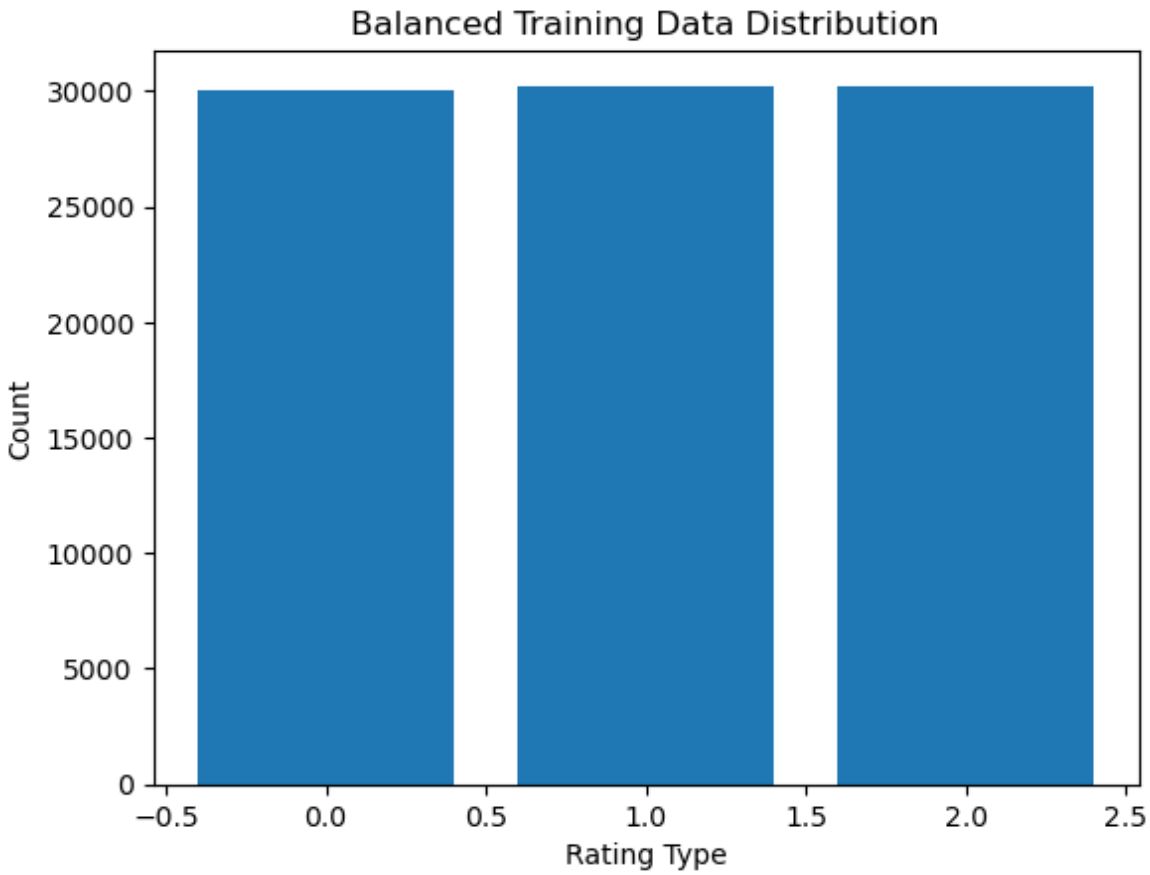

Figure 3: Balanced dataset of the training set

3.7 Feature extraction with TF-IDF

Listing 18: Step 6 - Feature extraction

```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
remover = StopWordsRemover(inputCol="words", outputCol="filtered_words")
hashingTF = HashingTF(inputCol="filtered_words", outputCol="raw_features",
    numFeatures=10000)
idf = IDF(inputCol="raw_features", outputCol="features")
lr = LogisticRegression(featuresCol="features", labelCol="rating_type",
    maxIter=10)

pipeline_tfidf = Pipeline(stages=[tokenizer, remover, hashingTF, idf, lr])
model_tfidf = pipeline_tfidf.fit(train_balanced)
prediction_tfidf = model_tfidf.transform(test_data)
```

For feature extraction I use TF-IDF (Term Frequency-Inverse Document Frequency), that captures the importance of words accross reviews. (1) I start by tokenizing the review text and split the text into individual words (tokens), and output them in a new column `words`. (2) Then I remove stopwords like "the", "and", or "is" from the `words` column, which are not usefull for classification using the `StopWordsRemover`. This output is stored in a new column `filtered_words`. (3) To convert these tokens of words into numerical values, I use the `HashingTF` function, to convert words into a fixed length feature vector. Setting `numFeatures` to 10000, means that each word in `filtered_words` is hashed into one of 10000 positions. This output is stored in the column `raw_features` This is efficient, because it avoids creating a very large dictionary. (4) The `IDF` adjust the term frequency (TF) from the `raw_features` column, which will weigh words according to their importance in a review. Words that

are very common gets downweighted, because they are less relevant, and those that appear more rarely will weigh more. This produces a new column `features`, which are the numerical representations of the review texts. (5) The `LogisticRegression` is a classification model that predicts the `rating_type` from the `features`.

I use `Pipeline` to chain the above stages together, which I use fit my model with the balanced training set. Finally I apply the model on my test data.

3.8 Evaluation

Listing 19: Step 7 - Evaluate the model

```
evaluator = MulticlassClassificationEvaluator(
    labelCol="rating_type",
    predictionCol="prediction",
    metricName="accuracy"
)

acc_tfidf = evaluator.evaluate(prediction_tfidf)
print(f"TF-IDF Accuracy: {acc_tfidf:.2f}")
```

To evaluate my model I use the `MulticlassClassificationEvaluator` which compares the accuracy between the original rating types to the predicted ratings. The evaluated accuracy score is **0.79**, and without a balanced dataset the accuracy score is 0.56.

3.9 Validation

Listing 20: Step 8 - Validation

```
example_reviews = [
    Row(text="The food was incredible and the staff were very friendly"),
    Row(text="Horrible service, my order was wrong and took too long"),
    Row(text="It was okay, nothing special but not terrible"),
    Row(text="Fantastic pizza and quick delivery!"),
    Row(text="The soup was cold and bland"),
]

example_df = spark.createDataFrame(example_reviews)

predictions_example = model_tfidf.transform(example_df)
predictions_example.select("text", "prediction").show(truncate=False)
```

To validate my model, I tried testing it on new reviews, which shows that my model is works well. The result can be seen in Table 10.

Table 10: Model prediction on new data.

text	**prediction**
The food was incredible and the staff were very friendly	2.0
Horrible service, my order was wrong and took too long	0.0
It was okay, nothing special but not terrible	1.0
Fantastic pizza and quick delivery!	2.0
The soup was cold and bland	0.0

3.10 Authenticity as a feature

A relevant consideration for improvements is incorporating authenticity detection could improve the robustness of my model by filtering out ”fake” reviews or spam, that could be deceptive. My model

relies on TF-IDF to capture sentiments and opinions, assuming all reviews are genuine, and risk training on noise. Identifying these and filtering them out, could help ensure the model learns from valid reviews.