Programming Assignment #4

Tests due: 4/30/23 @ 11:59pm
Implementation due: 5/7/23 @ 11:59pm

Assignment Link: https://classroom.github.com/a/IHqj4cDS

Please read through the entire writeup before beginning the programming assignment

Objectives

1. Implement common data science operations efficiently by using HashTables
    a. Load and clean two tables of data to prepare it for analysis
    b. Join the two tables of data on a common key to demonstrate how data can be efficiently de-anonymized if not carefully protected
    c. Aggregate the sensitive data on a common key to compute aggregate statistics that can be used for various types of analysis without potentially exposing sensitive information

Useful Links

1. The Scala API
    a. Scala Collections
    b. scala.io.Source
    c. scala.collections.mutable.Map
    d. scala.collection.mutable.HashMap
2. Scala Tour
3. Scala Resources
4. ScalaTest: Writing your first test
5. Maps in Scala
6. Scala File I/O (Scala Cookbook Excerpt)
7. Scala Exercises

Submission Process, Late Policy and Grading

Testing due date: 4/30/23 @ 11:59PM
Implementation due date: 5/7/23 @ 11:59PM
Total points: 30 (5 for testing + 20 for implementation + 5 for runtime)

The project grade is the grade assigned to the latest (most recent) submission made to Autolab (or 0 if no submissions are made). Autolab will pull your submission from your GitHub repository, so you must make sure that any changes you want to be included in your grade have been **committed and pushed.**

- If your submission is made before the deadline, you will be awarded 100% of the points your project earns.
- If your submission is made up to 24 hours after the deadline, you will be awarded 75% of the points your project earns.
- If your submission is more than 24 hours after the deadline, but within 48 hours of the deadline, you will be awarded 50% of the points your project earns.
- If your submission is made more than 48 hours after the deadline, it will not be accepted.

You will have the ability to use three grace days throughout the semester, and at most two per assignment (since submissions are not accepted after two days). Using a grace day will negate the 25% penalty per day, but will not allow you to submit more than two days late. Please plan accordingly. You will not be able to recover a grace day if you decide to work late and your score is not sufficiently higher. Grace days are automatically applied to the first instances of late submissions, and are non-refundable. For example, if an assignment is due on a Friday and you make a submission on Saturday, you will automatically use a grace day, regardless of whether you perform better or not. Be sure to test your code before submitting, especially with late submissions in order to avoid wasting grace days.

Keep track of the time if you are working up until the deadline. Submissions become late after the set deadline. Keep in mind that submissions will close 48 hours after the original deadline and you will not be able to submit your code after that time.

Note: No late submissions will be accepted for the testing portion of the assignment, and no grace days can be used on the testing portion of the assignment.

Background on Anonymization

Suppose you are working for Company X and your team is tasked with producing a health record data set to be released to the participants in an upcoming hackathon. Your colleague removed the columns for names and says that the data is ready to go but you aren't convinced. After reviewing the data set, you are concerned that if someone were to combine your "anonymous" data with other publicly available sources, there could be trouble. Unfortunately your colleague doesn't share in your concerns and requires proof.

Is this so far-fetched? In 2007, Netflix held a competition to improve their recommendation algorithms. This required releasing a large amount of anonymous data from their customers' movie ratings and viewing histories. By combining the Netflix data set with the Internet movie database (IMDb) site data, researchers found a way to link the identity of a Netflix user to a user's IMDb profile based on the select reviews that they published.

With more data being generated and released (and redacted) every day, this is not limited to an old and irrelevant data set. Consider the APIs that expose locations of bikeshare/scooter information (e.g., NABSA). Making these locations publicly available through an API allows for this data to be incorporated into ecosystems that extend beyond the provider's control. As a result you can get useful features like Google maps providing walking directions to the closest scooter/bike within a city, regardless of the service provider. But what is the cost of this? After all, these locations are public knowledge since we can physically see these bikes and scooters on the street. So the exact location of a scooter at a given time may not expose much. What about the pairs of start and end locations along with the date and time of usage? Now we can track usage habits at a particular location and possibly an address. Combining this data and auxiliary information we may be able to identify a set of candidates that are using the scooters. Further analysis may expose things such as daily habits -- particularly when someone normally leaves their house and returns, exposing further sensitive information. Fortunately there are techniques to thwart this.

To address this we have two main problems to tackle: (i) maintain the statistical relevance of the sensitive data and (ii) protect the people represented by the data. Fortunately, there is a well-studied body of tools and techniques under the umbrella of differential privacy for exactly this purpose. By training a model to understand an originally sensitive, but anonymized, data set we can generate synthetic data that looks statistically similar but further protects the individuals contained within. To see more about this, you can read the blog post by Alexander Watson and follow along with the example notebook they provide to understand further.

Setup

In order to complete this project, you must have completed PA0. If you are working on a machine other than the one you used in PA0, you must at least complete steps 2 and 4 in order to get IntelliJ and GitHub working properly.

Once you have ensured your development environment is setup as in PA0, you can accept the PA4 assignment in GitHub Classroom (here), and create a new IntelliJ project from VCS with your newly created repository.

Instructions

In this assignment you will implement a set of tools for doing some data analysis on two different sets of data, and demonstrate some of the risks associated with releasing "anonymized" data. You will implement functions that perform three common tasks in data science: loading and cleaning data, joining two different data sets, and computing aggregate statistics for different features of a data set.

Note: You **must NOT** modify any files other than **`DataTools.scala`** and **`DataToolsTests.scala`** to complete the lab.

As with PA3, the first phase of PA4 will be to write tests which must be able to pass on perfect implementations of the **`DataTools`** functions, and fail broken implementations. The second phase will be to implement the **`DataTools`** functions yourself.

After you complete your tests, make sure to commit **and push** your work to GitHub, and submit to the PA4 testing submission in Autolab. After completing your implementation, make sure to commit **and push** your work to GitHub, and submit to the PA4 implementation submission in Autolab.

Hint: It is advised that you commit and push frequently rather than waiting until you've completed everything.

Hint: Although you will get feedback from Autolab about correctness of your solutions, you should get in the habit of testing locally, and adding test cases as needed. This will be a more effective/efficient means of development, and will also give you a better understanding of the content of this programming assignment in the process.

0. Testing Phase (due 4/9/23) [5/30 points]

Modify the file **DataTools.scala** to include new test cases.

Your test cases will first be run against a correct implementation of **DataTools**. If your tests fail the correct implementation you will receive 0 points for the testing phase.

Your test cases will then be run against several broken implementations of **DataTools**. You will get points for each broken implementation that at least one of your tests fail.

1. Load and Clean Data [4/30 points]

Implement the following functions in **DataTools**:

```
def loadHealthRecords(filename: File): Seq[HealthRecord]
def loadVoterRecords(filename: File): Seq[VoterRecord]
```

Each of these functions should read the passed in CSV file (see scala.io.Source), and parse each line of the file into a **HealthRecord** (or **VoterRecord**) object. The output of these functions should be a **Seq** containing the parsed **HealthRecords** (or **VoterRecords**).

Properly formatted examples of each type of CSV file can be found in **src/test/resources**.

Header fields for health records are:

- "Birthday"
- "Zip Code"
- "Blood Type"
- "Allergic to Dogs?"
- "Allergic to Cats?"
- "Allergic to Peanuts?"
- "Allergic to Gluten?"

Header fields for voter records are:

- "First Name"
- "Last Name"
- "Birthday"
- "Zip Code"

Both functions can make the following assumptions:

- The passed in file exists
- The first line of the file is a header with the above fields (in that order)
- Every subsequent line can be formatted with **Strings split(...)** method
- Columns containing dates can be parsed using the provided **parseDate(...)** method
- Boolean columns will always contain "Yes" or "No" (but see below about capitalization)
- First and Last names can be read in exactly as they are in the CSV file
- If Zip Code is provided it can be read in exactly as it is in the CSV file

Issues you may need to address:

- Strings in each field may not have consistent capitalization
- Blood type should only be one of: A+, A-, B+, B-, O+, O-, AB+, AB-
    - If there is a typo for blood type, it should be entered as "N/A"
- Birthday and/or Zip Code may not be provided
    - If Zip Code is not provided then **null** should be stored
    - If Birthday is not provided, parseData will return **null** for you

2. De-Anonymize the Data [8/30 points]

Implement the following function in **DataTools**:

```
def identifyPersons(
    voterRecords: Seq[VoterRecord],
    healthRecords: Seq[HealthRecord]
  ): mutable.Map[String, HealthRecord]
```

The second function in this programming assignment shows the dangers of releasing "anonymized" data, by showing how easily that data can be de-anonymized if it can be related to other publicly available data. The function receives as input "anonymized" **HealthRecords** (personally identifiable information has been removed), and publicly available **VoterRecords**.

This function should return a **Map** of voters' full names (given by the **fullName** method of the **VoterRecord** class) to that person's **HealthRecord** if a unique match exists based on the person's Birthday and Zip Code.

A match between a **VoterRecord** and a **HealthRecord** exists if they have the same Birthday and Zip Code. The match is unique if there is only one **HealthRecord** that matches the **VoterRecord**, and there is only one **VoterRecord** that matches the **HealthRecord**.

Note: Some records may have missing data (ie the Birthday or Zip Code fields may be **null**). In these cases **null** is treated as a wildcard (it can match anything). For example, **(3/12/1990,null)** matches **(3/12/1990,14075)** AND **(3/12/1990,14127)**.

Hint: See the later section on joins for an explanation of various ways this matching can be done efficiently.

For ***m*** **VoterRecords** and ***n*** **HealthRecords**, the expected runtime of this function is ***O*(*m* + *n*).**

3. Compute Statistics [8/30 points]

Implement the following function in **DataTools**:

```
def computeHealthRecordDist(
    records: Seq[HealthRecord],
    attribute: HealthRecordAttribute
  ): mutable.Map[String, Double] = {
```

The last function in this programming assignment provides a potential solution to the de-anonymization problem from the previous function. It takes as input an "anonymized" sequence of **HealthRecords**, and a target attribute. It then returns the distribution of values the attribute takes so that statistically similar synthetic data can be generated and released. This data can be used by researchers to solve interesting problems, without risking the identities of the original **HealthRecords**.

If **attribute == HealthRecordBloodType**, then the returned **Map** should map the different blood types to the percentage of **HealthRecords** that have that blood type.

- The possible values for blood type are described in part one of this assignment
- Percentages should be a value in the range (0,1]

If **attribute == HealthRecordAllergies**, then the returned **Map** should map the different allergies to the percentage of **HealthRecords** that have that allergy.

- The possible allergies are "Dog", "Cat", "Peanut", "Gluten"
- Percentages should be a value in the range (0,1]

For ***n*** **HealthRecords**, the expected runtime of this function must be ***O*(*n*).**

4. Runtime complexity [5/30 points]

The final 5 points for this assignment will come after the submission deadline from separate testing that ensures your functions have the desired runtime complexity.

Additional Notes

Joins

Relational (SQL) databases like MySQL, Postgresql, SQLite, and others are used everywhere for data storage and processing. One of the main computational tasks relational databases handle on a regular basis is called a **join**. Although the computation you will be asked to implement in this assignment is not exactly a join, you may find the following both informative and helpful when implementing your assignment.

Relational Tables

A relational database stores data in **Tables** (you may have heard these called "Dataframes" or "Tensors" in other settings). A table is a collection of **Records**. All records have a common set of **Fields** (sometimes called Attributes). For example, the following is a table that we'll call **Customers**.

#	First Name	Last Name	Birthday	Zip Code
0	SIMON	DURAN	11/17/1978	14261
1	EMELIA	STEWART	9/23/1996	14201
2	NIA	GONZALEZ	7/12/1970	14210
3	VIOLET	HARMON	3/16/1986	14216
4	EDWIN	SUTTON	6/21/1986	14201
5	LILY	BAKER	8/31/1988	14213
6	MILO	ORTIZ	11/15/1973	14201
7	KARA	OLIVER	6/10/1968	14214
8	BRANDON	GARDNER	2/20/1957	14223
9	DEWEY	WILSON	3/14/1987	14212

There are 10 records (numbered 0-9) in this table. Each record has four fields: **First Name**, **Last Name**, **Birthday**, and **Zip Code**.

Here's another example that we'll call **Pizza By Zip**.

#	Zip Code	Closest Pizza
0	14201	La Nova Wing Incorporated
1	14216	Jet's Pizza Delivery
2	14223	Pie-O-Mine Greens
3	14213	Sports City Pizza Pub
4	14261	Imperial Pizza
5	14212	Pizza Express
6	14214	Just Pizza

Joins

A **relational join** (sometimes called an inner join, or just a join) links the records of two tables together on some field, called the **join key**. For each record in one table, we're going to find the corresponding record in the other table and produce a new "combined" record.

For example, let's join together the **Customer** and **Pizza By Zip** tables on their mutual **Zip Code** attribute (e.g., to find the recommended pizza place for each customer). For each record in **Customer**, we're going to find the record in **Pizza By Zip** with an identical **Zip Code**. The result should look like the following (the join operation is customarily denoted by the bowtie ⋈ operator):

Customer ⋈ Pizza by Zip

#	First Name	Last Name	Birthday	Zip Code	Zip Code	Closest Pizza
0	SIMON	DURAN	11/17/1978	14261	14261	Imperial Pizza
1	EMELIA	STEWART	9/23/1996	14201	14201	La Nova Wing Incorporated
2	VIOLET	HARMON	3/16/1986	14216	14216	Jet's Pizza Delivery
3	EDWIN	SUTTON	6/21/1986	14201	14201	La Nova Wing Incorporated
4	LILY	BAKER	8/31/1988	14213	14213	Sports City Pizza Pub
5	MILO	ORTIZ	11/15/1973	14201	14201	La Nova Wing Incorporated
6	KARA	OLIVER	6/10/1968	14214	14214	Just Pizza
7	BRANDON	GARDNER	2/20/1957	14223	14223	Pie-O-Mine Greens
8	DEWEY	WILSON	3/14/1987	14212	14212	Pizza Express

Note a few things about the above:

1. The output of the join operation is also a table.
2. The records in the output table have all of the fields of **both** input tables (in fact, the **Zip Code** attribute is repeated for this reason).
3. A single record in one table may join with multiple records in the other table (e.g., Record 0 of **Pizza By Zip** joins with Records 1, 4, and 6 of **Customers**).
4. If a record in one table does not match with any records in the other table, it is omitted from the output (e.g., Record 2 of **Customers** does not have a matching **Zip Code** in **Pizza By Zip**).

Nested Loop Join

The Join operation is usually defined by its simplest implementation, called the Nested-Loop Join. The algorithm, in lightly simplified form, appears as follows:

```
def nestedLoopJoin[R1, R2, K](
  tableA: Iterable[R1],
  tableB: Iterable[R2],
  getTableAKey: R1 => K,
  getTableBKey: R2 => K
): Seq[(R1, R2)] =
{
  val result = new ArrayBuffer[(R1, R2)]()
  for (a <- tableA) {
    for (b <- tableB) {
      if (getTableAKey(a) == getTableBKey(b)) {
        result.append((a, b))
      }
    }
  }
  return result.toSeq
}
```

The body of the function is a nested for loop. For every record in the **A** table, it iterates over every record in the **B** table to find the **B** record(s) that match. If it finds a matching record, it adds it to the result list. After it's done with every **A** record, it returns all of the matches it's made.

Runtime

The runtime of the nested-loop join algorithm is ***O*(|tableA|·|tableB|)**. From one perspective, this growth is only linear in each individual table's size. However, if we allow both tables to grow together (e.g., if we set **|tableA| = |tableB| = *n***), then the runtime becomes quadratic.

Hash Join

The runtime of the nested-loop join algorithm is high. To get a better runtime we can observe that a large part of the cost is repeatedly running the inner loop. As shown in WA3, we can sometimes benefit by doing a bit of preparation work up front before we start the algorithm. The result is what people who use and build relational databases call the "hash join" algorithm.

Note: Although the runtime of accesses to a HashTable are worst-case linear, we will use expected runtimes in this assignment.

Consider what happens when we load **`tableB`** into a **`HashMap`** first. Specifically for each record **`b`** in **`tableB`** , insert **`b`** into the **`HashMap`** with a key of **`getTableBKey(b)`** (this is typically called the *build* phase of hash join).

```
val hashTable = mutable.HashMap[K, List[R2]]()
for(b <- tableB){
  val key = getTableBKey(b)
  if(hashTable.contains(key)){
    hashTable(key) = b +: hashTable(key)
  } else {
    hashTable(key) = List(b)
  }
}
```

Note that multiple records of **`tableB`** may have the same join key. Since we can't rule this out in general, typically, the hash join will store a sequence of records for each join key.

Remember: The prepend operation, **`+:`**, on an immutable **`List`** only requires modifying the list head, and runs in ***O*(1)** time.

As discussed in class, building a HashMap with records at a predetermined load factor, α, takes an expected ***O*(*n*)** (i.e., ***O*(`|tableB|`)**) time, even accounting for any necessary resizes.

Next, when we loop over the records of **`tableA`** , we can recover all of the **`tableB`** records by probing the hash table instead of looping over **`tableB`**.

```
for(a <- tableA) {
  val key = getTableAKey(a)
  if(hashTable.contains(key)){
    for(b <- hashTable(key)){
      result.append((a, b))
    }
  }
}
```

We're still looping over every element of **tableA** , but now instead of a full iteration of **tableB**, we do a hash table lookup (technically two in the algorithm above: **contains(key)** and **hashTable(key)**), which has an expected runtime of ***O*(1)**. Since there might be multiple matches for a given record, **a**, we also need to iterate over all of these.

Although the worst-case runtime of this step (typically called the *probe* phase) can be ***O*(|tableA|·|tableB|)** (i.e., if every **b** record has the same key), it is typically much lower. As a result, it is customary to capture the runtime of the hash join by looking at the number of records it outputs. For example, if we know that every record in **tableB** joins with at most one record of **tableA** (as is the case for **Pizza by Zip**), then we can bound the number of records in the join output by **|tableB|**.

Observe that each iteration of the inner loop (over the records in **tableB**) appends one record to the result. Thus, the total runtime for this function is:

$$O\left(\sum_{a\in tableA}\left(1+\sum_{b\in tableB:a.key=b.key}1\right)\right)$$

Which simplifies to ***O*(|tableA|+|result|).** Specifically, it's possible for **|result|** to be as large as **|tableA|·|tableB|** (if there's exactly one join key value). but in practice, **result** is usually linear in the size of one (or both) tables.

Runtime

Combining the runtimes of the build and probe phases, we get an overall **expected** runtime for the hash join algorithm of: ***O*(|tableA|+|tableB|+|result|)**

Regardless of how the tables scale (and under the common assumption that the size of the output scales linearly with the size of the input), this function always grows **linearly.**

Other Tips

1. Before you actually start writing tests, come up with some example tables. They don't have to be particularly large or complex. Try to come up with interesting scenarios that may hit corner cases of what it is considered to be a "unique" match.
2. Once you start implementing, focus on a single function at a time. The functions in this programming assignment are all pretty self-contained.
3. For runtime complexity, remember to be aware of the cost of the various operations you are performing on your collections. Especially when dealing with sequences. Remember what was learned in PA3.

Academic Integrity

As a gentle reminder, please re-read the academic integrity policy of the course. I will continue to remind you throughout the semester and hope to avoid any incidents.

What Constitutes a Violation of Academic Integrity?

These bullets should be obvious things not to do (but commonly occur):

- Turning in your friend’s code/write-up (obvious).
- Turning in solutions you found on Google with all the variable names changed (should be obvious). This is a copyright violation, in addition to an AI violation.
- Turning in solutions you found on Google with all the variable names changed and 2 lines added (should be obvious). This is also a copyright violation.
- Paying someone to do your work. You may as well not submit the work since you will fail the exams and the course.
- Posting to forums asking someone to solve the problem.

Note: Aggregating every [stack overflow answer|result from google|other source] because you "understand it" will likely result in full credit on assignments (if you aren't caught) and then failure on every exam. Exams don't test if you know how to use Google, but rather test your understanding (i.e., can you understand the problems to arrive at a solution on your own). Also, other students are likely doing the same thing and then you will be wondering why 10 people that you don’t know have your solution.

Other violations that may not be as obvious:

- Working with a tutor who solves the assignment with you. If you have a tutor, please contact me so that I may discuss with them what help is allowed.
- Sending your code to a friend to help them. If another student uses/submits your code, you are also liable and will be punished.
- Joining a chatroom for the course where someone posts their code once they finish, with the honor code that everyone needs to change it in order to use it.
- Reading your friend’s code the night before it is due because you just need one more line to get everything working. It will most likely influence you directly or subconsciously to solve the problem identically, and your friend will also end up in trouble.

What Collaboration is Allowed?

Assignments in this course should be solved individually with only assistance from course staff and allowed resources. You may discuss and help one another with technical issues, such as how to get your compiler running, etc.

There is a gray area when it comes to discussing the problems with your peers and I do encourage you to work with one another to solve problems. That is the best way to learn and overcome obstacles. At the same time you need to be sure you do not overstep and not plagiarize. Talking out how you eventually reached the solution from a high level is okay:

> "I used a stack to store the data and then looked for the value to return."

but explaining every step in detail/pseudocode is not okay:

> "I copied the file tutorial into my code at the start of the function, then created a stack and pushed all of the data onto the stack, and finished by popping the elements until the value is found and use a return statement."

The first example is OK but the second is basically a summary of your code and is not acceptable, and remember that you shouldn't be showing any code at all for how to do any of it. Regardless of where you are working, you must always follow this rule: Never come away from discussions with your peers with any written work, either typed or photographed, and especially do not share or allow viewing of your written code.

What Resources are Allowed?

With all of this said, please feel free to use any [files|examples|tutorials] that we provide directly in your code (with proper attribution). Feel free to directly use anything from lectures or recitations. You will never be penalized for doing so, but should always provide attribution/citation for where you retrieved code from. Just remember, if you are citing an algorithm that is not provided by us, then you are probably overstepping.

More explicitly, you may use any of the following resources (with proper citation/attribution):

- Any example files posted on the course webpage (from lecture or recitation).
- Any code that the instructor provides.
- Any code that the TAs provide.
- Any code from the tour of Scala (https://docs.scala-lang.org/tour/tour-of-scala.html)
- Any code from Scala Collections (https://docs.scala-lang.org/overviews/collections-2.13/introduction.html)
- Any code from Scala API (https://www.scala-lang.org/api/2.13.0/)

Omitting citation/attribution will result in an AI violation (and lawsuits later in life at your job). This is true even if you are using resources provided.

Amnesty Policy

We understand that students are under a lot of pressure and people make mistakes. If you have concerns that you may have violated academic integrity on a particular assignment, and would like to withdraw the assignment, you may do so by sending me an email BEFORE THE VIOLATION IS DISCOVERED BY ME. The email should take the following format:

Dear Dr. Mikida,

I wish to inform you that on assignment X, the work I submitted was not entirely my own. I would like to withdraw my submission from consideration to preserve academic integrity.

J.Q. Student
Person #12345678
UBIT: jqstuden

When we receive this email, student J would receive a 0 on assignment X, but would not receive an F for the course, and would not be reported to the office of academic integrity.