University of Sheffield

Skill Distribution Across Job Descriptions: Trends and Analysis

Registration number: 220201791

Word Count: 3270

Setup

Dataset:

The Skill2Vec 50K.csv.gz dataset is a comprehensive resource that compiles data on skills that were taken from a wide range of job postings. It has enormous promise for a range of data-driven applications, such as workforce development plans, skill recommendation systems, and job market analysis. Users can effectively handle and analyze the Skill2Vec dataset by using PySpark, a distributed computing framework, by following the setup instructions provided in this report.

The O*NET dataset offers statistics on jobs, including information on technology and skills. ONET-SOC Code, Example, Commodity Code, Commodity Title, Hot Technology, and In Demand are a few of the columns that are included. Each row corresponds to a particular piece of software or expertise related to a job code. Job titles are identified by the ONET-SOC Code, and specific examples of software or skills needed for each position are shown in the Example column. The Commodity Code and Title columns group abilities into categories, and the Hot Technology and In Demand columns show which technologies are currently popular and in demand.

PySpark and Databricks:

Apache Spark, a potent open-source distributed computing engine, forms the foundation for Databricks, a unified data analytics platform. Data engineers, data scientists, and analysts may handle, examine, and get insights from huge datasets in this collaborative and scalable environment.

In order to facilitate data processing and analysis, Databricks primarily makes use of PySpark, the Python library for Spark. Users can interact with Spark's distributed computing engine by writing Python code using PySpark's high-level API. With PySpark, users may carry out a number of data operations on huge datasets, including data import, transformation, aggregation, and machine learning.

Databricks Instance:

A cloud-based virtual machine created expressly for data analytics and collaboration is called a Databricks instance. It offers users a strong environment for working together and exploring and analyzing data. Users have access to distributed computing capabilities for the effective processing of massive datasets thanks to PySpark's integration with Databricks. The variable computing resources provided by Databricks instances enable customers to assign CPU, memory, and storage according to their own requirements.

Creating an Instance:

1. Log in to the Databricks platform using your username and password.
2. Navigate to Databricks workspace and click on "New Workspace" and give the Instance a Name.
3. Based on your computing needs, such as CPU, memory, and storage, choose the instance type.
4. After looking over the configuration information, click "Create" or "Launch" to start the creation process.

Databricks Notebook:

A vital component of the Databricks platform is a Databricks notebook, which offers a collaborative, interactive workspace for code execution, data analysis, and documentation. Code cells provide compatibility for a number of programming languages, allowing users to take advantage of robust tools like PySpark for complex data processing. Users can add structured text, justifications, and directions to Markdown cells, giving the code and analysis some context.

Creating a Notebook:

1. Go to the Databricks workspace.
2. Select the folder where you want to create the notebook by clicking on it or the "Home" button.
3. The "Notebook" option will appear when you click the "Create" button.
4. From the drop-down menu, select "Python" as the programming language.
5. You can either choose "New Cluster" to build a new cluster or choose the cluster on which you want to execute the notebook.
6. To create the notebook, click the "Create" button.

Data Pre-processing

Importing Libraries and Functions

```python
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
from pyspark.sql.functions import explode
from pyspark.sql import functions as func
```

The PySpark library, which offers the necessary modules and classes for distributed data processing and analysis, is imported using the **import pyspark** command.

The SparkSession class, which serves as the starting point for dealing with structured data in Spark, is imported using the **pyspark.sql import SparkSession** statement.

A SparkSession object named spark is initialized by the statement **SparkSession.builder.getOrCreate()**. The application name, master URL, and other SparkSession configuration parameters are set via the builder API. A SparkSession will either be created or, if one already exists, obtained using the getOrCreate() method.

The explode function is imported using **pyspark.sql.functions import** statement. By generating a new row for each element in the array or map, the explode function can divide a column of arrays or maps into numerous rows.

The functions module is imported by the **pyspark.sql import functions as func** statement. A large variety of built-in functions, including aggregations, and string operations, are available in the functions module and can be used for data manipulation and transformation.

Loading the dataset

```
#Loading the dataset

csv_skill_50k = spark.read.format("csv") \
        .option("header", "false") \
        .load("dbfs:/FileStore/shared_uploads/sbharti1@sheffield.ac.uk/skill2vec_50K.csv.gz")
```

The data format to be read is specified by the **spark.read.format("csv")** declaration in this case.

The dataset does not have a header row, as indicated by the **.option("header", "false")** declaration.

The dataset is loaded from the supplied file location using the **.load("dbfs:/FileStore/shared_uploads/sbharti1@sheffield.ac.uk/skill2vec_50K.csv.gz")** command.

Importing necessary functions

```
from pyspark.sql.functions import array, col, explode, lit, array_except, expr, countDistinct, desc, lower
```

array(*cols): This function creates a new array column with the values of the inputted columns. It accepts a list of column names or expressions as input.

col(colName): Using the method col(colName), you can retrieve a column by its name.

explode(column): It creates a new DataFrame with a row for each element in the array by taking an array column as input. It "explodes" the array, resulting in numerous rows with the same values for the columns that are not part of the array but different values for the exploded column.

lit(literal): Using the lit(literal) function, a literal value can be generated that can be used as a constant in PySpark expressions.

array_except(col1, col2): It returns a new array column that contains the elements from the first array column that are missing from the second array column. This method accepts two array columns as input.

expr(str): Using the expr(str) method, a string can be evaluated as a Spark SQL expression. By slicing the "not_null" array column with the slice() function, a new column called "Skills" is added to the non_nullDF DataFrame in the preparation stage.

countDistinct(): It is used to calculate the number of distinct values in a column.

desc(): Used to sort a column in descending order.

lower(): Used to convert a string or column to lowercase.

Pre-processing Code:

```
csv_skill_50k_columns = csv_skill_50k.columns
list_csv_skill_50k = csv_skill_50k.select(array(csv_skill_50k_columns).alias("csv_skill_50k_list"))
```

The first line gets the csv_skill_50k DataFrame's column names and assigns them to the csv_skill_50k_columns variable. Later, it is used to reference the column names.

The second line selects all columns from csv_skill_50k and uses the array() method to turn them into an array column called "csv_skill_50k_list" that is then used to create a new DataFrame, list_csv_skill_50k.

```
non_null_list_csv_skill_50k = list_csv_skill_50k.withColumn("not_null",array_except("csv_skill_50k_list",array(lit(None))))
```

This line uses the array_except() function to remove any null values from the "csv_skill_50k_list" array column, creating a new column called "not_null" in the non_null_list_csv_skill_50k DataFrame.

```
non_null_list_csv_skill_50k = non_null_list_csv_skill_50k.drop("csv_skill_50k_list")
```

This line uses the drop() method to remove the "csv_skill_50k_list" column from the non_null_list_csv_skill_50k DataFrame.

```
non_nullDF = non_null_list_csv_skill_50k.withColumn('JD',non_null_list_csv_skill_50k['not_null'][0])
```

This line adds the "JD" column to the DataFrame by using indexing to take the first entry of the "not_null" array column.

```
non_nullDF=non_nullDF.withColumn("Skills",expr("slice(not_null,2,size(not_null))"))
```

This line uses the slice() method from the expr() function to add the additional column "Skills" to the DataFrame. Starting with the second element, the "not_null" array column is extracted.

```
non_nullDF= non_nullDF.drop("not_null")
```

This line removes the "not_null" column from the DataFrame.

```
Final_Job_Data = non_nullDF.select("JD", explode("Skills").alias("Skills"))
```

The final DataFrame is created by this line with two columns, 1: JD(Job Description column) 2: Skills(Number of skills)

Assignment Problems

Problem 1:

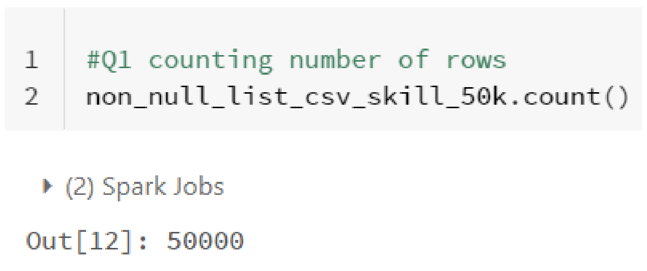
Assumptions:

Each job description is given a unique code in the "JD" column.

It is expected that the dataset being used is accurate and devoid of any corrupted or missing records that can interfere with the count function

Since the count operation is carried out on non-null records, it is assumed that any null values were effectively eliminated during the pre-processing stage.

Result:



```
#Q1 counting number of rows
non_null_list_csv_skill_50k.count()
```

```
▸ (2) Spark Jobs
Out[12]: 50000
```

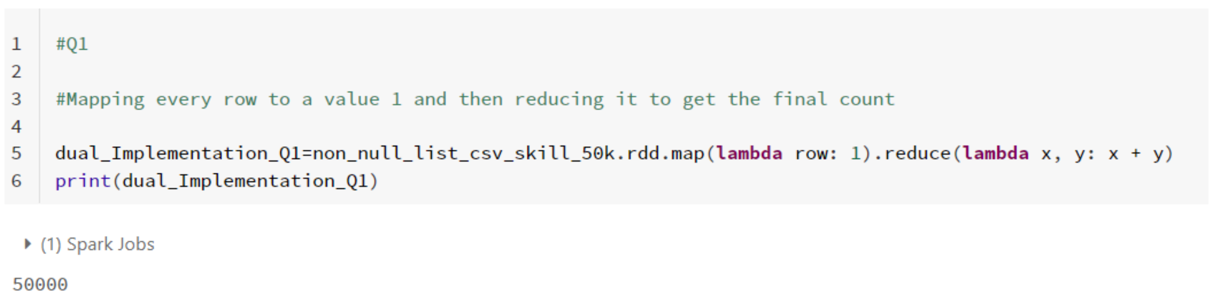
Discussion:

The dataset's extensive coverage of job descriptions leads to the conclusion that it is appropriate for in-depth investigation and analysis. This huge dataset makes it possible to thoroughly investigate industry preferences, skill needs, and job market trends, which helps with workforce planning and talent acquisition decisions. This dataset can be used by HR specialists, recruiters, and hiring managers to develop a thorough grasp of the job market trends and adapt their strategies accordingly.

Code:

The number of records in the DataFrame non_null_list_csv_skill_50k can be found using the method non_null_list_csv_skill_50k.count(). After eliminating any records with empty or missing information, the outcome of this function is the number of job descriptions that are present in the dataset. The entire number of accurate and comprehensive job descriptions that can be examined further is represented by this number.

Alternate Code:



```
#Q1

#Mapping every row to a value 1 and then reducing it to get the final count

dual_Implementation_Q1=non_null_list_csv_skill_50k.rdd.map(lambda row: 1).reduce(lambda x, y: x + y)
print(dual_Implementation_Q1)
```

```
▸ (1) Spark Jobs
50000
```

Code: dual_Implementation_Q1=non_null_list_csv_skill_50k.rdd.map(lambda row: 1). In the RDD (Resilient Distributed Dataset) produced from the DataFrame non_null_list_csv_skill_50k, reduce(lambda x, y: x + y) calculates the total of all the components. The number 1 assigned to each row in the RDD denotes the existence of a non-null record. The total count of non-null records is then calculated by combining these values together during the reduction step.

Problem 2:

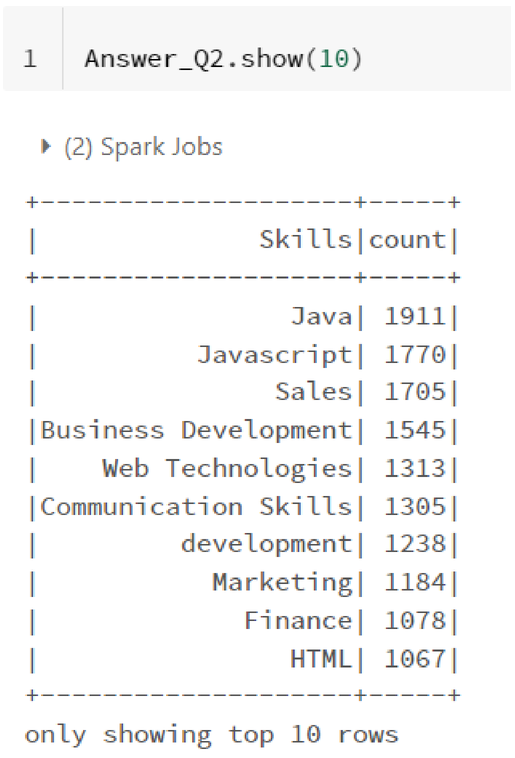
Assumptions:

The analysis relies on the skill identification procedure's precision and dependability to identify skills consistently across the dataset.

It assumes that words with different capitalization, such as "java" and "Java," are evaluated differently. It is believed that the dataset will include a vast array of job descriptions that represent various industries, jobs, and skill requirements.

The study makes the assumption that the proper precautions have been taken to guarantee data integrity through cleaning, standardization, and validation procedures while maintaining capitalization distinctions.

Result:



```
Answer_Q2.show(10)
```

(2) Spark Jobs

```
+--------------------+-----+
|              Skills|count|
+--------------------+-----+
|                Java| 1911|
|          Javascript| 1770|
|               Sales| 1705|
|Business Development| 1545|
|    Web Technologies| 1313|
|Communication Skills| 1305|
|         development| 1238|
|           Marketing| 1184|
|             Finance| 1078|
|                HTML| 1067|
+--------------------+-----+
only showing top 10 rows
```

Visualization:

```
# Converting the dataframe to pandas

visualisation_1 = Answer_Q2.limit(10).toPandas()
```

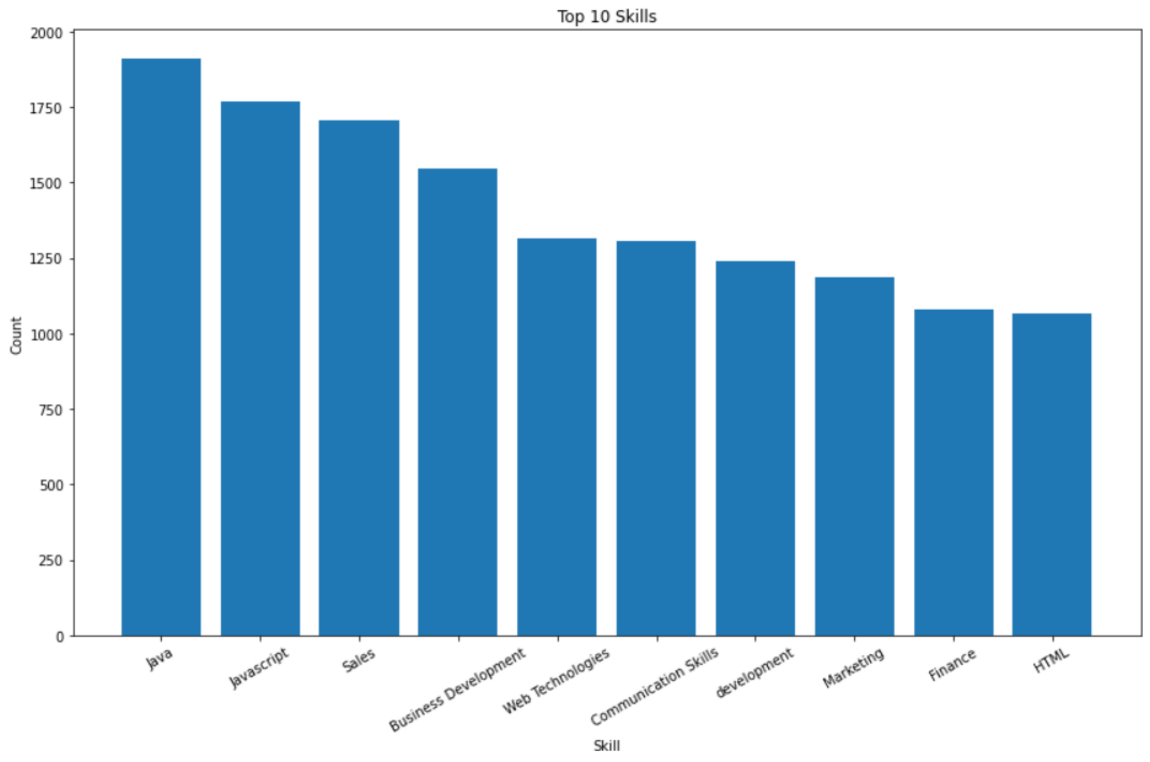
```
#creating bar chart

#size of chart
p.figure(figsize=(12, 8))

#type of chart
p.bar(visualisation_1['Skills'], visualisation_1['count'])

#naming the axis and title
p.title('Top 10 Skills')
p.ylabel('Count')
p.xlabel('Skill')

#rotating x-axis as the name of skills were overlapping
p.xticks(rotation=30)
p.tight_layout()
```



Discussion:

The dataset analysis provides many insights into industry trends and required skills. Java and Javascript's popularity demonstrate a high demand for programming languages in software and online development, highlighting their significance in the market. Communication skills is also considered very important in today’s market, employers look for professionals who can convey information to the clients clearly. The fact that finance is mentioned shows the necessity of it in decision-making. With the help of these insights, job seekers can better coordinate their efforts to grow their skills, and companies can improve their hiring procedures. In order to have a complete picture, it is crucial to take into account various data sources and elements.

Code:

```
#Q2

#grouping by skills and counting the nunmber of individual skills

Answer_Q2 = Final_Job_Data.groupby("Skills").count().sort(desc("count"))
```

A group-by operation is carried out on the "Skills" column of the dataset "Final_Job_Data" using .groupby("Skills"). Then, depending on the count, it determines the number of instances for each skill and arranges the results in decreasing order. This offers an answer that demonstrates how frequently various skills are referenced in job descriptions. Skills are shown in the output along with their counts in decreasing order.

Alternate code:

```
#Q2
#mapping each row to a tuple of (Skills, 1)
#summing up the counts for each unique skill
#sorting the count in descending order
#converting rdd to dataframe

dual_ImpQ2 = Final_Job_Data.rdd.map(lambda row: (row["Skills"], 1)) \
    .reduceByKey(lambda x, y: x + y) \
    .sortBy(lambda x: x[1], ascending=False) \
    .toDF(["Skills", "count"])
dual_ImpQ2.show(10)
```

The "Final_Job_Data" DataFrame is transformed into an RDD, which then conducts a map transformation to map each row to a key-value pair with the skill name ("Skills" column) serving as the key and the value being set to 1. To group the key-value pairs by skill name and perform a reduction operation in order to determine the overall count for each skill, the reduceByKey transformation is used. The skills are then arranged according to their counts in decreasing order, and the output is changed back into a DataFrame.

Problem 3:

Assumptions:

It is assumed that the dataset is particularly for an open job market.

The dataset consists of a number of JDs with different numbers of skills.

Additionally, it is likely that certain skills are listed more than once in a single job description, which could result in greater skill count totals. Due to the focus on certain competencies or the inclusion of several varieties of the same skills, there may be repetition in skill mentions.

Result:

```
+----------+-----+
|Num_skills|count|
+----------+-----+
|        10|10477|
|         5| 3432|
|         6| 3405|
|         1| 3386|
|         7| 3345|
+----------+-----+
only showing top 5 rows
```

Visualisation:

```
# Converting the dataframe to pandas


visualisation_2 = Answer3.limit(5).toPandas()
```

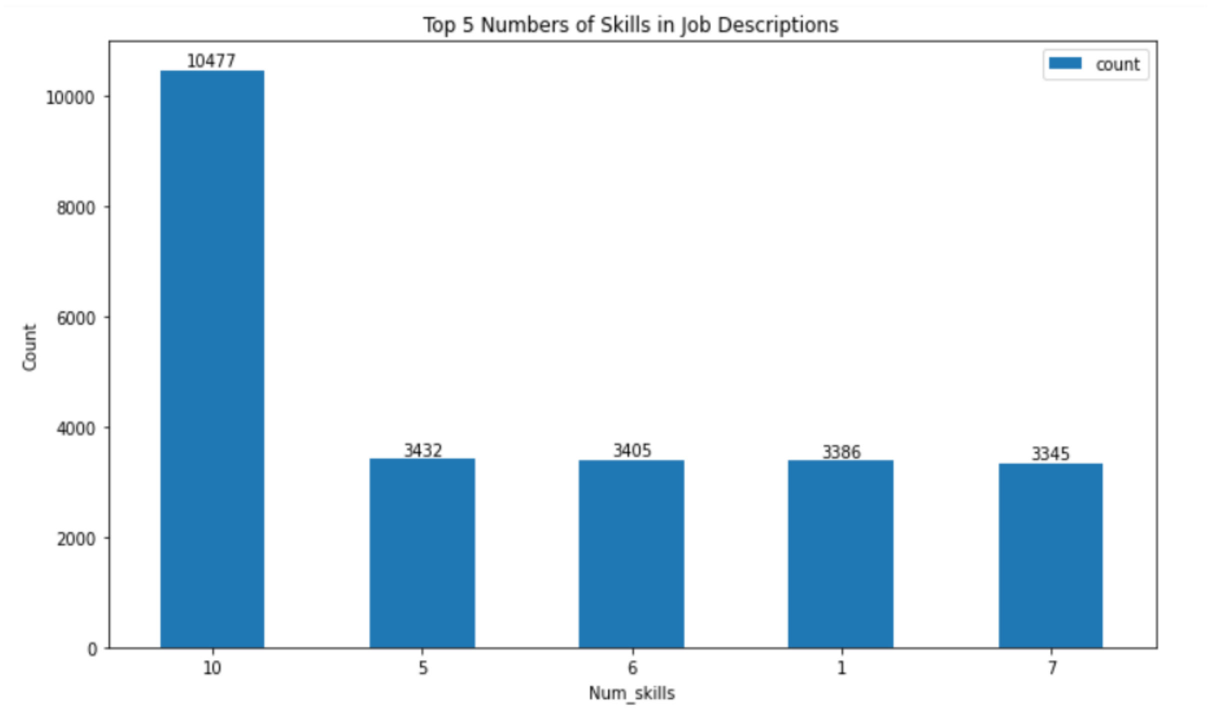
```
#creating bar chart

#plotting the chart

bar_2 = visualisation_2.plot(x='Num_skills', y='count', kind = 'bar', figsize=(10, 6))

#naming the chart
p.xlabel('Num_skills')
p.ylabel('Count')
p.title('Top 5 Numbers of Skills in Job Descriptions')
p.xticks(rotation=0)
p.tight_layout()

#adding number of counts to the bar
for x, y in enumerate(visualisation_2['count']):
    bar_2.text(x, y, str(y), ha='center', va='bottom')
p.show()
```



Discussion:

The study of the dataset demonstrates that the most often occurring job descriptions call for ten Skills, showing a considerable need for a wide skill set for a company. These professions are likely to need a mix of management, technical, and domain-specific skills, making them specialized or senior-level.

The existence of roles that place a high emphasis on specialization in a particular skill or domain is indicated by the considerable number of job descriptions that state the requirement of only one expertise. These findings give job seekers useful information that helps them determine the amount of experience and skill breadth that employers are looking for. Individuals can target opportunities that fit their skill profiles and adjust their applications by studying the skill requirements associated with various job descriptions.

Code:

```
#Q3

#Grouping by JD column and counting the number of instances

Question3 = Final_Job_Data.groupBy("JD").count()

#Removing JD column from dataset

Question3 = Question3.drop("JD")

#Renaming count column as Num_skills

Question3 = Question3.selectExpr("count as Num_skills")

#Grouping by Num_skills, counting the number of skills, sorting in decreasing order

Answer3= Question3.groupBy("Num_skills").count().sort(desc("count")).show(5)
```

The data is initially grouped by the "JD" column, and a count operation is then used to calculate the frequency of each distinct job description.

The "JD" column is subsequently removed from the resultant DataFrame because it is no longer required for the analysis.

The "count" column needs to be chosen and given the new name "Num_skills" using the selectExpr() function.

The dataset is sorted by the "Num_skills" column, and is then subjected to a count operation to ascertain the frequency of each distinct number of skills. Based on the count column, the results are arranged in descending order.

Alternate code:

```
#Q3

#every job description is mapped to a count of skills
#then reduced to frequency of each count

dual_ImpQ3 = Final_Job_Data.rdd.map(lambda row: (row["JD"], 1)) \
    .reduceByKey(lambda x, y: x + y) \
    .map(lambda a: a[1]) \
    .map(lambda count: (count, 1)) \
    .reduceByKey(lambda x, y: x + y) \
    .sortBy(lambda x: x[1], ascending=False) \
    .toDF(["Num_Skills", "count"])
dual_ImpQ3.show(5)
```

The dataset is processed by the code to ascertain how frequently various skill appear in job descriptions. Each job description is mapped to a count of skills, then reduced to the frequency of each count. The resulting RDD is converted into a DataFrame and sorted by frequency in descending order.

Problem 4:

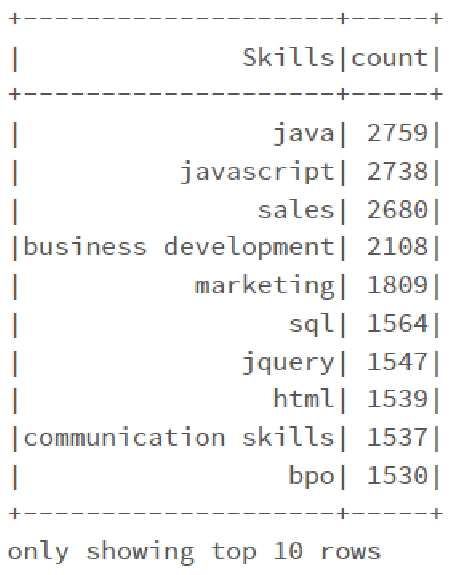
Assumptions:

It is assumed that case sensitivity has little to no influence on how employers assess talents. Additionally, it is assumed that lowercasing effectively conveys the skills' intended meaning and avoids ambiguity or misinterpretation.

These presumptions rely on the uniformity of job descriptions and the assumption that they accurately reflect the skills needed by the employer.

The analysis makes the assumption that lowercased skill frequency counts accurately reflect the demand for specific skills in the market.

Result:



```
+--------------------+-----+
|              Skills|count|
+--------------------+-----+
|                java| 2759|
|          javascript| 2738|
|               sales| 2680|
|business development| 2108|
|           marketing| 1809|
|                 sql| 1564|
|              jquery| 1547|
|                html| 1539|
|communication skills| 1537|
|                 bpo| 1530|
+--------------------+-----+
only showing top 10 rows
```

Discussion:

Even after lowercasing, the skill "java" is still quite prominent, demonstrating a persistent need for experts knowledgeable in the Java programming language. Similarly, "javascript" is still in a strong position, highlighting its significance in front-end programming and web development. The term "bpo" has emerged, indicating prospects in the business process outsourcing sector. Additionally, the popularity of "sql" has drawn attention to the need for qualified SQL users in positions involving database maintenance and data analysis. This information can help managers and job seekers make informed decisions about the changing skill requirements. A deeper understanding of skill patterns in the employment market may be possible with additional research and validation.

Code:

```
#Q4

#converting skills column to lowercase

question_4 = Final_Job_Data.withColumn("Skills", lower(Final_Job_Data["Skills"]))

#grouping by skills column, counting the number of instance and sorting in decreasing order

Answer_4 = question_4.groupby("Skills").count().sort(desc("count"))

#printing top 10 skills

Answer_4.show(10)
```

The lower() function is used in the code to lowercase the "Skills" column in the Final_Job_Data DataFrame. This makes sure that all skill names are consistently written in lowercase. The "Skills" column is then used to group the DataFrame, and the count() function is then used to determine the number of times that each distinct skill has been used. Based on the count, the outcome is ordered in descending order.

Alternate code:

```
#Q4

#RDD is mapped to a tuple with the value "Skills" in lowercase and a count of 1 and counting each distinct lowercased skills

dual_ImpQ4 = Final_Job_Data.rdd.map(lambda row: (row["Skills"].lower(), 1)) \
    .reduceByKey(lambda x, y: x + y) \
    .sortBy(lambda x: x[1], ascending=False) \
    .toDF(["Skills", "count"])
dual_ImpQ4.show(10)
```

The Final_Job_Data RDD is mapped to a tuple with the value "Skills" in lowercase and a count of 1. It then aggregates the counts for each distinct lowercased skill before reducing by key. The sortBy() function uses the count to sort the result in descending order. Finally, DataFrame with the columns "Skills" and "Count" is created from the resulting RDD.

Problem 5:

Assumptions:

The join operation between the JD dataset and the Example column of the O*NET dataset makes the assumption that lowercasing the skills offers a dependable and consistent way to align and compare them. It is based on the supposition that lowercasing will not cause any confusion or misinterpretation during the matching process.

The procedure is predicated on the idea that talents with identical lowercase names are treated as being the same, regardless of case or formatting differences. In order to guarantee reliable matching, it also assumes that the skill names in all datasets are uniform and correspond to similar naming patterns.

Result:

```
▸ (5) Spark Jobs

Before join: 463803
After join: 1101498
```

Discussion:

There were 463,803 distinct skills in the JD dataset prior to the join procedure. The total climbed to 1,101,498 after the datasets were combined, demonstrating a substantial overlap and intersection of skills between the two datasets. This shows that the ONET dataset's skills are extremely relevant and closely connected with the skills that employers desire, as seen by their presence in JDs. Job searchers may find this information useful in understanding the skills that are in demand and tailoring their resumes accordingly. Employers can use this information to make decisions about workforce planning, skill development, and talent acquisition.

Code:

```
#loading O*NET file

Tech_Skills_Onet = spark.read \
    .option("sep", "\t") \
    .option("header", "true") \
    .csv("dbfs:/FileStore/shared_uploads/sbharti1@sheffield.ac.uk/Technology_Skills.txt")
```

```
#Q5

#converting Example column to lowercase

question_5 = Tech_Skills_Onet.withColumn("Example", lower(Tech_Skills_Onet["Example"]))

#Changing name of example column to Skills

question_5 = question_5.select("Example")
question_5 = question_5.selectExpr("Example as Skills")
```

```
#taking the dataframe from question 4 and converting the skills column to lowercase

df_skills_jd = question_4.select(lower(col("Skills")).alias("Skills"))
df_skills_jd.show()
```

```
#joining the two dataframes

joined_final_df = df_skills_jd.join(question_5, "Skills")
```

```
#counting the result

before_join = Final_Job_Data.count()
after_join = joined_final_df.count()
print("Before join:", Q5_1)
print("After join:", Q5_2)
```

The lower() function is used to convert all of the skill examples in the "Example" column of the Tech_Skills_Onet dataset to lowercase.

The lowercased Tech_Skills_Onet dataset's "Example" column is chosen and added to the "Skills" column in the question_5 DataFrame.

The lower() function is used to further convert the "Skills" column in the question_4(original) DataFrame, which contains the lowercased skills from job descriptions.

The counts of records before and after the join operation are stored, respectively, in the variables Q5_1 and Q5_2. In the original JD dataset (Final_Job_Data), before_join reflects the number of records, and after_join represents the number of records following the join operation.

Alternate Code:

```
#loading O*NET file

Tech_Skills_Onet = spark.read \
    .option("sep", "\t") \
    .option("header", "true") \
    .csv("dbfs:/FileStore/shared_uploads/sbharti1@sheffield.ac.uk/Technology_Skills.txt")
```

```
#loading before_join df from the Final_Job_Data dataframe and converting skills column to lowercase

before_join_df = Final_Job_Data.withColumn("Skills", lower(Final_Job_Data["Skills"]))
```

```
#Q5

#creating teamp view

Tech_Skills_Onet.createOrReplaceTempView("Tech_Skills")
before_join_df.createOrReplaceTempView("Before_join")
Final_Job_Data.createOrReplaceTempView("Final_Job_Data")

# creating new temp view with lowercase skills

spark.sql("SELECT lower(Example) AS Skills FROM Tech_Skills").createOrReplaceTempView("Tech_Skills_lower")

# joining the views

final_df = spark.sql("""
    SELECT jd.Skills, ds.Skills AS Example
    FROM Before_join jd
    JOIN Tech_Skills_lower ds ON lower(jd.Skills) = ds.Skills
""")

# counting the number of instances

dual_Q5_1 = spark.sql("SELECT COUNT(*) FROM Final_Job_Data").first()[0]
dual_Q5_2 = final_df.count()

#
print("Before join:", dual_Q5_1)
print("After join:", dual_Q5_2)
```

The temporary views "Tech_Skills," "before_join_df"," and "Final_Job_Data" are registered for the datasets Tech_Skills_txt, before_join, and final_job_data, respectively. This makes it possible to use SQL queries on these views.

The "Tech_Skills" lowercase "Example" column is chosen using a Spark SQL query. A new temporary view with the name "Tech_Skills_lower" is created to store the outcome.

A SQL query is run to do an inner join between the "Tech_Skills_lower" view and the "before_join_df" view, which contains the lowercase JD skills. The JD skills and the Tech_Skills skills are matched by the join operation. The variable "final_df" is given the resultant DataFrame.

The number of records for both DataFrames is then counted and stored in "dual_Q5_1" and "dual_Q5_2", respectively.

Problem 6:

Assumptions:

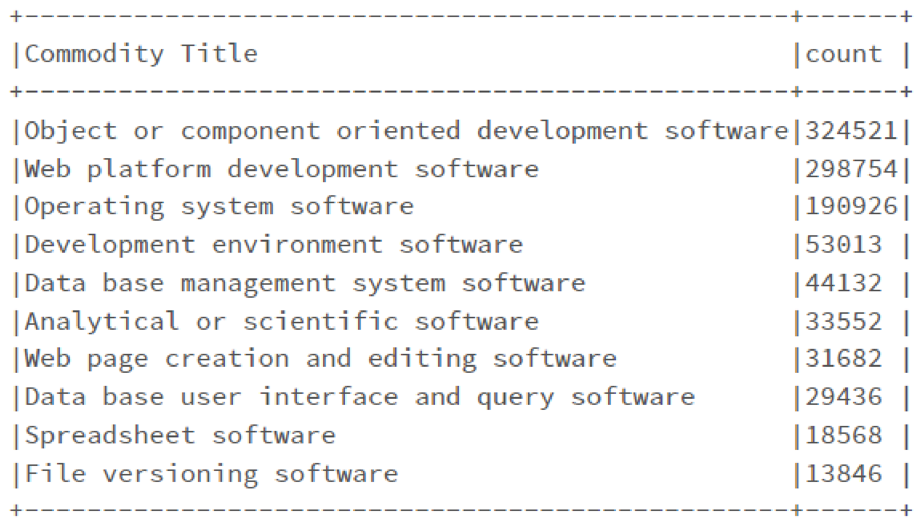
Each distinct value in the "Commodity Title" column denotes a particular class or subcategory of a skill.

Text descriptions of the skills are provided in the "Commodity Title" column.

If many skills fall under the same category or subcategory of technology or software, they can all map to the same "Commodity Title".

The number of times each distinct "Commodity Title" appears in all job descriptions can be used to calculate how frequently each "Commodity Title" appears.

Result:



```
+-------------------------------------------------+------+
|Commodity Title                                  |count |
+-------------------------------------------------+------+
|Object or component oriented development software|324521|
|Web platform development software                |298754|
|Operating system software                        |190926|
|Development environment software                 |53013 |
|Data base management system software             |44132 |
|Analytical or scientific software                |33552 |
|Web page creation and editing software           |31682 |
|Data base user interface and query software      |29436 |
|Spreadsheet software                             |18568 |
|File versioning software                         |13846 |
+-------------------------------------------------+------+
```

Discussion:

Specific Commodity Titles' frequency distribution in job descriptions shows the importance and demand they have in the market. For instance, the popularity of terms like "Web platform development software" and "Object or component-oriented development software" suggests a significant need for experts in web development and object-oriented programming. Various job titles emphasize the demand for applicants with knowledge in various fields to satisfy requirements particular to the sector. Job searchers who are proficient in these tools will have an advantage in landing positions that call for such abilities. Employers can use this information to tailor their hiring procedures to the most in-demand abilities and draw in qualified individuals. Employers and job seekers can use these insights to make well-informed decisions about hiring and skill development.

Code:

```
#Q6

#converting example column to lowercase

question_6 = Tech_Skills_Onet.withColumn("Example", lower(Tech_Skills_Onet["Example"]))
question_6 = question_6.withColumnRenamed("Example","Skills")
question_6.show()
```

```
#joining the two dataframes

joined_6_df = df_skills_jd.join(question_6, "Skills")
```

```
#grouping and counting the commodity title

commodity_title = joined_6_df.groupBy("Commodity Title").count()
```

```
#sorting in descending order

final_count = commodity_title.orderBy(desc("count"))
```

```
#extracting top 10 results

top_10 = final_count.limit(10)
```

The lower() function is used to lowercase the column "Example" in the "Tech_Skills_txt" DataFrame, and the output is saved in a new column named "Skills".

The withColumnRenamed() function renames the column "Example" to "Skills" to join "jd_skills_df" and "question_6" dataframes.

The resulting dataframe is then grouped by the “Commodity Title” column and the count of each title is determined using the count() function. The orderBy() function is used to sort the resulting counts in descending order.

The limit() function is used to choose the top 10 titles based on counts.

Alternate code:

```
# creating temporary views

Tech_Skills_Onet.createOrReplaceTempView("Tech_Skills_Onet")
df_skills_jd.createOrReplaceTempView("df_skills_jd")

# joining the views

joined_6_df = spark.sql("""
    SELECT df.Skills, sd.`Commodity Title`
    FROM df_skills_jd df
    JOIN Tech_Skills_Onet sd ON lower(df.Skills) = lower(sd.Example)
""")

# Counting the Commodity Titles

commodity_title = joined_6_df.groupBy("`Commodity Title`").count()

# Sorting in descending order

final_count = commodity_title.orderBy(desc("count"))

#selecting top 10 results

top_10 = final_count.limit(10)

# Displaying the top 10 titles

top_10.show(truncate=False)
```

Using the createOrReplaceTempView method, the DataFrames Tech_Skills_Onet and df_skills_jd are registered as temporary views.

The df_skills_jd view and the Tech_Skills_Onet view are joined to produce the joined_6_df DataFrame.

GroupBy is applied to the Commodity Title column and the number of occurrences is counted.

The Dataframe is then sorted in decreasing order and the top 10 results are printed using limit() function.