Last Edited: 8/6/2018

Introduction

This research project was focused on extracting "hits" from a large set of data. The data consisted of readings from plates that contained oxides of various elements. Depending on the plate type, either high or low readings, relative to a standard, would constitute a "hit". The goal was to extract and display information on these hit plates in a manner that is efficient for a user.

The data for these plates is uploaded and stored in the phpMyAdmin database. It can be accessed by navigating to "solararmy.fsuhosting.com/cpanel". Under the databases section there is a link called "phpMyAdmin". Clicking this link brings you to a page from where you can access the directories of this database. The data for the plates is stored in the "solararm_data" which is a subdirectory of "solararm".

This database allows a user to run an SQL (Structured Query Language) query any of the directories and export the results. This means that it is possible for a user to ask the database to create a table that contains all information on all plate data stored in that database. This table can be exported as a file and stored on a local computer. A Java program in the form of a JAR file ("PlateDataFormatter.jar") then reads this file and sorts the contents into various formatted output text files. These output files are how the user quickly obtains information on hit plates. Detailed instructions on how to use the program are in the "Program Instructions" section of this report as well as in the program itself. To access the instructions from the program, simply click on the "Instructions" button of the program. This will display a dialog with text instructions on how to use the program.

I had two Java-based classes before I began this project. The classes were Object Oriented Programming and Network Programming. While writing this program my primary focus was combining the tools I had learned in these classes rather than learning new tools. Essentially, this program was a new application of techniques I had already learned. I highly recommend taking at least Object Oriented Programming before attempting to improve or change this program. I believe that class covers everything used in this program aside from threads which are a Network Programming topic.

This program was the most convenient – for me – method of extracting information from the plate data stored in the database. Initially, the idea for this program was conceived during a meeting with Dr. Hossain wherein he suggested creating a flat file from the database and parsing it on a local computer with a custom program. Since I had a decent understanding of the Java programming language, I had only to learn a basic understanding of how to use the software interface of the database and create an SQL query that fit our requirements. From that point on I was essentially working with familiar concepts applied to a new problem. This means that, comparatively, I could create a working version of the program in a relatively short period of time as opposed to a solution that is integrated in the Solar Army website. The cost is, obviously, convenience. With this program the user has to go through a fairly complicated process to obtain the flat file as opposed to simply entering the desired settings into a webpage and clicking "Go".

This report's primary purpose is to explain how the PlateDataFormatter program works to an individual with a basic understanding of Object Oriented Programming and Java. It is split into sections that cover the different pieces of the program. If you want a fairly detailed explanation for how the program

behaves in a single execution, read the sections "The PlateDataFormatter" and "The PlateDataMiner" in that order.

There are two further sections in this report that list all the files related to this project. The two sections are "Integral Files" and "Result Files". "Integral Files" contains descriptions for all the files that are either part of the program itself or are required by the program in some form. "Result Files" contains information on the output files created by the program when it is run.

This report also has an appendix in which further details are explained. Appendix 1 contains the instructions on how to use the program. Appendix 2 contains definitions for a variety of terms and phrases used throughout this report. Appendix 3 contains miscellaneous but critical aspects of the program that, without the necessary information, make little sense.

Program Overview

The purpose of this program was to read a CSV file, parse it, sort the resulting objects, and print the results to various files to allow a user to efficiently figure out which plates were "hits". Prior to the creation of this program, the fastest way to figure out which plates had good values would have involved using a custom SQL query on the database and attempting to read the resulting tables. Clearly, this is not a viable approach since it requires the user to know enough SQL to write a fitting query and it forces them to comprehend the contents of hard-to-read data tables. With this program, the user will run a prewritten SQL query on the database, place the resulting file in the same directory location as the program, run the program executable, and choose the desired options from the program's GUI (Graphical User Interface). The program will then create several result text files which contain easier-to-read plate information.

The program was written in Java and uses an object-oriented method to handle the extraction of useful information from the CSV input file. There are 11 files that are integral to the operation of the program and 10 result files. These files are listed with brief descriptions toward the end of this report.

When the program is run a GUI will be displayed. In this GUI the user can select the input file name, the lowest acceptable rating a valid plate can have, the highest acceptable coefficient of variation for a valid plate, and a button to enable or disable the creation of the "target_cells.txt" file. If this last button is selected, the program will display another window – partway through the formatting process – that will allow the user to select the desired elements from all elements found in all plates. Once the program has finished, the result files will be in same directory location as the program itself.

The PlateDataFormatter

When the user double clicks on the "PlateDataFormatter" JAR file an instance of the PlateDataFormatter class is created. This class contains the GUI components that display the initial frame in which the user can specify the desired rating, coefficient of variation, etc. When the user clicks the "Format" button in this frame, the PlateDataFormatter's setupFormat() method is called. This method performs some initial checks to make sure the user input is valid. If none of these checks fail, then it disables the GUI buttons, constructs four Vectors, and constructs a new Thread to execute the run() method of the

PlateDataFormatter. The run method constructs an instance of the ProgressDialog class which contains an instance JProgressBar. Effectively, this ProgressDialog displays as "loading bar" to tell the user that the program is indeed doing something. Then it calls the format() method of the PlateDataFormatter.

This format() method calls a variety of other PlateDataFormatter methods in a specific order. Initially, it calls processFile() which creates a new Plate object and reads one line at a time of the input CSV file. Each line is split around the space character and stored in a String array called "parts". Since each row in the input CSV file will always have the same number of columns, this parts array will always be the same size. Therefore, the items in this array can be parsed by their index positions. To parse the contents of parts, the parsePlateData() method is called.

This method takes the current Plate object and the parts array as arguments. This method systematically goes through all indexes of parts and fills in the data members of the Plate object. To read about the Plate object, see the "The Plate Object" section of this report. Once the entire line has been parsed, the code loops and another line is read. This process continues until the "data_id" part of each line does not match the previous line's "data_id". The "data_id" is a large number that is unique for each plate's data. If the current "data_id" is different from the previous one, it must mean that information for a new plate has been found. When this occurs, a conditional breaks the line-reading loop and the handlePlate() method is called.

The handlePlate() method does three primary things to the current Plate object. It calculates values for various data members of the Plate object by calling a consecutive series of methods, it applies a series of conditional checks on the Plate to determine whether it is valid, and it sorts the Plate into two of three Plate Vectors. The values that are calculated are the averages of the standard columns, the sample standard deviation of the standards, the sample coefficients of variation of the standards, and the sample coefficients of variation of the nonstandard columns. It also recalculates the "valueDividedByLeftAverage" and "valueDividedByRightAverage" data members of each Cell object of the Plate as well as the rating of the Plate itself. These recalculations were done to correct errors in the data stored in the database.

After these calculations, the conditional checks are applied. There are two boolean variables involved in this process: "problemPlate" and "inconsistentRatios". "problemPlate" is a data member of the PlateDataFormatter. If this variable is "true" then the current Plate object is considered invalid. "inconsistentRatios" is a data member of the Plate object. If this variable is true, then it means the ratios of the elements down each column were not consistent. Rather than listing this plate as invalid, a warning is printed to the output text files for each of these plates.

Initially, the code checks whether the combination of atomic symbols down each column is consistent. If this fails "problemPlate" is set to "true". At the same time, the ratios of these elements are also checked. If these are not consistent, then inconsistentRatios is set to "true". Next the Plate's rating is checked, if it is below the user-specified minimum value, "problemPlate" is set to "true". Then the averages of the standards are checked. For a red-lead plate, the iron standard should have a higher value than the copper standard. For a black-lead plate, the inverse should be true: the copper standard should be higher than the iron. The program checks these standards and sets "problemPlate" to "true" if any of these fail.

At this point the conditional checks are complete. The next step is to add the current Plate object to two of three Vectors. These Vectors are called "allPlatesVector", "goodPlatesVector", and "problemPlatesVector". These Vectors are data members of the PlateDataFormatter. The method adds the current Plate object to the "allPlatesVector" regardless of whether "problemPlate" is true or false. It adds the same Plate object to "goodPlatesVector" if "problemPlate" is false. Finally, it adds the Plate object to "problemPlateVector" if "problemPlate" is true.

At this point the data for the current Plate object has been fully parsed and handled and the Plate has been stored in the correct Vectors. Now the execution jumps back to the line-by-line reading code to begin with the next Plate object. This process continues until the program attempts to parse an empty String as a line of data. When this occurs, it means the end of the input CSV file has been reached. A boolean called "done" is set to true and the execution breaks out of the entire processing loop and exits the processFile() method.

Now the program is back in the format() method and proceeds to call the sortPlateVector() method on each of the three Plate Vectors. This method uses a selection sort algorithm to sort the Plate objects – which are stored in the aforementioned Plate Vectors – by their respective ratings into descending order. Then the writePlateVectorToFile() method is called three times for each of these Vectors. This method takes a String argument to figure out which Plate Vector it is printing. Depending on the value of the argument, it will print different prelude content for each output file. Then it will print the information for every single Plate in the Plate Vector in a formatted manner to the file. Once this is complete, the createSimplePlatesFile() method is called. This method creates concise versions of the previous three files. The idea here is that a user can quickly glance over this file to find the best plates. From there the user has the option to look at the other files if they require more information. At this point the program has finished handling the Plate objects and it can begin processing at a cell level.

To begin this process, the program calls the createDataMiningFileWithHeaders() method which creates a file with formatted information for every single Cell in every Plate. Then it constructs an instance of the PlateDataMiner and tells it the name of the newly created mining file as an argument to its constructor. Then it calls the initialize() method of the PlateDataMiner to start its operation.

The PlateDataMiner

When the PlateDataMiner's default constructor is called, it constructs two Vectors for Cell objects and one Vector of String objects. These Vectors are called "cells", "targetCells", and "targetElements", respectively. Once the PlateDataMiner's initialize() method is executed with no exceptions, a new Thread is created to execute the PlateDataMiner's run() method. Initially, the processFile() method is called from inside this run() method. This processFile() method is, essentially, a simpler version of the PlateDataFormatter's processFile() method. It reads one line from the mining file, splits and parses the line around the space character, creates a Cell object for each line, and stores the Cell object in the "cells" Vector if the Cell was not a standard. Then it loops back to the beginning and the process is repeated. This continues until an empty line is read.

The next method that is called inside run() is sortCellsByRating(). It is almost identical to the PlateDataFormatter's sortPlatesByRating() method except that it handles Cells rather than Plates. This

method sorts the Cells in the Vector based on the appropriate ratio-to-standard value of that Cell. Then the newly sorted Cells are printed to a file via the writeCellsToFile() method.

If the user selected the "Find cells with target elements:" button, then an instance of the ElementCombinationDialog is created at a specific point in the formatting process. This dialog contains a series of JRadioButtons for selecting any combination of atomic symbols present in the original input data file. When the user clicks the "Search" button in this dialog, the getCellsWithTargetElementCombination() method is called. This method finds all Cell objects that contain the target elements – which were obtained from the ElementCombinationDialog as user input – and stores them in the targetCells vector. Then, via the getPlatesWithTargetElementCombination() method, information on the Plate objects that contain a target Cell object is stored in a String Vector called "plateInfo". Then writeTargetCellsToFile() method is called, and the contents of the "plateInfo" and "targetCells" Vectors are printed to the "target_cells.txt" file.

To create the "target_cell_ratio_counts.txt" and "cell_ratio_counts.txt" files, two sets of methods are called. The first set consists of getTargetCellRatioCounts(), sortTargetCellRatioCounts(), and printTargetCellRatioCounts(). The second set is composed of getCellRatioCounts(), sortCellRatioCounts(), and printCellRatioCounts(). As the names suggest, the only difference between these two sets is that the first set specifically handles target cells (cells that contain target elements) while the second set handles all cells. As such the code for these methods is almost identical: the largest difference is that they use different Cell Vectors as "pools" to draw Cell objects from. To avoid repetition, only the first set of methods are explained in this report.

getTargetCellRatioCounts() initially constructs two Vectors: "countedCells" and "targetRatioCounts". The first Vector is used to determine whether a Cell has already been counted. This is necessary because the goal is to create a file that counts different element and ratio combinations separately. The program will use the first Cell in the "targetCells" Vector as a basis to compare all other Cell objects to. This means that when the program finishes comparing the first Cell object to all other Cell objects and starts with the second Cell object, there is a chance that that second Cell has the same element and ratio combination as the initial Cell. Consequently, there would be two rows in the output file with the same element and ratio combination which is not useful. To solve this problem, all counted Cell objects are stored in the "countedCells" Vector. When the program either compares a Cell to another Cell or uses a new Cell as a basis, it will check whether the current Cell is in the "countedCells" Vector. If it is, it will ignore that Cell since it has already been counted.

The second Vector "targetRatioCounts" will contain ElementRatioCount objects for each individual element and ratio combination. To read about the ElementRatioCount object, see the "The ElementRatioCount Object" section of this report.

Once these two Vectors are constructed, the program takes the first Cell object from the "targetCells" Vector as a basis. It constructs a new ElementRatioCount object and stores the basis Cell's element and ratio combination in that ElementRatioCount. It then loops through all Cells in the "targetCells" Vector and compares each Cell to the basis Cell. If the current Cell has the same element and ratio combination as the basis, "totalCount" data member of the ElementRatioCombination is increased by one and the current Cell is stored in the "countedCells" object. If it so happens that the Cell has a rating greater or equal to the rating threshold, then the "goodCount" data member of the ElementRatioCount is also increased by one. Once all Cells have been compared to the basis, the count information for the current

ElementRatioCount object has been fully filled in. Now the program calculates the percentage of the good cells compared to the total number of cells and stores the result in the "percentage" data member of the ElementRatioCount. Finally, the current ElementRatioCount object is stored in the "targetRatioCounts" Vector. The program then proceeds to take the next Cell in the "targetCells" Vector and repeats the above process until all Cells have been handled.

Now that the "targetRatioCounts" Vector contains all the ElementRatioCount objects, the sortTargetCellRatioCounts() method can be called. This method is very similar to the Cell and Plate sorting methods from earlier. The only difference is that it sorts ElementRatioObjects instead of Cells or Plates and it uses the "percentage" data member instead of the "rating". As with the other two methods, it sorts the objects into descending order – the highest percentage will be at the top of the file.

The last step is to call the "printTargetCellRatioCounts()" method. This method simply loops through each ElementRatioObject stored in the "targetRatioCounts" Vector and prints the contained information to the "target_cell_ratio_counts.txt" output file. The final step of the run() method is to enable all the GUI buttons and fields that were disabled at the start of the formatting process.

The Plate Object

A Plate object is intended to represent the data of a single plate. More precisely, it represents a specific plate number, plate type, and data_id combination. The Plate owns several data members which can be seen in the Plate.java file. The less intuitive ones are:

Cell cellTable[][];

int iron_nonStandard;

int copper_nonStandard;

BigInteger id;

The cellTable[][] data member represents the 6x6 table of cells that each physical plate has. It is a two dimensional array of Cell objects. The Cell object is explained in the "The Cell Object" section of this report.

The iron_nonStandard and copper_nonStandard data members will have a value of 1 if the element iron or copper was used in one of the inner four columns of the cellTable[][]. Currently, these data members ARE NOT USED in this program. Another method is used to identify the standards and nonstandards.

The id holds the data_id of the data for that Plate object. This is a variable that is stored in the database for each Plate upload. It is a large number that is, theoretically, unique for each plate. It is used to differentiate between Plate objects in this program. More precisely, it is used to identify when all rows of a Plate object have been read from the input CSV file – when the id of the current row no longer matches the id of the previous row, it means a new Plate object should be created.

The Cell Object

The Cell object is used to represent a single cell in the 6x6 table of cells that a physical plate has. A cell in this plate will have a reading to represent the flow of electrons for a particular combination of elements. The Cell object has several data members, the least intuitive are as follows:

double valueDividedByLeftAverage;

double valueDividedByRightAverage;

Vector<Element> elements;

The "valueDividedByLeftAverage" and "valueDividedByRightAverage" data members contain the value of the current Cell object (which is stored in the "value" data member of the Cell) divided by the average of the leftmost or rightmost column of the Plate object's 6x6 array of cells. These leftmost and rightmost arrays are standards. When a cell's value is divided by a standard and the resulting value is greater or equal to 1, then that cell's value is as good as or better than the average of those standards in terms of the flow of electrons.

The "elements" data member is a Vector of Element objects. Since the Cell object represents a position in the 6x6 table of cells in a plate, it will have a corresponding combination of elements. This means that for each Cell, the elements used in that Cell must be stored somewhere. In this program, they are stored as Element objects in a Vector. The Element object is explained in the "The Element Object" section of this report.

The Element Object

The Element object is very simple. It is used to represent a single Element in a Cell object and its corresponding ratio to any other Elements present in that Cell object. The atomic symbol of the element is stored in the "atomicSymbol" data member. The ratio is stored in the "ratio" data member.

The only unintuitive data member is "pos". When a row of information is read from the input CSV file, each row will only have one element. This means that there are multiple rows that represent the same cell in a plate. The only difference between these rows is their elements and the ratios of these elements. It is potentially useful to know the order in which these elements were listed in the database. Therefore, the position of the atomic symbol compared to the other atomic symbols is stored in the "pos" data member. Currently, the "pos" data member IS NOT USED in this program.

The ElementRatioCount Object

This object is used to represent a unique element and ratio combination and keep track of how many occurrences of this combination exist. This object contains four data members:

int goodCount;

int totalCount;

double percentage;

Vector<Element> elements;

Vector<BigInteger> data_ids;

"goodCount" contains the number of Cell objects that have the current element and ratio combination as well as a rating that is greater or equal to the rating threshold.

"totalCount" contains the number of Cell objects that have the current element and ratio combination regardless of their rating.

"percentage" contains the result of dividing "goodCount" by "totalCount" and multiplying by 100. Essentially, it contains the percentage of good cells to total cells for that element and ratio combination.

"elements" contains the current element and ratio combination in the form of a Vector of Element objects. To read more about the Element object, see the "The Element Object" section of this report.

"data_ids" contains the data_id of every plate that contained the current unique element and ratio combination.

Files

Integral Files:

input_data.csv – this is the input CSV file.

PlateDataFormatter.java – this Java file contains the main() method with which the program should be executed.

PlateDataMiner.java – this class primarily handles individual Cell objects obtained by reading the "mining.txt**"** file.

ElementCombinationDialog.java – a class to represent a dialog from which the user can select desired elements. Used by the PlateDataMiner.

ProgressDialog.java – a JFrame with a JProgressBar. Used in the program to tell the user that the program is actually doing something.

HelpDialog.java – a JFrame which contains and displays the contents of the "READ_ME.txt" file.

Plate.java – the class that outlines the properties of a Plate object. A Plate has a two-dimensional array of Cell objects.

Cell.java – the class that specifies what things a Cell should own. A Cell has an array of Element objects.

Element.java – the class that details the properties of an Element for the purposes of this program.

mining.txt – this file is created by the PlateDataFormatter. It is used by the PlateDataMiner to extract additional information on a Cell level.

READ_ME.txt – this file contains information on how to operate the program. It is used by the HelpDialog class which is instantiated when the user clicks on the "Instructions" JButton on the program's GUI.

PlateDataFormatter.jar – this is the Java program executable. Simply double click on this file to run the program.

Result Files:

all_data_sets.txt – this file contains information on all plates regardless of whether they failed any conditional checks.

all_data_sets_simple.txt – this file is a brief and more concise version of the "all_data_sets.txt" file.

valid_data_sets.txt – this file contains information on all plates that passed all the conditional checks. The data set conditions that must be met are:

Ratings >= 1.33

Sample coefficient of variation of the high standard was <= 50.0%

Correct high/low standards depending on plate type

valid_data_sets_simple.txt – this file is a brief and more concise version of the "valid_data_sets.txt" file.

invalid_data_sets.txt – this file contains information on all plates that failed any of the conditional checks. The data set conditions that must be met are:

Ratings < 1.33

Sample coefficient of variation of the high standard was > 50.0%

Incorrect high/low standards depending on plate type

invalid_data_sets_simple.txt – this file is a brief and more concise version of the "invalid_data_sets.txt" file.

cells.txt – this file contains information on all nonstandard cells from valid plates. Cells are sorted by their ratings into descending order. A single row in this file will have the following information in this format: *plate_number* *plate_type* *row* *col* *plate_rating* *elements* *data_id*

target_cells.txt – this file contains information on all cells that contain the target elements. The cells in this file are obtained from valid plates only. At the beginning of the file, brief information for each of the plates that contain a target cell is displayed. The target elements are specified by the user through the GUI.

target_cell_ratio_counts.txt – this file is intended to help the user determine which combination of elements and ratios is most prone to producing good cell readings. This file contains information on all cells – obtained from valid plates – that contained the target elements. More specifically: the file has

three columns consisting of a percentage, a ratio, and the element and ratio information for that row. The percentage is determined from the number of cells that had ratings higher than the rating threshold (good cells) divided by the total number of cells. The ratio is simply the number of good cells compared to the total number of cells. The element and ratio information has the following format: [*element_1* *ratio_of_element_1* *element_2* *ratio_of_element_2* ... *element_N* *ratio_of_element_N*].

In-file description:

"This file contains information on element and ratio combinations for all valid nonstandard cells that contain one or more target elements.

The purpose of this file is to show which of these combinations is most likely to produce useful cell readings.

The percentage shows how many cell readings were greater or equal to the rating threshold (which is determined by the user).

The ratio gives you the exact number of cells that were good as well as the total number of cells.

The left number in the ratio is the number of good cells while the right is the total number of cells.

The element and ratio combination for the current line is printed last.

This file should be read line by line: each line contains information unique to the element and ratio combination on that line."

cell_ratio_counts.txt – this file is the same as the previous file except that it disregards the target elements. It finds ALL combinations of elements and ratios.

In-file description:

"This file contains information on element and ratio combinations for all valid nonstandard cells.

The purpose of this file is to show which of these combinations is most likely to produce useful cell readings.

The percentage shows how many cell readings were greater or equal to the rating threshold (which is determined by the user).

The ratio gives you the exact number of cells that were good as well as the total number of cells.

The left number in the ratio is the number of good cells while the right is the total number of cells.

The element and ratio combination for the current line is printed last.

This file should be read line by line: each line contains information unique to the element and ratio combination on that line."

Appendix 1: Program Instructions

==========

FILE FORMAT:

This program requires a custom CSV file with a specific format.

In a single row of the input file should have the following values in the exact same order:

plate_number

plate_type

row

col

reading

reading / average_of_left_standards

reading / average_of_right_standards

atomic_symbol

atomic_symbol_position_in_element_ratios

element_ratios

iron_nonstandard

copper_nonstandard

data_id

In practice a few rows in the file should look like this:

101 red 0 0 1.545824824174791 1.2228187919463085 0.5592960196459634 fe 0 [0:0:0:1] 1 0 1500907535

101 red 0 0 1.545824824174791 1.2228187919463085 0.5592960196459634 cr 1 [0:0:0:1] 1 0 1500907535

101 red 0 0 1.545824824174791 1.2228187919463085 0.5592960196459634 sr 2 [0:0:0:1] 1 0 1500907535

101 red 0 0 1.545824824174791 1.2228187919463085 0.5592960196459634 cu 3 [0:0:0:1] 1 0 1500907535

101 red 0 1 3.213822411621355 2.542281879194631 1.1627954568709713 fe 0 [3:4:5:0] 1 0 1500907535

101 red 0 1 3.213822411621355 2.542281879194631 1.1627954568709713 cr 1 [3:4:5:0] 1 0 1500907535

101 red 0 1 3.213822411621355 2.542281879194631 1.1627954568709713 sr 2 [3:4:5:0] 1 0 1500907535

101 red 0 1 3.213822411621355 2.542281879194631 1.1627954568709713 cu 3 [3:4:5:0] 1 0 1500907535

==========

GETTING THE INPUT FILE:

Login to cpanel (URL should be something like: solararmy.fsuhosting.com/cpanel).

Once you have logged in, you should see several sections with different labels.

You should be able to see "FILES", "DATABASES", "DOMAINS", etc.

Under the "DATABASES" category, click "phpMyAdmin".

Once phpMyAdmin has finished loading, look at the leftmost side of the screen.

You should be able to see an expandable directory called "solararm".

You can tell that it is expandable by the "+" to the left of the "solararm" name.

Click the "+".

Now you should see several new expandable directories underneath "solararm" such as "solararm_backup_data", "solararm_data", etc.

Click on the name "solararm_data" - DO NOT just click on the "+" next to "solararm_data".

If you correctly click on the name, the solararm_data directory and all its subdirectories should be highlighted in a gray color.

Now that you have selected the solararm_data table, you can run an SQL (Structured Query Language) query on it.

Along the top horizontal edge of the screen, you should see several tabs called "Structure", "SQL", "Search", etc.

Click on the "SQL" tab.

You should now be looking at a large blank text area wherein you can type an SQL query.

Copy the following SQL query and paste it into this text area.

SELECT element_data.plate_no, plate_type, ratio_data.row_index, ratio_data.col_index, result.reading, result.ratio_to_standard1, result.ratio_to_standard2, element_data.atomic_symbol, pos, ratio, iron_nonstandard, copper_nonstandard, data_id

FROM element_data, plate_data, ratio_data, result

WHERE (element_data.plate_no = plate_data.plate_no AND element_data.plate_no = ratio_data.plate_no AND element_data.plate_no = result.plate_no AND ratio_data.row_index = result.row_index AND ratio_data.col_index = result.col_index)

ORDER BY data_id ASC

On the bottom right you should see a button called "Go", click this when you have pasted the SQL query into the text area.

If you get an error, it is very likely you did not copy the query in its entirety.

The query itself should take less than a second to complete.

If it takes longer than 10 seconds, odds are the query was incorrect.

You should now be looking at a table with headers such as "plate_no", "plate_type", "row_index", etc.

Above this table, you should see a yellow-highlighted line that says something like:

"Showing rows 0 - 24 (117900 total, Query took 0.3022 seconds.)"

Your screen should have a number equal or greater to the "117900" part of that line.

If this is not the case, it means some data was deleted from the database OR the query was not correct.

At the bottom of the screen you should see several buttons like "Print", "Copy to clipboard", "Export", etc.

Click on "Export".

You should now see two sections called "Export method" and "Format".

Under "Export method", select "Custom - display all possible options".

Now you should see several more sections.

I will go through each of them and tell you the setting it should be set to.

Some of these settings may already be correct by default.

Under "Format" select "CSV". NOTE: "CSV" stands for "Comma-Separated Values" which is NOT the kind of file we want.

We select the "CSV" option regardless because it enables some settings that we will need.

We will use the space character instead of the comma to separate our columns.

Under "Rows" select "Dump all rows".

Under "Output" unselect "Rename exported databases/tables/columns".

unselect "Use LOCK TABLES statement".

select "Save output to a file".

for "File name template:" write "@TABLE@" and select "use this for future exports".

for "Character set of the file:" select "utf-8".

for "Compression:" select "None".

unselect "View output as text".

make sure "Skip tables larger than ____ MiB" is left empty.

Under "Format-specific options:" change "Columns separated with:" to " " (a space character).

change "Columns enclosed with:" to "" (no character).

change "Columns escaped with:" to "" (no character).

change "Lines terminated with:" to "AUTO".

change "Replace NULL with:" to "NULL".

unselect "Remove carriage return/line feed characters within columns".

unselect "Put columns names in the first row".

That should be all the settings.

At the bottom of the screen there should be a "Go" button.

Click this and you will be prompted with a standard download-a-file dialog.

Find the directory you want to save the file to (probably the same folder that contains the ".jar" file).

Rename the file if you want.

It does not matter which file name you choose, but you MUST NOT change the file extension.

All extensions other than ".csv" will not be recognized by the program.

Once you enter the desired file name and find the correct directory, click "Save" and wait for the file to download.

Now you have the custom "CSV" file.

==========

RUNNING THE PROGRAM:

The program should be in the form of a ".jar" file - the file extension should be ".jar" - and it should reside in the same folder as the input data file.

Double-click on the ".jar" file.

You should now see a small dialog titled "Plate Data Formatter".

It should have four different options for the user.

I will go through them and explain their effects.

"Input file name:"

This is the file name of the input data file (the ".csv" file).

This file MUST be in the same location as the ".jar" file.

"Lowest acceptable rating:"

This is the lowest acceptable rating a "valid" plate can have.

This can be a decimal or integer value.

The rating is determined by finding the highest value divided by the average of either the left or right standard.

A rating of 1 for a red-lead plate would mean the plate has at least one cell wherein the value of the cell's reading

is the same as the average of the iron standard.

This means that cell's reading is "as good as" the average of the iron standard on that plate.

"Highest acceptable coefficient of variation in %:"

This is the highest acceptable coefficient of variation a "valid" plate can have.

If you are unfamiliar with the term "coefficient of variation", it - for our purposes - is the same

as "relative standard deviation".

This threshold is only used on the "high" standard of a plate.

A plate will always have two standards where one of them is higher than the other.

The low standard is more susceptible to high coefficients of variation so it is not checked.

If the high standard of the plate has a coefficient of variation lower than the threshold, it is considered "valid".

"Show target elements dialog:"

If this option is selected, a dialog titled "Select Target Elements" will appear during the formatting process.

This dialog will allow the user to select the "target elements" via a list of buttons with atomic symbols next to them.

When the user clicks the "Search" button on this dialog, the program will create an additional file called "target_cells.txt".

This file will contain all plates and all cells that contain the selected elements.

When you are finished setting the above options, you can run the program by clicking the "Format" button at the bottom of the dialog.

NOTE: if you run the program a second time, all the output text files in the programs folder will be OVERRIDDEN.

If you do not want this to happen, move the output text files to a different folder WITHOUT the program executable file.

The "Clear Fields" button will empty the fields of the first three options and unselect the fourth.

The "Instructions" button will display the contents of a text file called "READ_ME.txt" in a dialog.

This text file contains instructions on how to use this program.

==========

Appendix 2: Definitions

"Plate type" – in this report this term refers to which electrical lead (red or black) was on the physical plate when the data was collected. The phrase "black type plate" means the plate was run with a black lead. Similarly, "red type plate" means the plate was run with a red lead.

"Plate number" – the phrase "plate number 100" can refer to one or more "data sets" of information on the readings of one or more plates. The number of data sets depends on how many times a plate for a

specific plate number is uploaded. With every upload, a new set of data for a 36 cell plate is added to the database with the corresponding plate number.

"Data set" – this term is used to prevent possible confusion and algorithmic errors from the use of ambiguous term "plate". A plate is identified by a number, like "plate number 100". However, this plate can have multiple data sets associated with it since plates can be run and uploaded multiple times. Each data set contains all the data for a single PHYSICAL plate. A data set does NOT contain information on all data sets associated with a plate number. Since most plates are run at least twice, once with a red lead and once with a black lead, most plates will have at least two data sets associated with it. This program does NOT use plate numbers to determine when to create a new Plate object. Instead, it uses the unique "data_id". See the following definition to learn more about this term.

"data_id" – this refers to a UNIQUE identifier in the form of a large positive integer. This number is determined when the data for a data set is uploaded to the database. This number is used by this program to differentiate between data sets. If, for whatever reason, the data_id is NOT unique, this program would fail to create separate Plate objects for the data sets with the same data_id.

"Rating" – this term means two different things depending on whether it is used in reference to a Plate or to a Cell object. For a Plate object it represents the highest value obtained from taking a single NONSTANDARD cell reading and dividing it by the average of the appropriate standard column. For a Cell object it represents the reading of the cell divided by the average of the appropriate standard column. The "appropriate standard" depends on the plate type. If the plate type is black, then the copper standard should be the divisor. If the plate type is red, then the copper standard should be the divisor.

"Rating threshold" – this term represents the ratingThreshold variable used by the program to determine whether a Plate or Cell object is good enough. Currently, a rating threshold of 1 should give roughly 70 valid plates.

"Coefficient of variation" – this is the result of a mathematical formula applied to all readings down a column of cells. For our purposes it is the same as the "relative standard deviation". The coefficients of variation of the standards are used to determine whether a Plate should be considered as valid. NOTE: for a single plate only ONE coefficient of variation of a standard column is used to determine if the plate is valid. More specifically, the HIGH coefficient of variation of a standard column is used. This is because the lower the voltage readings on a physical plate the more likely small experimental inconsistencies create large outlier readings that cause the coefficient of variation to be misleadingly high. A relative standard deviation of 50% seems to provide a reasonable number of valid plates.

"Inconsistent elements down a column" – this phrase means that for all cells of a particular column, the combination of elements for each cell was NOT identical to all other cells in that column. Plates that have this property are listed as "problem plates".

"Inconsistent element ratios down a column" – this phrase means that for all cells of a particular column, the ratios of the elements for each cell of the column were NOT identical. Plates that have this property are NOT listed as "problem plates". Rather, a boolean data member of the Plate object is set to true and when the information for that Plate object is printed to a file, a warning is printed along with it.

"Problem plate" – this refers to the boolean data member of the PlateDataFormatter class that is used to determine whether the current Plate object is a "problem" or not. If it is a "problem", then the Plate object is excluded from the goodPlatesVector and included in the problemPlatesVector. Vice versa if it is

not a "problem". If any of the following conditions are "true", then the current Plate object is considered a "problem".

1. The rating of the plate is less than the rating threshold.
2. The sample coefficient of variation of the high standard was higher than maximum allowed coefficient of variation.
3. The wrong standard (copper or iron) had a high or low reading for the current plate type. This catches plates that were somehow "flipped" so that the averages of the standards did not match the expected results for the elements of those standards with regard to the current plate type.
4. The plate has inconsistent element combinations down a column. This is occurs when a plate is turned 90 degrees. The result is that the standards occur across the upper and lower rows rather than the left and right columns.

Appendix 3: Miscellaneous Information

Preface: while the name "Miscellaneous Information" may make you think this information is of less importance than some of the other content of this report, I will assure you this is absolutely not the case. The following are strange aspects of the program that, without the knowledge I am about to provide, seem to make no sense. Therefore, it is imperative that anyone who needs to fully understand this program is aware of the following aspects and problems.

1. There are several data sets in the data base that are missing a combination of the following cell readings: the bottom leftmost, the bottom rightmost, or the ENTIRE bottom row. As a result, when calculating values for the averages of the standards and the coefficients of variation, these plates may have INCORRECT values. Since the only semi-consistent pattern here was that the copper standard's last cell reading is frequently left off, the program only looks at the first 5 cells for that copper standard column rather than all 6 cells. If, for whatever reason, the iron standard is missing its last cell or the entire bottom row is missing, then the program will fail to correctly calculate the averages and coefficients of variation. Consequently, the readings for Plates and Cells of Plates with this issue will be skewed as well.
2. There is a fair amount of redundant code or optimizable code. This is the result of having an evolving end goal for the program while it was being created. This means that throughout development new features were continually conceptualized and attached. Some of the redundant or optimizable code is as follows:
    a. In the PlateDataFormatter.java file:
        i. the calculateSampleStandardDeviationOfStandards(), calculateCoefficientsOfVariationOfStandards(), and calculateCoefficientsOfVariationOfNonstandards() functions could be combined since the calculateCoefficientsOfVariationOfNonstandards() function does a very similar operation as the previous two functions. The primary difference is that the calculateCoefficientsOfVariationOfNonstandards() function applies to the inner four columns.
        ii. createDataMiningFile() and factorOf100Plate() functions are outdated and unused. I left them in the file since they might be of use to someone at some point.
    b. In the PlateDataMiner.java file:
        i. eliminateFactorsOf100() function is outdated and unused. As with the previous point, it is still in the file since it might give someone an idea at some point.

   c. Technically, it should be possible to optimize some of the code that prints content to the various files as there are several code sections that are very similar to each other. However, this would probably be fairly time-consuming and would not provide any new features that a user would notice. I would put this as the lowest priority.
3. For some reason, the ratio values in the database are incorrect for some plates. These ratios are the result of taking a cell's reading and dividing it by the appropriate average of a standard. Some of these ratios are off by a factor of 100 which throws off many of the calculations done in this program. To solve this issue, every single Plate's ratio values and ratings are recalculated via the calculateRatios() and calculateRating() functions in the PlateDataFormatter's handlePlate() function.