DSCI 617 – HW 02 Instructions

General Instructions

Navigate to the **Homework** folder inside of your user directory in the Databricks workspace. Create a notebook named **HW_02** inside the **Homework** folder.

Any set of instructions you see in this document with an orange bar to the left will indicate a place where you should create a markdown cell. For each new problem, create a markdown cell that indicates the title of that problem as a level 2 header. Any set of instructions you see with a blue bar to the left will provide instructions for creating a single code cell.

Add a markdown cell that displays the following text as a level 1 header: **DSCI 617 – Homework 02**. Within the same cell, on the line below the header, add your name in bold.

Add a code cell that imports the **`SparkSession`** class and the **`pandas`** package. Use the standard alias for **`pandas`**. Also import the **`punctuation`** string from the **`string`** library.

Add another code cell to create **`SparkSession`** and **`sparkContext`** objects named **`spark`** and **`sc`**.

Problem 1: Word Count

In the next few problems, we will work with a text file that contains the complete works of William Shakespeare. The data file using for this problem is located at: **`/FileStore/tables/shakespeare_complete.txt`**.

We will begin by loading and processing the file and tokenizing the lines into individual words.

Complete the following steps in a single code cell:

1. Read the contents of the file **`shakespeare_complete.txt`** into an RDD named **`ws_lines`**.
2. Create an RDD named **ws_words** by applying the transformations described below. This will require several uses of **`map()`** and **`flatMap()`** and a single call to **`filter()`**. Try to chain together the transformations together to complete all of these steps with a single statement (that will likely span multiple lines).
    - Tokenize the strings in **ws_lines** by splitting them on the 8 characters in the following list:

      `[' ', '-', '_', '.', ',', ':', '|', '\t']`

      The resulting RDD should consist of strings rather than lists of strings. This will require multiple separate uses of **`flatMap()`** and **`split()`**.
    - Use the Python string method **`strip()`** with the **`punctuation`** string to remove common punctuation symbols from the start and end of the tokens. Then use **`strip()`** again with the string **`'0123456789'`** to remove numbers from the start and end of the tokens.

(Code cell continued on next page.)

(Code cell continued from previous page.)

- Use the Python string method **`replace()`** to replaces instances of the single quote/apostrophe `"'"` with the empty string `''`.
- Convert all strings to lower case using the **`lower()`** string method.
- The steps above will create some empty strings of the form `''` within the RDD. Filter out these empty strings.

3. Create a second RDD named **`dist_words`** that contains only one copy of each word found in **`ws_words`**.
4. Print the number of words in **`ws_words`** and the number of distinct words using the format shown below. Add spacing so that the numbers are left-aligned.

```
Total Number of Words:    xxxx
Number of Distinct Words: xxxx
```

We will now use **`sample()`** to get a sense as to the types of words found in **`ws_words`**.

Draw a sample from **`ws_words`** using the arguments **`withReplacement=False`** and **`fraction=0.0001`**. Collect and print the results.

Problem 2: Longest Words

We will now find the longest words used by Shakespeare. We will start by looking for the single longest word.

Complete the following steps in a single code cell:

1. Write a Python function with two parameters, both of which are intended to be strings. The function should return the longer of the two strings. If the strings are the same length, then the function should return the word that appears later when ordered lexicographically (alphabetically).
2. Use the function you wrote along with **`reduce()`** to find the longest word in the RDD **`dist_words`**. Print the result.

We will now find the 20 longest words used by Shakespeare.

Use **`sortBy()`** with the Python **`len()`** function to sort the elements of **`dist_words`** according to their length, with longer words appearing first. Print the first 20 elements of this RDD.

Problem 3: Word Frequency

We will now create a frequency distribution for the words appearing in our document in order to determine which words were used most frequently by Shakespeare.

Complete the following steps in a single code cell:

1. Create an RDD named **`pairs`**. This RDD should consist of tuples of the form **`(x, 1)`**, where **`x`** is a word in **`ws_words`**. The RDD **`pairs`** should contain one element for each element of **`ws_words`**.
2. Use **`reduceByKey()`** to group the pairs together according to their first elements (the words), summing together the integers stored in the second element (the 1s). This will produce an RDD with one pair for each distinct word. The first element will be the word and the second element will be a count for that word. Sort this RDD by the second tuple element (the count), in descending order. Name the resulting RDD **`word_counts`**.
3. Store the first 20 elements of **`word_counts`** in a list. Then use that list to create a Pandas DataFrame with two columns named **"Word"** and **"Count"**.
4. Display this DataFrame (without using the **`print()`** function).

Problem 4: Removing Stop Words

You will notice that, unsurprisingly, the words most frequently used by Shakespeare are very common words such as "the" and "and". We will remove these common words and then recreate our frequency distribution. Words that are filtered out prior to performing a text-based analysis are referred to as **stop words**. There is no commonly accepted definition of what is and what is not a stop word, and the definition used could vary by task. A document containing a list of stop words to use in this assignment has been provided at the following path: **`/FileStore/tables/stopwords.txt`**. The document contains one word per line.

Complete the following steps in a single code cell:

1. Read the contents of the file **`stopwords.txt`** into an RDD named **`sw_rdd`**.
2. Print the number of elements in this RDD.
3. To get a sense as to the contents of the RDD, display a sample of elements contained within it. Perform the sampling without replacement and set **`fraction=0.05`**.
4. Store the full contents of **`sw_rdd`** in a list named **`sw`**.

We will now filter our collection of words to remove stop words and will then determine the number of distinct non-stop words.

Create an RDD named **`ws_words_f`** by removing from **`ws_words`** the elements that are also contained in the **`sw`** list. Then create an RDD named **`dist_words_f`** consisting of only the distinct elements from **`ws_words_f`**. Print the number of distinct non-stop words using the following format:

```
Number of Distinct Non-Stop Words: xxxx
```

We will now recreate our frequency distribution using only non-stop words.

Repeat the steps from Problem 3 using **`ws_words_f`** rather than **`ws_words`**.

Problem 5: Diamonds Dataset

We will now use the Diamonds Dataset to get an idea of how you might use RDD to work with structured data. This problem will provide useful practice for working with RDDs, but it should be mentioned that the DataFrame class (which we will discuss later in the course) is a much better tool the RDD class for working with structured data.

The file containing the data for this problem is located at the path: **`/FileStore/tables/diamonds.txt`**

This dataset contains information for nearly 54,000 diamonds sold in the United States. For each diamond, we have values for 10 variables. A description of each of the variables is provided below, in the order in which they appear in the file. The variables **cut**, **color**, and **clarity** are ordinal, or ranked categorical variables. The levels for these variables are provided below in order from worst to best.

• `carat`	Weight of the diamond.
• `cut`	Quality of the cut. **Levels: `Fair`, `Good`, `Very Good`, `Premium`, `Ideal`**
• `color`	Diamond color. **Levels: `J`, `I`, `H`, `G`, `F`, `E`, `D`**
• `clarity`	A measure of diamond clarity. **Levels: `I1`, `SI2`, `SI1`, `VS2`, `VS1`, `VVS2`, `VVS1`, `IF`**
• `depth`	Total depth percentage
• `table`	Width of top of diamond relative to widest point
• `price`	Price in US dollars
• `x`	Length in mm
• `y`	Width in mm
• `z`	Depth in mm

You can find more information about the Diamonds dataset here: Diamonds Data.

Read the contents of the tab-delimitated file **`diamonds.txt`** into an RDD named **`diamonds_raw`**. Print the number of elements in this RDD.

We will now get a glimpse at the contents of the RDD.

In a new cell, use a loop and the **`take()`** action to display the first five elements of **`diamonds_raw`**.

Note that the first element of **`diamonds_raw`** contains header information for the columns. In the next cell, we will filter out this row and will process each other row by tokenizing the rows and coercing each resulting element into the appropriate data type.

Create a function named **`process_row`** with a single parameter **`row`**, which is intended to accept strings representing (non-header) lines from the diamonds data file. The function should tokenize the line by splitting it on tab character **`'\t'`**, and then return a list of the individual tokens coerced into the correct data types. The data types for the tokens are, in order:

`float`, **`str`**, **`str`**, **`str`**, **`float`**, **`float`**, **`int`**, **`float`**, **`float`**, **`float`**

Use **`filter()`** to remove the header row from **`diamonds_raw`**. Then use **`map()`** to apply **`process_row()`** to each element of the filtered RDD. Store the results in a variable named **`diamonds`**.

Use a loop and the **`take()`** action to print the first 5 elements of this RDD.

Problem 6: Grouped Means

A diamond's cut is a categorical feature describing how well-proportioned the dimensions of the diamond are. This feature has five possible levels. These levels are, in increasing order of quality, **Fair**, **Good**, **Very Good**, **Premium**, and **Ideal**.

We will now use pair RDD tools to calculate the count, average price, and average carat size for diamonds with each of the five levels of cut. Note that for any tuple within the **`diamonds`** RDD:

- The **`carat`** size for the associated diamond is stored at index 0 of the tuple.
- The **`cut`** level for the associated diamond is stored at index 1 of the tuple.
- The **`price`** for the associated diamond is stored at index 6 of the tuple.

Complete the following steps in a single code cell:

1. Create a list named **cut_summary** by performing the transformations and action described below. Try to perform all of the steps with a single (multi-line) statement by chaining together the methods.
    - Transform each observation into a tuple of the form **`(cut, (carat, price, 1))`**. Note that the first element of this tuple indicates the cut level (which we will be grouping by), while the second element of the tuple is another tuple containing other information in which we are interested.
    - Use **`reduceByKey()`** to perform an elementwise sum of the tuples **`(carat, price, 1)`** for each separate value of the key, which is represented by the **cut** value. This will produce an RDD with 5 elements of the form **`(cut, (sum_of_carat, sum_of_price, count))`**.
    - Use **`map()`** to transform the tuples in the previous RDD into ones with the following form: **`(cut, count, mean_carat_size, mean_price)`**. Round the two means to 2 decimal places.
    - Call the **`collect()`** method to create the desired 5 element list.
2. To better display the results, use **cut_summary** to create a Pandas DataFrame named **`cut_df`**. Set the following names for the columns of the DataFrame: **`Cut`**, **`Count`**, **`Mean_Carat`**, **`Mean_Price`**.
3. Display **`cut_df`** (without using the **`print()`** function).

Submission Instructions

When you are done, click **Clear State and Run All**. If any cell produces an error that you are unable to correct, then manually run every cell after that one, in order. Export the notebook as an HTML file and then upload this file to Canvas. Do not alter your notebook on DataBricks after submitting unless you intend to resubmit a new HTML file. The notebook on Databricks should match the HTML file on Canvas.