B. COMP DISSERTATION

Rapid Hypothesis Testing and Exploration

By: Ang Yan Sheng, Mark

Department of Computer Science

School of Computing

National University of Singapore

2014/2015

Project Code: H114380

Project Supervisor: Dr. Wong Lim Soon

Deliverables:

Report: 1 Volume

Program: 1 Zipped File

Table of Contents

Abstract

Statistical testing is a powerful tool used to strongly prove observed trends and users' assertions using real-world data. With more data being widely available, there is a great motivation to harness it. For example, much work is done in the medical field to discover patients' responses to various drugs, therapies, and identify groups within the population which require the most research attention.

Unfortunately, the processes that trained data analysts perform in order to make valid conclusions are not clear to the untrained user. Issues within the statistical testing realm like the handling of outliers are foreign concepts to people who want to analyse data, but did not undergo the necessary training to avoid the risk of having their own data mislead them. Untrained users, not understanding the importance of initial data exploration, would not know if there are any confounding data variables that could influence each other's behaviors, and come to inaccurate conclusions about their data.

My program aims to not only expose simple statistical analysis to untrained users, but automatically handle outliers within the data used, and give users some insight into possible confounding variables that correlate with observed test statistics.

Subject Descriptors:

Descriptive Statistics

Linear Regression

Keywords:

Linear Regression, Human-Computer Interaction

Implementation Software and Hardware:

Windows 8.1, Python 3.4.1, PyQt 4.8.6

Acknowledgement

I thank God for seeing me through to finish this project.

I would also like to thank my supervisor, Dr. Wong Lim Soon, for keeping me on track and bring the project to where it currently is.

I would also like to thank Dr. Li Jialiang, whose invaluable assistance filled the gaps in my statistics knowledge. His familiarity with how data analysts perform data analysis, being a data analyst himself, shaped the vision I had for my program's user experience.

A heartfelt thanks goes to my family for supporting me through the whole project, even during the times that I felt daunted by the challenges I faced in making the project happen. They have, as they always had, my gratitude for keeping me going amidst every challenge.

Abbreviations / Terminology

UI – User Interface

ANOVA – Analysis of Variance

RPN – Reverse Polish Notation

1. Introduction

1.1 Statistical Testing for the untrained

Statistical testing arose from the inability to collect complete population data against the need to assess those populations. For example, a bottling factory might want to know if its bottles are being made of a certain thickness or more. Since it produces tens of thousands of bottles daily, it would not be feasible to measure every bottle. Instead, it would pick a reasonable number of bottles at random, measure them, and then perform a statistical test on these bottles, assuming that their thicknesses can represent all the bottles' thicknesses.

In order to extrapolate statistics from the sample to the population, some leeway is given such that the sample's observed statistics can possibly differ from the actual population's statistics. In many tests, this leeway comes in the form of the assumption that the sample's statistic, or in our case, the bottle's thickness, follows some model of random distribution.

Statistical testing helps people to assess population statistics quickly, but not measuring the entire population opens up a different set of problems. For example, the sample must fairly represent the population. For example, if all the sample's bottles came from the same machine, then the sample cannot represent the factory's entire output since the other machines' outputs are not reflected in the sample.

The understanding of these concepts does not come naturally, and must be taught to people who wish to obtain solid results from data analysis. However, not only will some people lack the resources to undergo this training, they may even ignore the importance of verifying that their data can represent the population properly.

1.2 Typical problems faced in statistical testing

The data analyst faces a host of unknowns when he first receives his data, which he has to understand before he can make any concrete claims using the data.

To illustrate this, let's take a hypothetical case of Company ABC being accused of being sexist, paying women unfairly less than men. In response, the company commissions a data analyst to sift through its employees' data and investigate the allegation. The naïve

analyst, checking to see if men's salaries were indeed greater than women's salaries, would simply compare men's average salaries against women's average salaries in the company, and find that men's salaries are indeed greater than women's.

However, the naïve analyst has not considered many things which could confound his analysis. His analysis assumes that all men and women have equal demographic distributions (Aside from being men and women). For example, say managers are paid much more than other occupations in this company, and the majority of managers in this company are men. In the light of this knowledge, it is possible that men are paid more on average because they are managers, as opposed to simply being men.

Furthermore, the possibility of anomalous or even incorrect data points can skew the overall assessment. It would be prudent in many situations to consider the removal of such outlying points thus.

1.3 Correlation: A data feature

One of the aims of this project is to automate discovering and isolating possible influences of variables within the population on the observed test statistics. To achieve this, we focus on two key features: Linear regression and analysis of variance (ANOVA).

Linear regression is the assessment of the likelihood of two variables sharing a linear correlation. It is used to determine if two continuous variables correlate with each other. For example, we could use linear regression to determine if salaries of employees increase along with age (Or even decrease with age!)

ANOVA is the assessment of the likelihood of different sample groups coming from the same (or different) test statistic distributions. For example, we could use an ANOVA to determine if clerks, managers, and supervisors all share the same distributions of salaries.

By using these tests, we are able to model the relationships between variables against the test statistic.

Once we model these relationships, we can control for them by adjusting the observed statistics according to their respective controlled variable. For example, if the observed value comes from a clerk, and the control occupation is 'Manager', we adjust the statistic to simulate the test statistic as if it had come from a manager.

It is important to note that *correlation does not imply causation*. This project does not attempt to identify causal relationships; rather, by discovering correlations, the user can then be prompted to perform further investigation for causality.

1.4 Overall objective

Given the increasing ease of data collection, many people who have not received data analysis training stand to benefit from leveraging on statistical analysis to make informed decisions using their data. My project thus aims to expose statistical analysis to people who do not know how to perform statistical testing properly.

Since statistical data itself is prone to possible issues such as outliers, my project aims to automatically handle them.

As a step towards automating descriptive data analysis, this project also aims to help the user identify correlations between variables in the data against the test statistics, and adjust the test statistics to account for these correlations.

To summarise, my project aims to:

- Expose statistical analysis to the untrained user.
- Automatically handle outliers in the user's data.
- Provide simple variable elimination functionality.

2. Design / Scope

This section will discuss the overall design and architecture of the program, as well as decisions made in the course of the project.

2.1 Overall Design

Care was taken in designing this program to keep the number of UI windows to a minimum. Please see Appendix A for an overall architecture diagram, as well as Section 5 for more discussion on the design of the user interface.

2.2 Hypothesis Input

Since our target user is not trained in data analysis, he might not know how to translate his inquiry to fit a statistic test, or what test to use for that matter.

Thus, my program set out to abstract this problem away from the user by exposing means difference tests by allowing the user to input 2 infix test expressions in the form of a comparison statement; one for each sample. For example, the user may type in not only type in “Salary” into the statement, he could also type in “Salary * 1.25” or “Salary + (Bonus / 12)”. Note that up to one side may have a fixed value; the test will be a 1-sample location test in those cases.

By using simple mathematics to hide the actual test selection process, the program aids in the hypothesis input process by providing the user a more intuitive way to input his hypothesis.

Due to the nature of the test expression being limited to having only two sides, the project handles one-sample and two-sample location tests.

This project is limited to comparing of continuous variables as the proposed variable correction method cannot be directly applied to categorical variables; more work can be done to extend the method to handle categorical variables, possibly via machine learning methods.

2.3 Display of results

Throughout the project, two modes of display were considered: Text and Visual Graphics. The final product uses both modes for different aspects in the program.

Initially, a pure-text display was planned, due to the challenges associated with having to implement the graphs correctly. During the course of the project, however, the amount of data to be shown to the user began to grow (With stratified testing, many tests with many results would have to be shown).

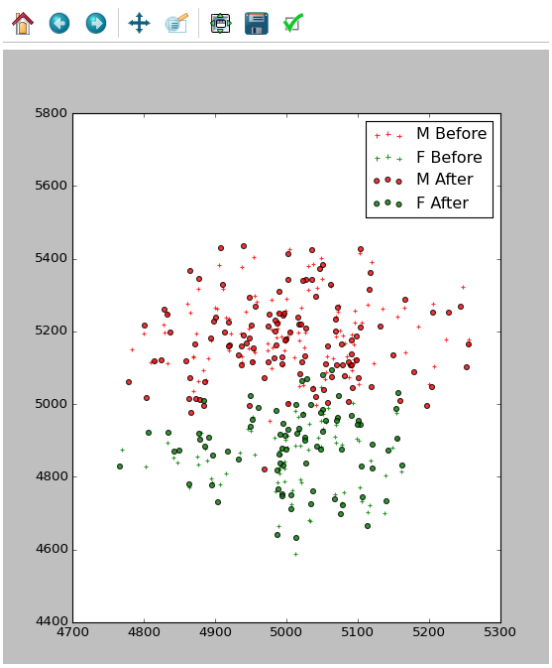


Figure 1: Example graph showing test statistic distributions before and after correction

It was discovered then that a pure-text display would either show the user too much text (and numbers) for him to make sense of, or be forced to hide potentially interesting information from the user. Hence, the decision was made to implement visual graphs.

To display the test results, a graph showing the distributions of the test statistics is plotted for the user to view and compare. For example, this graph shows the distribution of test statistics for a Dependent T-test style of hypothesis.

Simpler features such as verification reports are displayed via text browsers.

2.4 Demonstration of correlation

Demonstrating the concept of correlation to untrained users was a little tricky, since it has to be assumed that the user does not understand the concept of correlation, let alone appreciate the use of correcting for variables using their correlations to the test statistics.

To guide the user towards understanding and appreciating correction of test statistics against some variables, a text print summarizing the effects of a variable against the test statistics and a series of graphs essentially telling the story of how the program performs this variable elimination was produced within the program. This was done to not only demonstrate the correlation and its possible influence on the test statistics, but to teach users about how variables can appear to influence the test statistics in a narrative

manner. More details on this can be found in Section 3.4: “How to display variable correction?”

2.5 Scope

As mentioned in Section 2.2, my project is limited to analysis of continuous variables. The decision to do this was made in order to fully demonstrate variable correction.

In addition, my project aims to demonstrate correction of individual variables, as opposed to handling multiple variables simultaneously. While the benefits from performing the latter are appreciable, they might not be easily demonstrated to the untrained user. Furthermore, for basic analysis purposes it would be less confusing to show elimination of individual variables and allow the user to find the variable which seems to exhibit the largest influence on the test statistics. More discussion on this topic can be found in Section 5: Recommendations for future work.

One of the features initially planned for the program was to automatically stratify the data, splitting the data into many sub-categories and testing each sub-category in anticipation of each sub-category having different distributions. For example, men and women may appear to have similar heart attack history distributions, but upon further splitting of data we may find that men and women *smokers* (*and non-smokers*) have different heart attack history distributions. To discover these relations, the goal was to split all the data by every category as far as possible whilst maintaining the validity of each sub-group to perform a test on it, and perform a test for each sub-group to produce a large collection of tests for further analysis.

However this feature was eventually dropped in favor of displaying individual data points as either a scatter or a histogram due to complications discovered during the project. This feature is discussed more in Section 3.1.

3 Concepts

This section discusses the statistics concepts used in the project.

3.1 Outlier Handling

One of the aims of the project is to automatically discover and eliminate outliers.

To identify outliers, Grubb's outlier test is used iteratively along each continuous variable in the dataset, until no more data entries are picked up by the test.

Grubb's test is an outlier detector test which operates on one outlier at a time. For each possible outlier, it performs a one-sided location test on either the minimum value or the maximum value of the dataset against the mean of the dataset, with the null hypothesis being that no outliers exist, and the alternate being that at least one outlier exists (I.e. The target value, and possibly others). The target value is considered an outlier if the confidence value is significant enough (The threshold used is <5% confidence to the null hypothesis).

Basically, if the target value is considered too far from the rest of the data, then it's considered an outlier.

Once the outlier is detected, it is removed from the dataset, and the test is repeated sans the removed outlier. In this implementation, the minimum value is checked first, followed by the maximum value. This is repeated until either not enough data points exist to perform the outlier test meaningfully, or no outlier is detected on both the minimum and maximum values.

Currently, the outlier removal operates on all the continuous variables within the dataset. Grubb's test assumes that the data is normally distributed. This poses some challenges since this project aims to detect multiple normally distributed populations in the data.

For example, men and women might present different salary distributions. If the test is performed without checking for men and women distributions, the test might assume too many points to be outliers because they would all be too far away from the mean (which is somewhere in between the men's and women's distributions) anyway.

Future developers might be able to overcome this problem by detecting data clusters using an unsupervised machine learning algorithm such as hierarchical clustering, removing clusters with very small sizes (such as single-element clusters). Such an

implementation would resolve most of the issues faced when using a limited technique such as Grubb's test.

Nonetheless, Grubb's test was chosen for its ease of implementation, and as no specific outlier detection test was included in the standard Python libraries.

3.2 Discovery of correlations

Discovering correlations of the variable against the test statistic is done using either of two tests: Linear regression and the one-way analysis of variance (ANOVA). If the confidence value of the test is sufficiently significant, the user is informed of a likely correlation between the variable and the test statistic. Linear regression is a well-known technique of establishing correlations between variables, and ANOVA is a well-known technique to determine if a set of samples do (or do not) have identical distributions.

If the variable to be tested against the test statistic is continuous (Like height or weight), a linear regression test is performed between the variable against its corresponding test statistic. The linear regression test's p-value is used to determine if said variable exhibits a strong correlation with the test statistic.

If the variable to be tested against the test statistic is categorical (Like gender or occupation), a one-way ANOVA is performed instead. As with the linear regression test, the p-value is used to determine if the occurrence or non-occurrence of different variables within a column can influence the test statistic distribution.

3.3 How to correct for a variable?

Once a correlation is discovered, the program will attempt to correct for the variable by simulating the test statistics as if they all had the same variable category / value (I.e. As if all entries have same height/gender).

For numeric values, the program consider the variable in question to share a linear relationship with the test statistic. This linear relationship is discovered using linear regression on all the variables against their corresponding test statistics in the entire dataset, and the relationship modeled using the following equation:

$$T = mV + c$$

Where T is the test statistic, m is the linear correlation between the variable and the test statistic, and V is the variable.

C is an arbitrary value that is unique to each entry due to the stochastic nature (randomness) of the measured data.

The linear regression discovers the value of m.

Now for every test statistic T_0 and its associated V value V_0, we apply the same relationship, but this time use our pre-calculated m value on the equation.

$$T_0 = mV_0 + C_0$$

To perform the correction, we replace the V_0 value with some arbitrary common value V_1 (In this project, we use the mean of all the V_0 values in the entire dataset), holding m and C_0 constant. Once done, we calculate corrected test statistic T_1 using the formula:

$$T_1 = mV_1 + C_0$$

For categorical variables (Gender, Smoker/Non-smoker), the formula is different since linear correlation cannot be established on distinct categories. For every test statistic T and its associated V category, we model their behavior using the following equation:

$$T = x * \text{Mean}(T(V))$$

Where x is an arbitrary value that is unique to each entry due to the stochastic nature (randomness) of the measured data, and Mean(T(V)) is the mean of the test statistics for the entire subpopulation of category V.

For some test statistic T_0, we find its corresponding x value by firstly dividing T_0 by its corresponding $\text{Mean}(T(V_0))$.

$$X_0 = T_0 / \text{Mean}(T(V_0))$$

Once X_0 is found, the corrected value T_1 is found by multiplying X_0 by some arbitrary common value(In this project, this is the mean of the test statistics in all the data), producing the final formula:

$T_1 = X_0 * \text{Mean(All T)}$

While this model is simple to implement, it is prone to issues. For example, this model does not take into account negative values for individual entries whose sub-population means are positive, which may cause X_0 to become negative. The correction performed on negative X_0 values would simulate wildly inaccurate corrected statistics. A more robust model might take into account the variances of the sub-populations' test statistics and the differences of the observed test statistics from their respective sub-populations' test statistic means. This can be accounted for in future work.

3.4 How to display variable correction?

Once a correlation is discovered and accounted for, the user will be shown the data points before and after correction to demonstrate the strength of correlation between the variable and the test statistics.

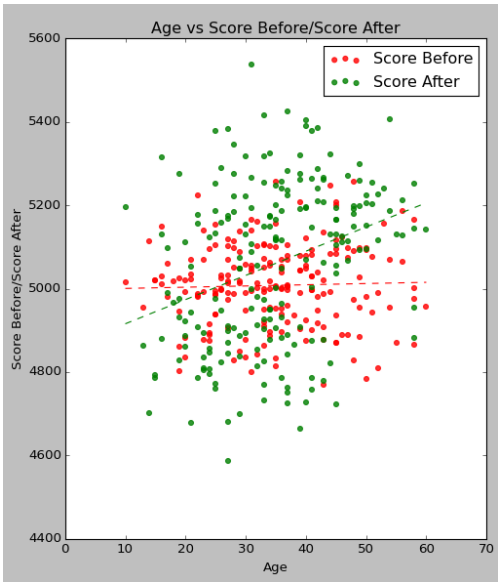


Figure 2: Relation of entries' variable against their corresponding test statistics.

To facilitate this, a total of 3 graphs are displayed.

The first graph shows the relation of the variable in question against their corresponding test statistics. If the variable is continuous, a linear regression is performed as per Section 3.3, and the discovered equation is shown as a dashed line. If the variable is categorical, the means of the categories' test statistics are displayed instead. This graph will demonstrate how the occurrence or behavior of the variable can appear to influence the test statistics. For example, if taller people are found to earn more money, this section will show it.

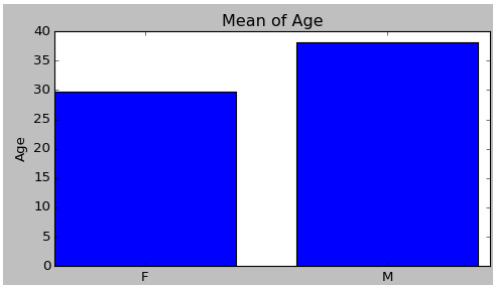


Figure 3: Graph showing the different age means of men and women. A large mean difference indicates a skew in the distributions of this variable.

The second graph shows the variables' distributions within sub-groups of data split by a different user-defined variable (the splitting variable). This graph will show any skew of the distribution of data between subpopulations.

If the splitting variable is continuous, the sub-groups will be split into 3 groups: Lower quartile, Inter-quartile range, and Upper quartile. If the splitting variable is a text variable, the sub-groups will be split by the categories within the variable. For example, if the user checks for gender, and women are significantly shorter than men, this section will show it.

If the variable in question is continuous, the means of this variable in the different sub-groups are displayed. If the variable is categorical, then the different counts of categories within this variable is shown instead. For example, if the user checks for smokers/non-smokers, and more men smoke than women, this section will show it.

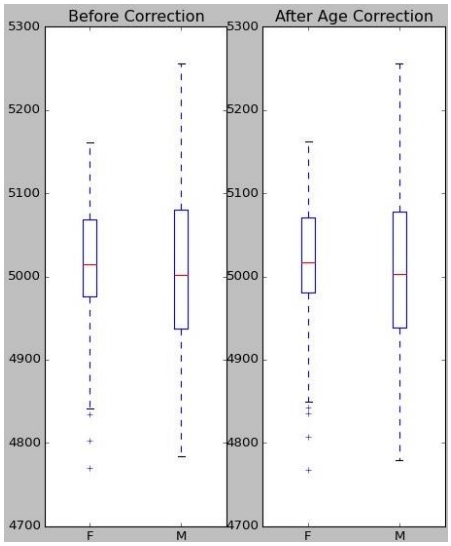
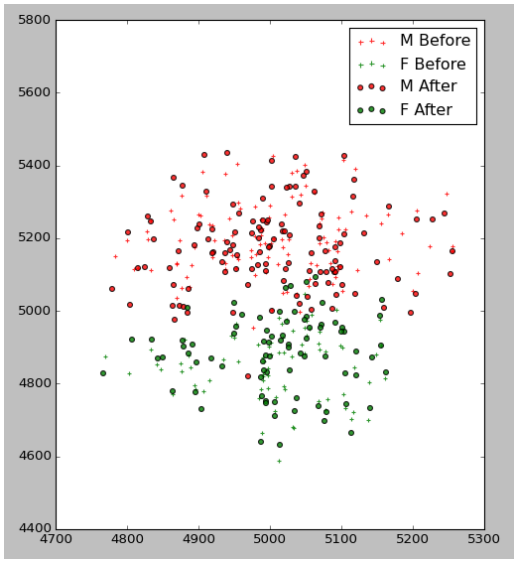


Figure 4: Graph showing distributions of a 1-sample test statistic before and after correction.

Finally, for the demonstration of the potential correction of the influence of this variable against the test statistics, the third graph shows the data points before and after correction will be generated.

For one-sample tests, this is given as a list of histograms before and after correction, with each histogram generated by dividing the data by a user-defined category.

For two-sample tests, this is shown as either 2 lists of histograms, corresponding to the Left-Hand Side and Right-Hand Side. If the data points from both sides are exactly the same, a graph with X-axis corresponding to the Left-Hand Side and Y-axis to Right-Hand Side will be used, and two scatter plots for before and after correction will be generated (In this graph, 'Before' values are displayed using '+' symbols and 'After' values are displayed as circles).



3.5 Other features

Figure 5:Graph of values before and after correction for a 2-sample test.

Other auxiliary functions such as simple sample data verification are performed, such as checking if there is partial overlap between the two input test samples. Other functions such as checking if the distributions of the data were as the statistical test required (Goodness-of-fit tests) were implemented, but not used due to the shift away from hypothesis testing.

The program also determines what statistical test best fits the user's input hypothesis, although the test is no longer actually performed as explained in Section 3.1.

This information is displayed in the Results Explorer, under the 'Overall' tab.

4 Design Considerations

4.1 Challenges

There were some challenges realized from the beginning of the project, as well as in the course of it. This section discusses these challenges, and how they were overcome.

Firstly, the student working on this project was a Computer Science major with little background in statistics. There was much ground to cover regarding the statistics concepts of the project, as well as discovering what design would best help untrained data analysts in learning more about his data. To overcome this, the student worked closely with NUS's Statistics Counseling Centre to ensure that his work not only applied the statistics correctly, but also to obtain feedback to further improve the flow of the program.

Furthermore, the initial goal of stratified testing was replaced with correcting of the test statistics against variables. A good amount of the work done to support stratified testing was no longer useful to the project. In anticipation of such problems, a good portion of the code was written to be easily re-usable and extendable, thus becoming an appreciable demonstration of the benefits of good software engineering practices. Nonetheless, this shift mid-project left some bugs and unexpected behaviors unresolved due to the lack of time to fully account for all of them.

4.2 How to design for the untrained?

The dual goals of exposing statistical analysis to untrained analysts and correcting for potential influences of individual variables on the test statistics presents a unique problem:

How can the latter goal be not only achieved, but presented to the untrained user in a meaningful and easily appreciable way?

To this end, a series of graphs were produced in the Results Explorer window, designed to 'tell a story'.

Say the program is testing if varying the height can potentially influence the test statistic. The first graph would show how the test statistics would vary when plotted against height. The second graph would show how sub-groups of data (Say, men and women) exhibit different height distributions. The final graph would show the test statistics before and after correcting for height.

In addition, a text browser verbally discussing the above three graphs' results is produced alongside the graphs.

4.3 Alternative Designs

4.2.1 Data Explorer

One feature considered early in the project was a data browser and explorer, giving the user a quick overview of the data with basic statistic features such as means and modes. The user would also be able to compare different subsets of data by viewing the different statistic features, as well as compute their respective test statistics. Its purpose was to allow the user to better understand the data before he makes his hypothesis.

However, this ran counter to our original goal of exposing statistical analysis to the untrained user, since even exploring data requires some degree of statistical knowledge to understand what the values mean. Furthermore, given limited time and resources, it was decided that it would be more prudent to focus on producing the correlation feature, which is one of the main goals of this project.

4.2.2 Stratified Testing

Another feature that persisted in the early stages of the project was the concept of stratified testing; That is, to re-apply the hypothesis on subsets of data, differentiated according to different data filters and restrictions. For example, rather than testing all the data, if there is a 'Gender' column with both Male and Female entries, two tests will be performed, one with male-only samples and one with female-only samples. This is

compounded with increasing numbers of columns, requiring the program to split the data by potentially all the columns.

The motivation behind this concept is that with different sub-groups having different distributions, there should be some tests on sub-groups which exhibit significantly different distributions when compared side-by-side. For example, tall men smokers could have higher chance of death by heart attack than tall women smokers. The project would then find these tests and alert the user of these different distributions.

However, it was discovered mid-project that this approach may produce significant results from many tests (The test sample data, containing 6 variables, caused the program to perform more than 100 tests). At the number of columns increase, so too will the number of potential tests, and the sample sizes for each test on average will decrease given equal sample sizes. Due to the assumed random nature of data, with more tests done and reduced sample sizes per test, some tests will ultimately produce significant confidence values even if the distributions between the test samples are actually the same. Stratifying the data also increases the risk of encountering non-normal distributions due to simply not having enough data to fully represent these strictly defined sub-populations.

Simply put, if you perform enough hypothesis tests on different subsets of data, you will eventually find a test that produces statistically significant results, even if the samples are taken from the same population.

Furthermore, stratifying the data, while offering a stronger means of determining different sample populations, does not actually tackle the problem of figuring out which variables correlate with the test statistics. Ideally, different tests' statistics could be compared (For example, we could compare the test statistics for men-only and women-only test results). However, this opened up a host of issues that made using this approach too cumbersome to implement.

On the other hand, the alternative of using individual entries as data points was simpler to implement. Using individual entries as data points was also helpful in determining the distribution of the data, allowing the user to visually determine how the test statistics are distributed instead of showing test results under the assumption of some distribution

(which again may not be true for the subset of data, especially if the subset of data is of a small size).

Overall, it was found that stratified testing introduced a list of complications, making its implementation untenable within the time period given to this project. It was decided thus that a simpler means of display, eschewing hypothesis testing in producing the data points, would be used instead.

5 Implementation

This section discusses the implementation details of the project. Software and libraries used will be discussed here, as well as the data objects used and created to facilitate the project.

5.1 Software, Libraries Used

This project was programmed on Windows 8.

Python 3 was used as the programming language of choice in this project due to its ease of use and extensive library support, especially in the fields of statistical analysis. Most of the statistical tests used were just a library function away, and only the one-location Z-test and the Grubb's outlier test had to be implemented manually. As with many Python programs, Numpy and Scipy were used for data handling.

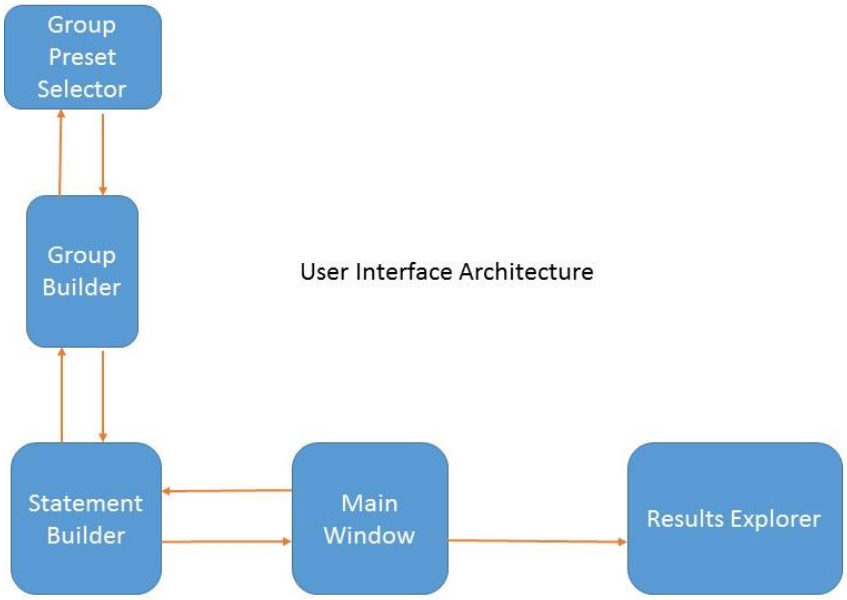
Anaconda's included Spyder Integrated Development Environment (IDE) was chosen for programming as the student was familiar with the interface, eliminating the teething problems associated with having to learn to code using an unfamiliar IDE.

For the GUI, Qt Designer was used to code all the statically generated front-end interface features, with conversion of the Designer's .ui files to Python code via the pyuic4 module. The GUI is run on Python using the PyQt4 library. Any dynamically generated front-end interface features (Checkboxes corresponding to categories in a column, for example, have to be generated on-the-fly) are manually coded into the pyuic4-generated python file.

Graphs were plotted using Python's Matplotlib library. Although other libraries like seaborn produce more visually pleasing plots and offer more powerful functions, Matplotlib

was used as the project did not require complex graphs, and to make any future conversion of the project into an executable file using a library such as py2exe easier.

5.2 Overall UI Architecture



The above diagram outlines the overall UI Architecture of the program. The individual interactions between the different components will be discussed in more detail in the upcoming sections.

Since the overall purpose of the project is to expose statistical analysis to untrained analysts, the windows were kept to a minimum, and as simple as possible for ease of use. In addition, the window hierarchy was kept simple to make the process flow as clear to the user as possible.

In addition, it was realized early in the project that each window should have a clear purpose (E.g. Statement Builder builds statements, Group Builder builds groups). If the UIs were to be combined to serve multiple purposes, this would lead to large modules which are not only unwieldy to manage, but could be difficult to extend should user requirements change (E.g. What if Statement Builder and Group Builder are combined in one large interface, but Groups are no longer needed? Removing one module is easier than finding out which codes need to be removed).

The overall layout for each UI window also complements the expected process flow of the program, such that the features which the user should use first are always placed higher up in the layout. This was done to ensure consistency and ease of use throughout the program as this matches with how users visually process information (Like reading a book, or a menu).

5.3 Notable UI Objects

This section discusses the functions of the UI windows and dialogs created for use in this project.

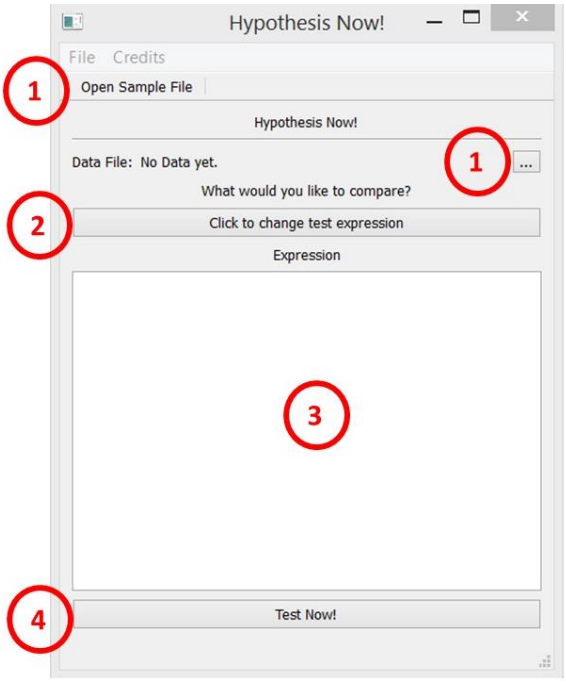


Figure 6 Main Window

5.3.1 Main Window

The Main Window is the first screen the user will see, and serves a few functions:

1) To load the sample data (a CSV file with headers). To load the file, click on the 'Open Sample File' button (1) and select the file using the file dialog. Note that the other functions cannot be performed without loading of the sample data.
2) To store the statement. To go to the Statement Builder dialog, click on the button labelled 'Click to Change Test Expression' (2). If no sample data is loaded, an error message will be displayed instead.
3) To display the loaded statement. Once a statement is loaded, it will be displayed in the statement display box (3).
4) To call the Result Explorer. To call the Result Explorer, click on the button labelled 'Test Now' (4). If no sample data is loaded or no statement is loaded, an error message will be displayed instead.

This module is named main.py, and is the root module within the package.

5.3.2 Statement Builder

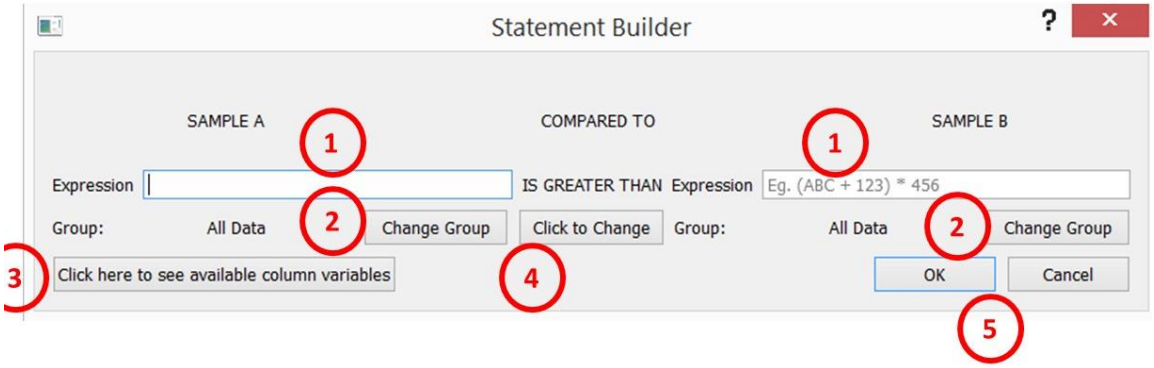


Figure 7: Statement Builder window

The Statement Builder is the second screen the user will see, and serves as the interface for the user to translate his inquiry into a comparison expression for testing purposes.

There are a few functions that the user can perform in this window:

1) Input a simple math expression with which the test statistic is calculated, using the text input box (1). This function is to be performed for both sides of the comparison expression.

   The input expression can be either a function of a single or multiple variables in the data, or a fixed value, although both expression boxes cannot contain fixed values simultaneously (There wouldn't be anything meaningful to test then)

   In addition, a basic auto-completer is implemented on both boxes. The auto-completer's dictionary comprises of the names of the columns in the data, and will only prompt the user to auto-fill to those strings only. Currently it cannot handle auto-completing for, say, the last word of the string.
2) Call the Group Builder dialog to select categories and rules to filter the data in one side of the comparison expression. Note that this can be performed for both sides of the comparison expression. Call the Group Builder dialog by clicking on the 'Change Group' button (2).
3) Call a simple dialog displaying the variables that may be used within the test expression (3).

4) Toggle the comparator used in the text expression (4).

   Note: This feature is currently non-functional since hypothesis tests are no longer performed directly on the dataset.

5) The well-known ‘OK’ and ‘Cancel’ dialog buttons. ‘OK’ saves the test expression and group data in a Statement object (After checking if the relevant fields are valid) and exits the dialog, returning the user back to the main window. ‘Cancel’ discards all changes done in the Statement Builder and returns the user back to the main window.

This module is named StatementBuilder.py.

5.3.3 Group Builder

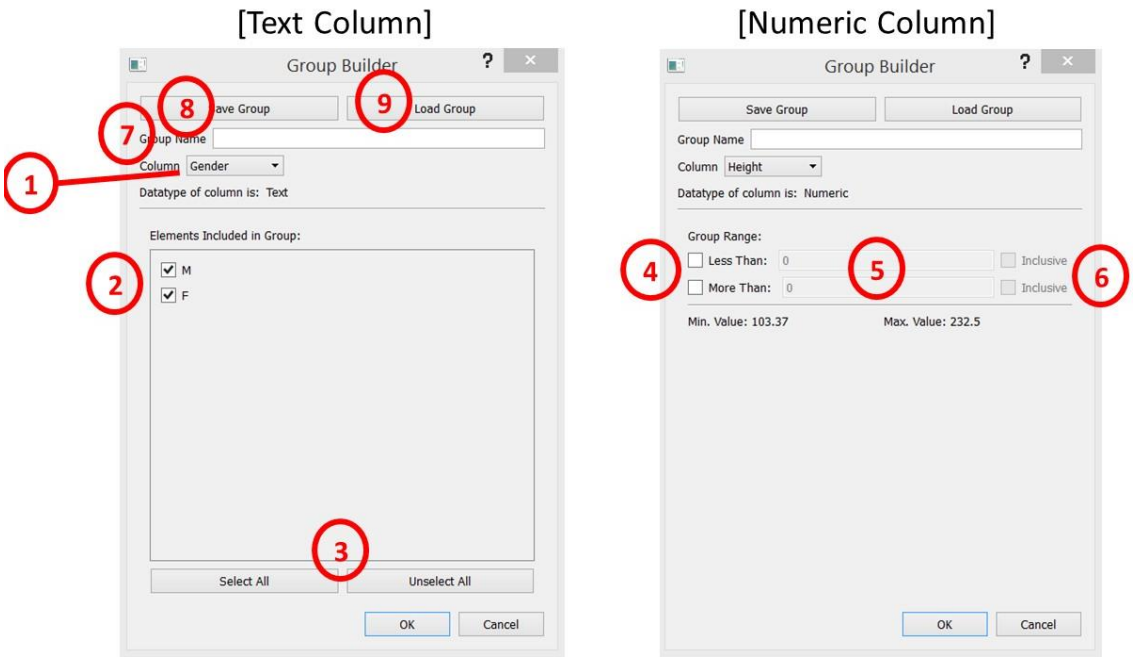


Figure 8: Group Builder window states. The window changes state depending on the type of data used in the column. If the column is categorical (E.g. Gender) the left window is shown. If the data is numeric, the right window (E.g. Age) is shown.

The Group Builder is the third screen the user will see, and is used to define the filters used to filter the dataset used for either side of the test expression. It is called from the Statement Builder window and upon successful exit, returns a Group object to Statement Builder if ‘OK’ was pressed.

The functions the user can use to create the group are as follows:

1) Select the column to setup a rule for. The column is selected using the column selection combo box (1).
2) If the selected combo is of type text, the dialog will show a list of possible categories to keep when filtering data using the group. These categories can be selected using the checkboxes (2). Alternatively, all checkboxes can be selected or unselected using the 'Select All' and 'Unselect All' buttons (3)
3) If the selected combo is of type numeric, the dialog will show a list of possible rules to apply.

   Currently, only 2 rules exist: 'Greater Than', which keeps data entries whose values are greater than its own input value, and 'Less Than', which keeps data entries whose values are less than its own value.

   The rules can be activated by checking their respective boxes (4), their input values can be input in their respective entry fields (5), and the option for the rule to include the input value itself can be toggled using their respective 'Inclusive' checkbox (6).
4) The group can also be named by the user via a text input (7). This name (ideally) serves as a title to remind the user of what subset of data the group refers to.
5) The user can save groups. Saving the group is done by pressing the 'Save Group' button (8).
6) The user can also load saved groups. Loading is done by pressing the 'Load Group' button. Pressing this button opens a dialog to select a previously saved group to load.

Saved groups can be modified by modifying the group, and saving it with the same name as the group to be modified.

There is currently no function to delete groups, although all groups are erased upon terminating the program. Group persistence after program termination is possible using python's pickle library, but opens up a range of problems that deemed working on this feature disruptive to the project.

This module is named GroupBuilder.py.

5.3.4 Results Explorer

Figure 9: Results Explorer window. The Text tabs are on the left, and the Graph tabs are on the right.

The Results Explorer is the final screen the user will see. It displays the statistical analysis performed by the program, and is called by the Main Window.

The Results Explorer window can be split into two sections, the text section on the left and the graph section on the right. For a list of diagrams illustrating the window's various tabs, please see Appendix A: Results Explorer Screens.

There are 4 tabs in the Text Section:

1) The overall tab. This tab shows the number of data points used in the test, as well as basic information about the data.
   Since this project was originally written with hypothesis testing in mind, most of the verification functions were written to verify the hypothesis tests before performing them. No hypothesis tests are actually performed now (No confidence values derived), but the verification tests are still performed.

2) The filter tab. This tab allows the user to further filter the data points used in the test for further analysis.

3) The Group Comparisons tab. This tab shows the textual report of any correlation that might exist between individual variables against the test statistics. The user can select the report of each variable using the variable selection combo box in this tab.

4) The Options tab. This tab allows the user to toggle removal of outlying data points, as well as select the column by which to divide the data into different categories of.

There are 3 tabs in the Graph Section, each corresponding to one part of the correlation report.

1) Part One shows the relationship between the data entries' variable with their corresponding test statistics. If both sides of the test are not fixed, both will be plotted on the same graph.
   This graph's purpose is to show if the variable correlates with the test statistic(s).
2) Part Two shows the means of this variable against each sub-group, categorized according to a column selected in the Options tab.
   If there are significant differences between the variable's behaviors between sub-groups, this graph will show it.

3) Part Three shows the test statistic(s) before and after correcting for the variable. If the sub-groups seem to be distinct before correction and appear to merge after, then it is suggested that there exists a strong correlation between this variable and the test statistic, and that differences between sub-groups' test statistics could be explained by their different variable distributions rather than the categories of the sub-groups themselves.

This module is named ResultsExplorer.py in the package.

5.4 Notable non-UI objects

The non-UI objects contain many variables which would be too disruptive to this report if discussed here. Please see Appendix B for a list of variables and functions used by the objects.

5.4.1 Data Manager

'Data Manager' contains all the static information obtained from pre-processing the data-set. It is generated using a special dmParser object (dmParser.py) called by the Main Window. The 'Data Manager' object's reference is then parsed over to every dialog and referenced as needed.

In this project, the only modifications performed on Data Manager after it is created are to its 'groups' dictionary, and only by the Group Builder dialog.

This class is defined within the Typedef.py module.

5.4.2 Statement

'Statement' is a command object used to communicate the hypotheses created using the Statement Builder to Results Explorer.

This class is defined within the StatementBuilder.py module.

5.4.3 Rule

'Rule' is an object that stores user-defined rules used to filter data entries.

Two different types of 'Rule' objects extend from the 'Rule' object:

1) RuleNum, which deals with numeric columns.
2) RuleText, which deals with text columns.

These two rules, although extending from a common Rule object, do not apply the software engineering principle of polymorphism due to the different natures of handling numeric and text data types. Although this was done for ease of development, this may not be a desirable state for future developers who would prefer to adhere to such software engineering principles for extension purposes.

RuleNum applies two rules:

1) Is the entry's variable greater than (or equal to) some input number?
2) Is the entry's variable less than (or equal to) some (other) input number?

This provides a simple way to filter values.

RuleText applies one rule:

1) Does the entry's variable exist in my set of variables?

This too is a simple method to filter values.

These classes are defined within the GroupBuilder.py module.

5.4.4 Group

'Group' is an object that stores 'Rule' objects and filter data entries by user-defined rules on each category.

This class is defined within the GroupBuilder.py module.

5.3.5 Test Logic / Correlation Report

'Test Logic' is the object responsible for performing the hypothesis tests. It would perform Z-Tests and T-tests, and recursively perform stratified testing, returning the complete list of test results. However, since the hypothesis testing is no longer actually done, this aspect of the 'Test Logic' is no longer used.

With the scope adjusted towards discovering correlations and correcting test statistics for individual variables, many helper functions within Test Logic were still applicable within this new scope. Hence the decision was made to keep the old Test Logic object (leaving the option to use stratified statistical hypothesis testing open for future work), and make the Correlation Report an extension of the Test Logic object.

Being the cornerstone of this project's logic, many functions are performed by this function, some of which are shared out to Typedef.py and HNStats.py modules. For a complete list of these functions, please see Appendix B.

The 'Test Logic' object is defined within the TestLogic.py module, and the Correlation Report object is defined within the CorrelationReport.py module.

5.4.5 HNStats / Typedef

Some scripts require common, typically lower-level functions such as verifying data-types, as well as global variables. Rather than redundantly copying all the functions and variables onto every script, the functions are written into a helper module, which performs

these functions on behalf of the caller scripts. In this project, HNStats.py contains the functions, while Typedef.py maintains all the global variables.

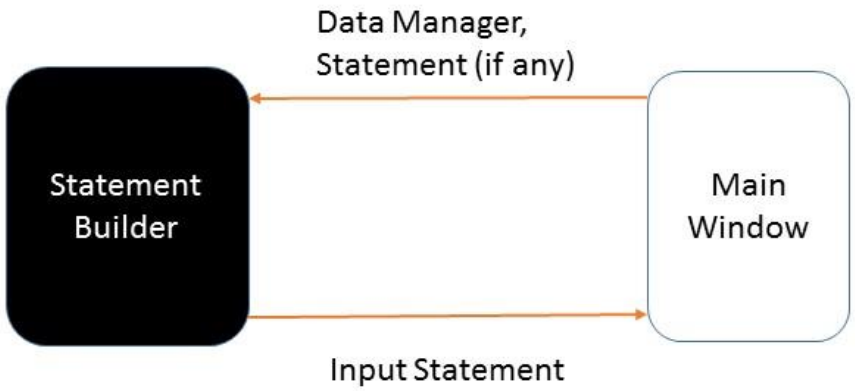
Note that HNStats and Typedef are not objects, but are a collection of functions (for HNStats) and variables (for Typedef). HNStats is found in HNStats.py, and Typedef is found in Typedef.py.

5.5 Notable interactions between objects

This section discusses general interactions between different objects within the project.

Note: For the following diagrams, the white object (also situated on the right) always calls the black object, not the other way round. (Or as they say in chess, 'White starts first')

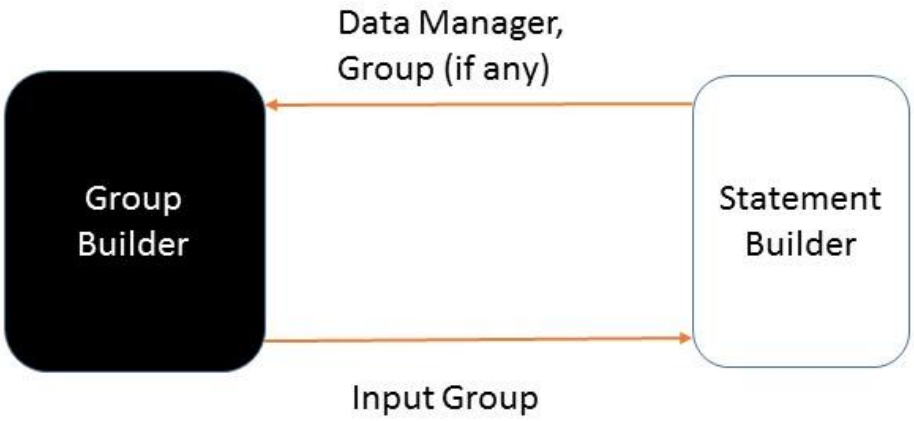
5.5.1 Main Window – Statement Builder



The Main Window, upon calling the Statement Builder, sends a reference to the Data Manager to the Statement Builder object. It also sends an optional Statement if it already has received one from a previous call of Statement Builder.

If the Statement Builder processes successfully (The user presses 'OK' and the input values are valid), it returns the input statement to the Main Window. If the process fails (The user presses 'Cancel'), a 'None' value is returned.

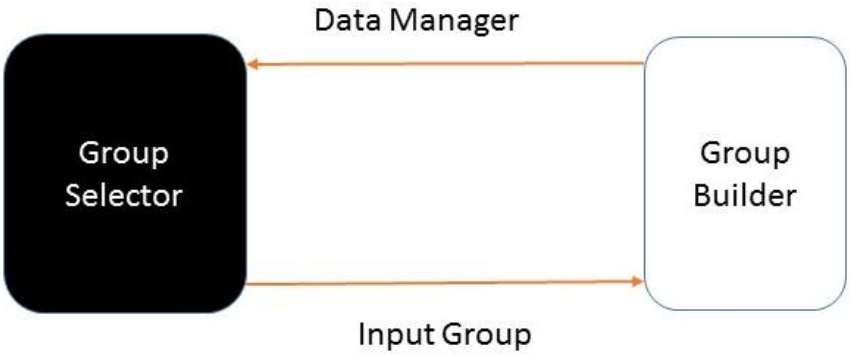
5.5.2 Statement Builder – Group Builder



The Statement Builder, upon calling Group Builder (when the user clicks on a 'Change Group' button), sends its Data Manager reference to the Group Builder object. It also sends an optional group to the Group Builder if already has received one from a previous Group Builder call.

Similar to how Statement Builder responds to Main Window, if Group Builder processes successfully (The user presses 'OK' and the input values are valid), it returns the input group to the Statement Builder. If the process fails (The user presses 'Cancel'), a 'None' value is returned.

5.4.3. Group Builder – Group Selector



The Group Builder, upon calling Group Selector, parses its Data Manager reference to the Group Selector object.

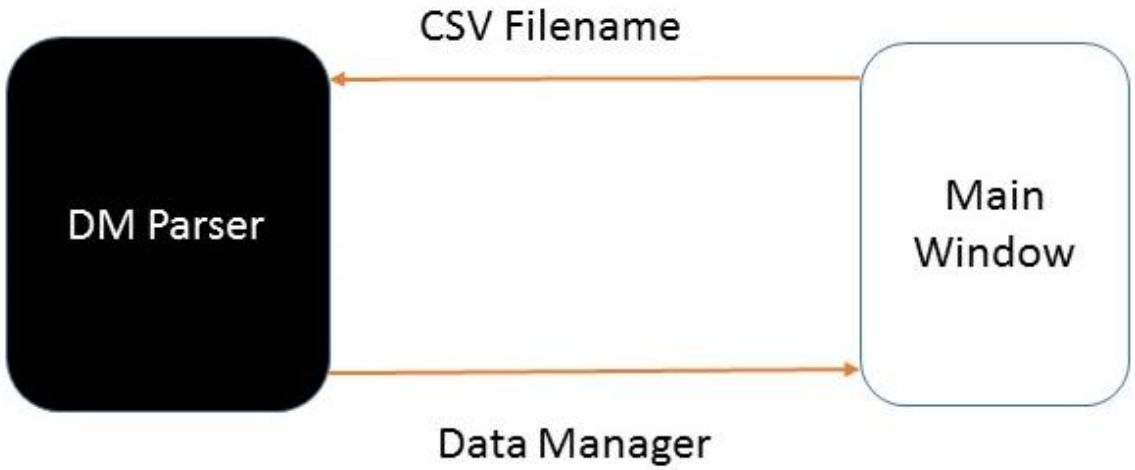
If the Group Selector processes successfully (The user presses 'OK' and a group is selected), then the input group is returned to Group Builder.

5.3.4. Main Window – Results Explorer

Main Window parses its Data Manager reference and Statement into Results Explorer. Results Explorer then proceeds based on data from both sources accordingly.

Results Explorer is not expected to return any data to Main Window.

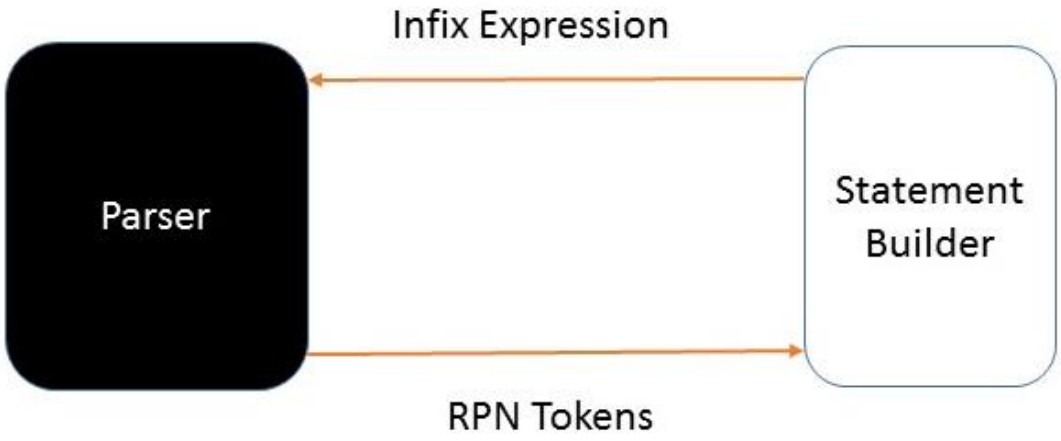
5.3.5 Main Window – DM Parser



Main Window sends the file path of the sample data (in CSV file form) to the DM Parser.

DM Parser then sends back a Data Manager object containing the processed data contained within the sample data.

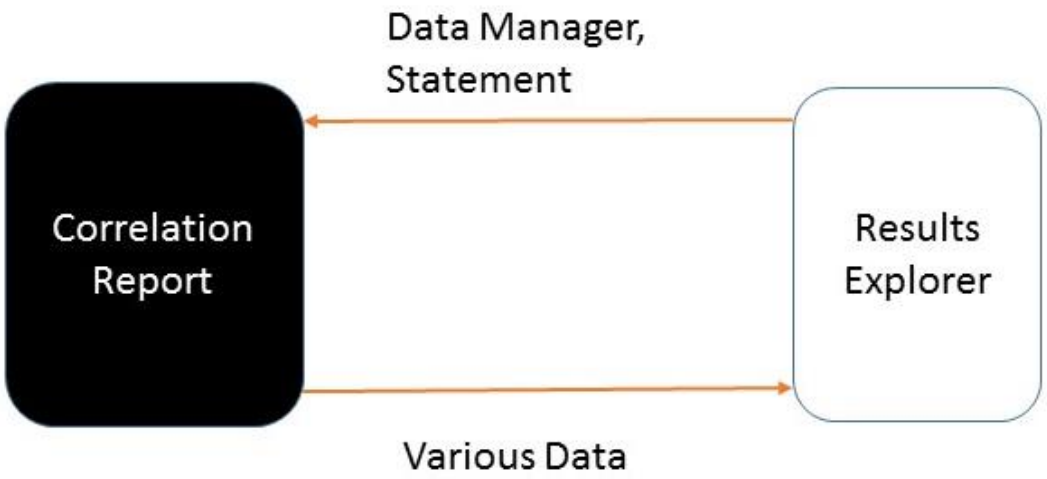
5.3.6 Statement Builder – Parser



The Statement Builder sends infix expressions to the Parser.

The Parser then returns a list of tokens representing the in-sequence RPN version of the input expression.

5.3.7 Results Explorer – Correlation Report



Results Explorer sends the Data Manager and Statement objects to the Correlation Report for processing.

After processing, Results Explorer will make calls for data from Correlation Report to populate its graphs and text reports, which the Correlation Report will furnish. The nature of these graphs is duly explained in Sections 2 and 3.

6 A simple use case

This use case illustrates how a typical user might use the program, and the processes he will perform during his analysis.

6.1 Scenario

Every half-year, Company Techno Koay sends its employees through a general aptitude assessment test as part of their drive to "improve staff skills". Just before their latest assessment test, though, the company sent 100 of its employees through a 3-day intensive productivity workshop. The HR manager wants to quickly assess if their staff benefited from the workshop using their aptitude assessment tests as a gauge of the staff's skills. He decides that the two values to compare are the scores before and after the workshop.

He surmises that if the test scores after the workshop are significantly higher than the test scores before, then the workshop does indeed improve the company staff's skills.

The HR manager begins by collating employee details from the database. Details like their scores before and after attending the workshop are included, as well as details such as the employees' gender, age, and even height and weight (Techno Koay is quite thorough about employee data collection). Once the data has been collated into a CSV file (with relevant column headers), he can begin using this program.

Note: A possible confounding factor is that people could have performed better on the second test than on the first regardless of whether they attended the workshop (The second test could have been easier than the first, so people who took both tests generally score better on the second than the first). To eliminate this confounding factor, the HR manager must also separately analyze the scores of those who didn't attend the workshop, and verify that these people either didn't score better on the second test, or didn't show as much improvement as those who attended the workshop.

6.2 Sample Data

The sample data used in this scenario is generated using the generateSampleData.py module.

Some key features about the data to note:

1. Score Before is set to be distributed equally between men and women, with a mean of 5000.
2. Score After is set to be distributed unequally between men and women, with men about 5050 and women about 4850.
3. Age is set to be distributed unequally between men and women, with women around 30 and men around 40.
4. Salary distribution correlates with the age distribution via the equation: Salary = 2000 + (100*Age)

Other factors are equally distributed between men and women.

6.3 Use Experience

When the user runs the program, he will be greeted by the Main Window.

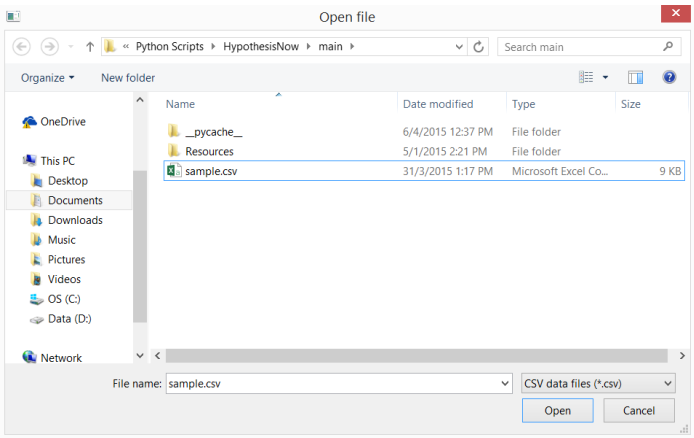


Figure 11: Open FIle Dialog

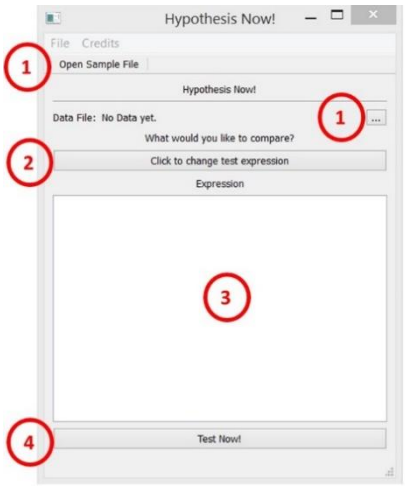


Figure 10: Main Window

The first thing he must do is load the sample file. He does so by either pressing the 'Open Sample File' button, or pressing the tool button next to the sample filename display (Figure 12, Labeled '1').

Pressing the button opens a file dialog which he can then use to locate and select his file (Figure 11). Once the file is selected, he is brought back to the Main Window, with the file loaded.

After loading the file, the user can proceed to modify his hypothesis.

The program allows users to input hypotheses via the use of test expressions. The user proceeds to do so by firstly clicking on the 'Click to Change Test Expression' button (Figure 12, Labeled '2'). Doing so opens the Statement Builder window.

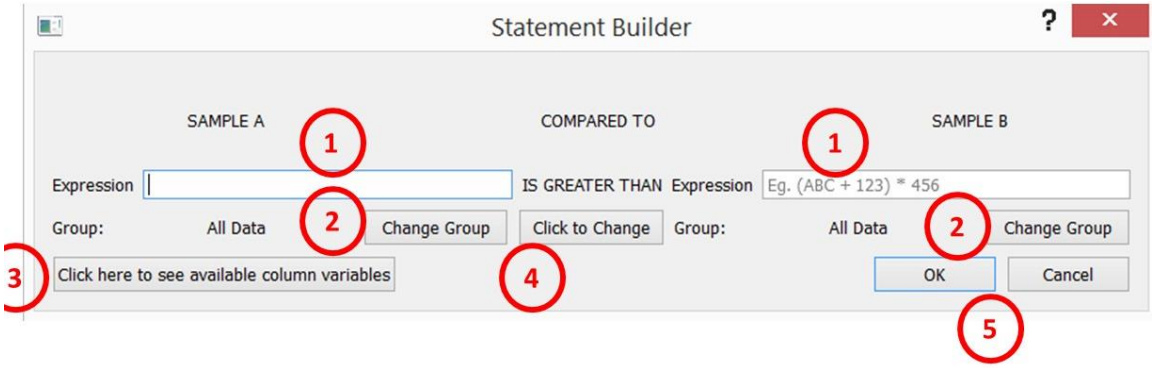


Figure 12: Statement Builder window

Once in the Statement Builder, the user can begin to setup his hypothesis. Since he is comparing the employees' scores before and after the workshop, he inputs this into the test expression by typing 'Score Before' into the left-hand side, and 'Score After' into the right-hand side (The fields labeled '1')

Once done, the user can confirm his statement by pressing 'OK'. This brings him back to the Main Window. In the Main Window, the test expression that the user just input will be displayed in the expression display box.

To check the program's analysis, the user clicks the 'Test Now!' button (See previous page, Figure 12, Labeled '4'). This brings the user to the Results Explorer window.

Figure 13: Results Explorer window upon loading

The Results Explorer window shows the distribution of the test results. To view this, the user selects the graph tab labeled '3: Test Results' (circled in above figure, right side)

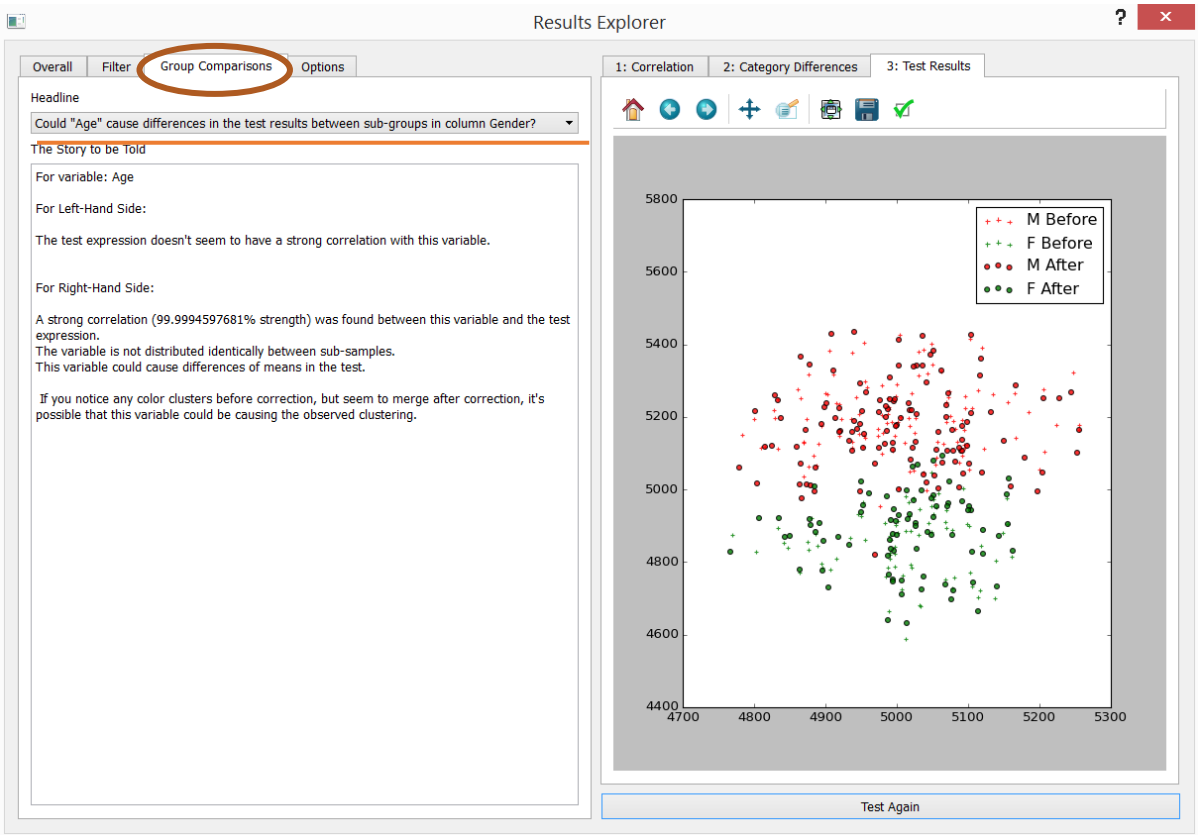


Figure 14: The Results Explorer window with Group Comparisons tab and the Test Results tab open.

The Results Explorer window also shows the user how the test statistics can be corrected for the possible influence of an individual variable. This variable can be selected by selecting the text tab labeled 'Group Comparisons' (Circled in above figure, left side), and selecting the variable to be corrected for under the combo box in this tab (Underlined in above figure, left side). Once the variable is selected, the text box below it will display the correlation report describing this variable's correlation with the test statistics and the potential for the data to cause this variable to impact the test statistics' behaviors, in the text box titled 'The Story to be Told'.

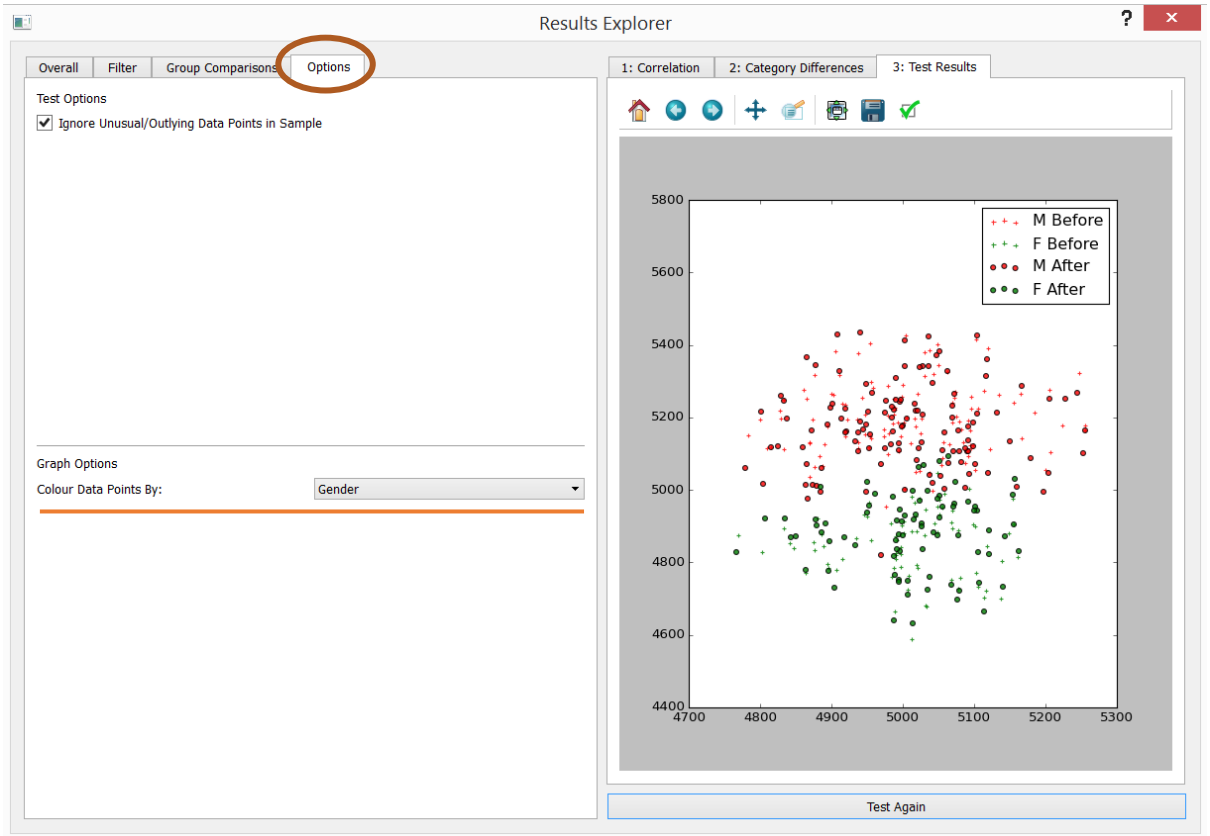


Figure 15: When the user clicks on the Options Tab from Fig. 15's screen, the Options Tab is opened to show this.

If the user wants to select which categories of data to view, he does so by entering the 'Options' tab (Circled in above diagram), and selects the column he wants to colour the data on the graph by with the 'Colour Data Points By:' combo box (Underlined in above diagram). Doing so groups the data according to categories within the selected column, allowing him to see if the data groups exhibit separate test statistic distributions before and after correction.

7 Recommendations for future work

7.1 Improving on correlation model

One possible improvement to the current correlation model is to expand the list of correlation types. Currently, the project only considers linear correlations for continuous variables. With relatively minor modifications to the program, it should also be able to handle simple logarithmic and quadratic regressions. With various functions in the Statsmodels library, this project could be furthered to handle more complex regression models.

In addition, as suggested in Section 2.5, this project could be extended to attempt to correct for categorical test statistics.

7.2 Automatic Data Exploration

One feature considered but eventually dropped, as discussed earlier, is the initial data exploration module. For the benefit of those who are interested in exploring the data before making hypotheses, this project could be furthered by extending it with a data exploration module.

In addition, this data exploration module could be used to apply more complex statistic models to the data, and automatically discover which models best fit certain behavior within the data.

7.3 Making this project market-worthy

Although the project intends to assist untrained data analysts, in the interest of proving the statistical concepts, some functions expected of a market-quality statistics product were unfortunately left out of the program. Relatively simple features such as reading Excel files and packaging the project into an executable file were not done. While the project sufficiently demonstrates the use of variable correction in statistics, many more user-experience related functions need to be implemented before this program can truly begin to help people.

In addition, as the storyboard of graphs was envisioned quite late into the project, it was inevitable that some bugs could not be ironed out in time. These bugs include, but are not limited to:

- Results Explorer's Graph 3 displaying a scatter plot instead of boxplots when the test expression and its data follows an Independent T-test pattern.
- Tests get performed multiple times with a change of the column to colour the data by, causing unexpected behavior such as multiple overlapping boxplots being drawn.

In addition, many things can be improved in the program, such as:

- Making the graphs clearer and more aesthetically pleasing using graphing libraries such as seaborn.

- Sample data storage using the program. This opens the doors to other practical functions such as saving of persistent filter groups to the program.
- Removal of unused features such as the comparator selection button in the Statement Builder (as suggested in Section 5.2.2)

Indeed, this project only scratches the surface of what can be performed automatically, and much more work can be done towards automatically exploring data.

8 Conclusions

Creating a general user-facing standalone application is not an easy task. Not only must the conceptual groundwork be solid, it must be conveyed to the target users in a simple-to-understand manner. Features to expose to the user must be prioritized by importance, and the less useful features must be hidden away or abstracted away from the user altogether, so as to not overload the user with unimportant information.

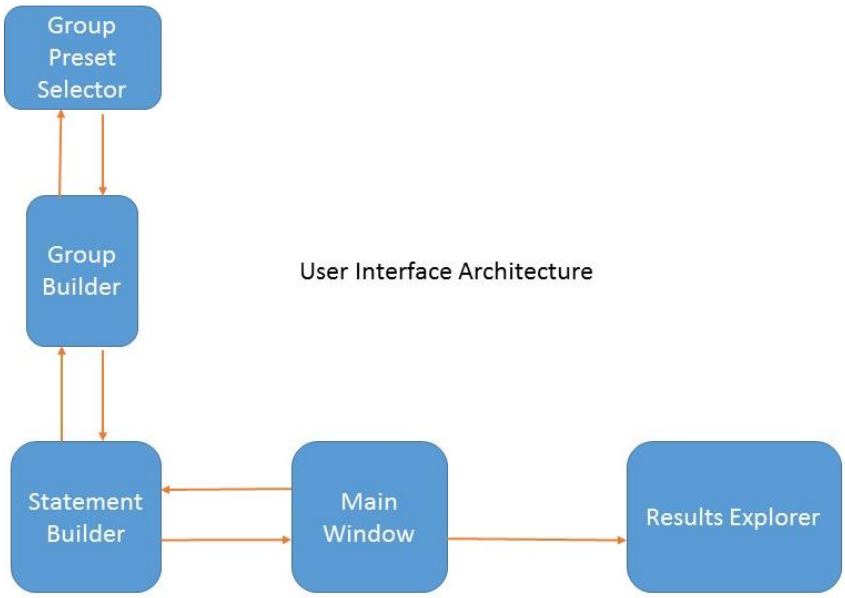
Although this project has a long way to go before it can serve people effectively, it establishes a good base for further work towards bridging the knowledge gap between those untrained in data analysis with trained data analysts. Allowing untrained people access to the power of statistical analysis of increasingly available data is key to unlocking new insights into data by people from all walks of life.

This project also impressed the importance of software engineering principles upon the student. Even a relatively small client program such as in this project is not feasible given this amount of time without the application of principles such as helper methods, abstraction, and command objects. The student has gained an appreciation of these principles through applying them in the course of this project.

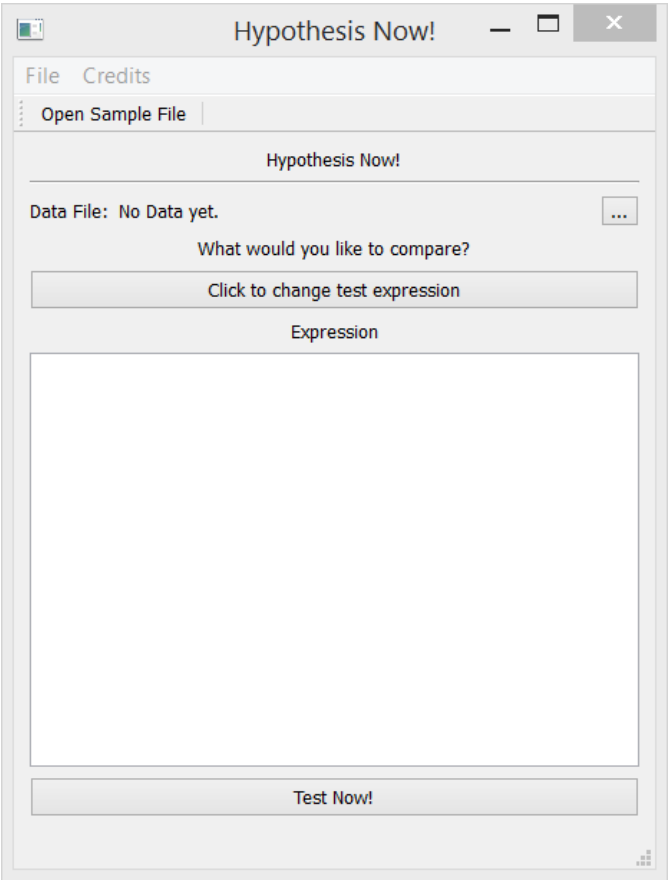
9 Appendices

Appendix A:

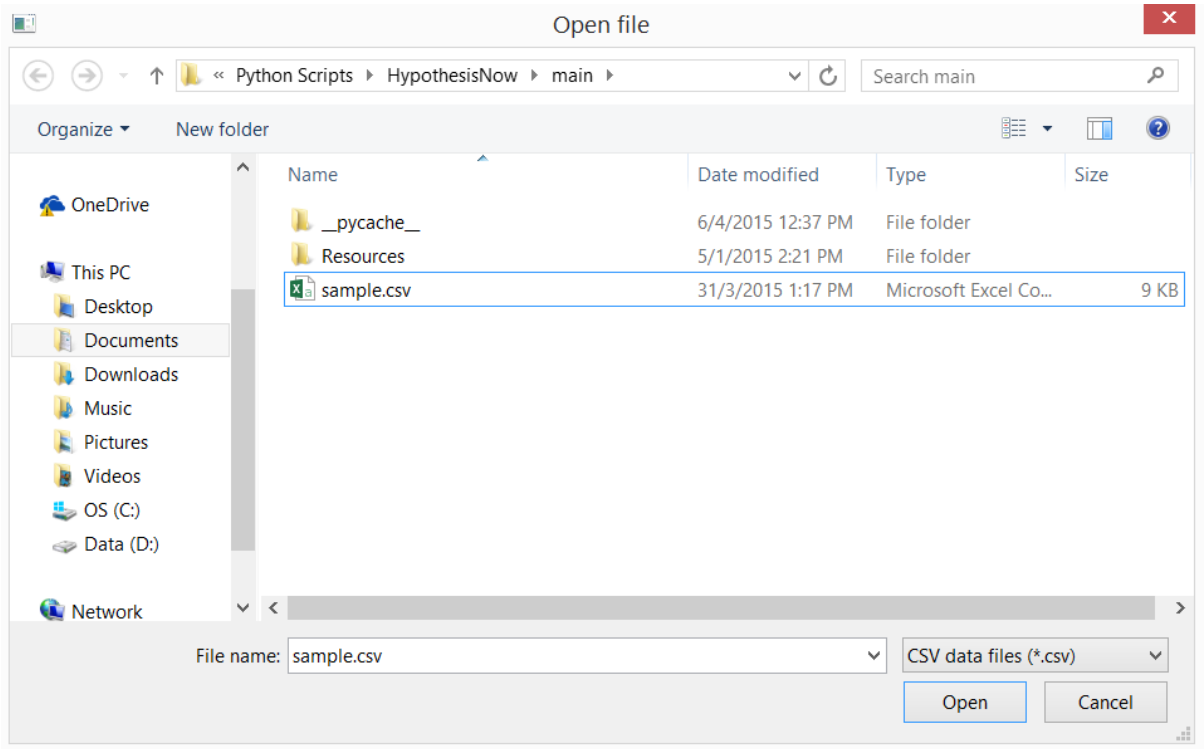
Overall Architecture Diagram (Diagram 1)



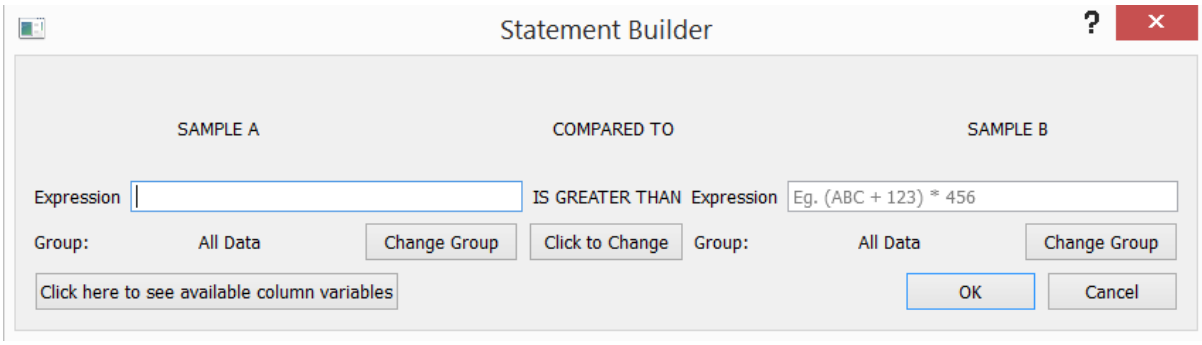
Main Window Screen:



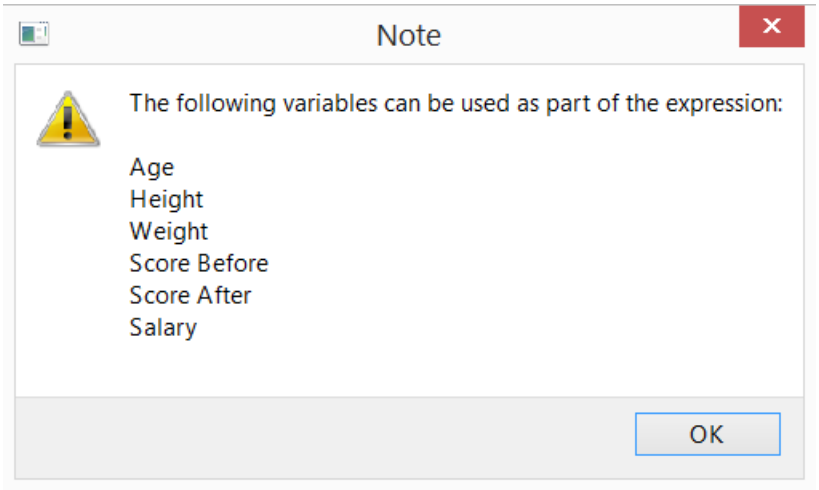
Open File Window (Opens when user selects ‘Open Sample File’ button):



Statement Builder

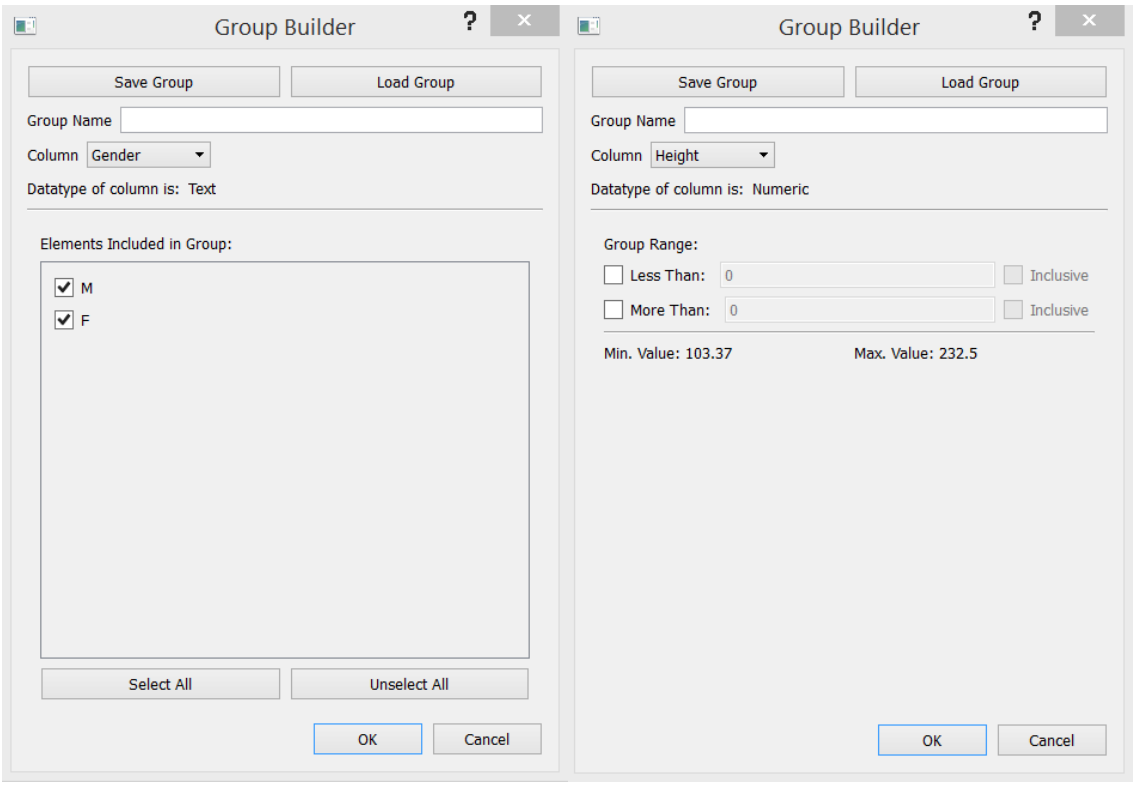


Usable Variables Popup (Opens when user clicks ‘Click here to see available column variables’):



Group Builder:

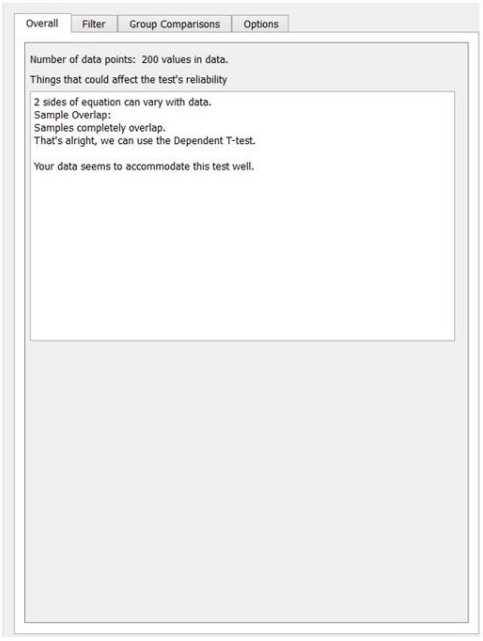
(When selected column is of text type) (When selected column is of numeric type)



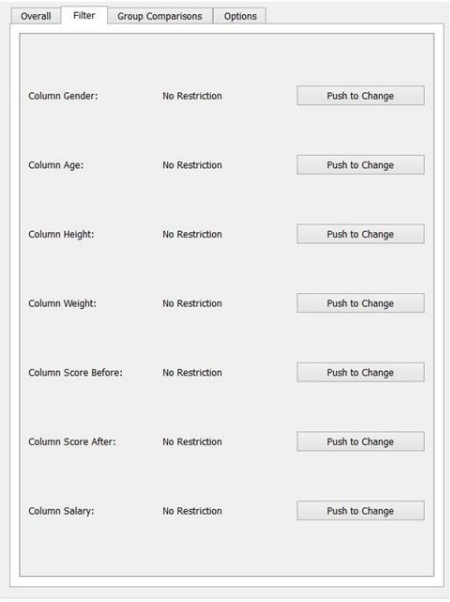
Results Explorer screens:

General Layout (Text Tabs on left, Graph Tabs on right)

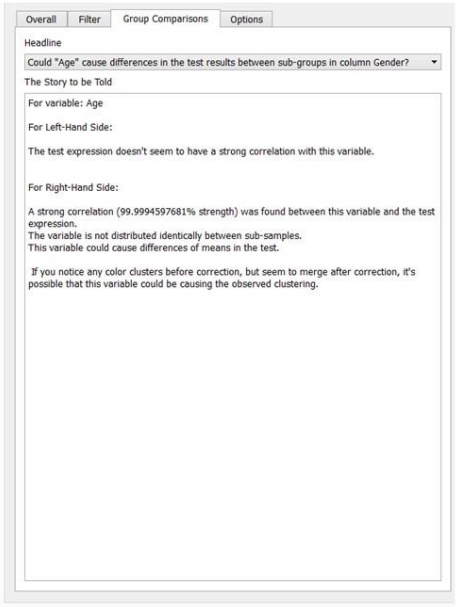
Results Explorer Text Tab 1: Overall



Results Explorer Text Tab 2: Filter

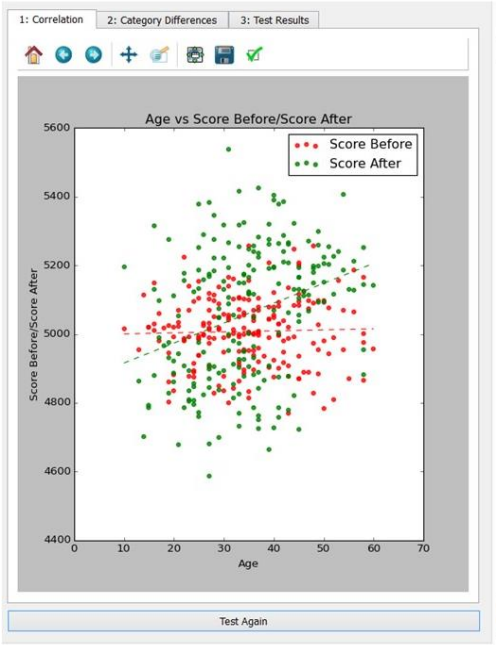


Text Tab 3: Group Comparisons

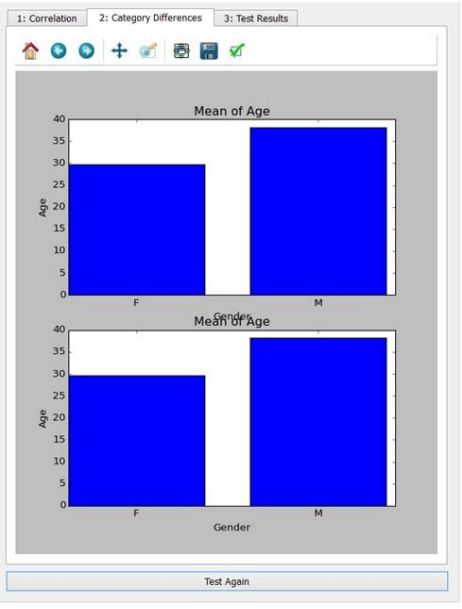


Text Tab 4: Options

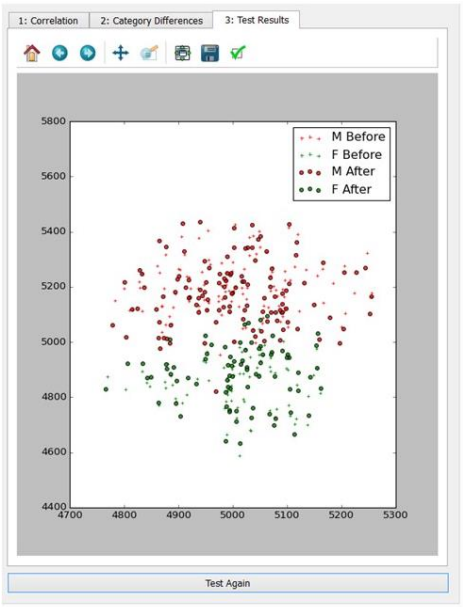
Graph Tab 1: Correlation



Graph Tab 2: Sub-Group Differences



Graph Tab 3: Test Results Before and After Correction



Appendix B:

Data Manager API:

Function	Description	Return(s)
getGroups()	Returns the dictionary of saved groups.	Dict(String = Group)
getGroupNames()	Returns the list of saved groups' names.	List(String)
getNumColumns()	Returns the number of columns in the dataset.	Int
getColType(token)	Returns a column type according to the corresponding column in token. If the token is of integer type, that column index's corresponding type is returned. Else if the token is a string, the column name's corresponding type is returned.	Int
getEntries()	Returns the full list of entries obtained from the sample data.	List(list(string))
getColumnTypes()	Returns the full list of column types, ordered by their corresponding column.	Numpy array
getColumnNames()	Returns the full list of column names, ordered by their sequence in the data.	List(string)
getColumnNameOfIndex(index)	Returns the name of the column of given index.	string
getColumnIndexOfName(token)	Returns the index of the column of given name.	Int
getTextSet(token)	Returns the textSet of the column of given name, if token is a string. If token is numeric, then the textSet of the column of given index is given instead. If the given name or index does not have a textSet, 'None' value is returned.	Set(string) / None
getTextSets()	Returns the dictionary of textSets.	Dict(string = set(String))
addGroup()	Adds the given group groupData of given groupName into the group dictionary. No value is returned.	None

Statement API:

Function	Description	Return(s)
getTestData()	Returns the test data.	List(list(string))
setTestData(data)	Sets the test data.	-
getLHSInfix()	Returns the infix expression on the left-hand side of the expression.	String
getLHSTokens()	Returns the RPN tokens for the left-hand side of the expression.	List(string)
getLHSGroup()	Returns the group applied on the left-hand side of the expression. If there is no group applied, this function returns 'None'.	Group / 'None'
getExpression()	Returns the comparison operator used by the expression.	String
getRHSInfix()	Returns the infix expression on the right -hand side of the expression.	String
getRHSTokens()	Returns the RPN tokens for the right-hand side of the expression.	List(string)
getRHSGroup()	Returns the group applied on the right -hand side of the expression. If there is no group applied, this function returns 'None'.	Group / 'None'
getStmtPrint()	Returns a string describing the test expression, as well as the applied filter groups on both sides of the test expression.	String

Group API:

Function	Description	Return(s)
getName()	Returns the name of the group.	String
setName(name)	Sets the name of the group.	-
Filter(data)	Returns the filtered list of entries.	List(list(String))
getRules()	Returns the list of rules in this group.	List(Rule)
changeRule(rule, index)	Changes the rule of that index	-
isEqualTo(group)	Compares this group with the input group, checking if the rules are the same. Returns True if same, false otherwise.	Boolean
printGroup()	Returns a detailed print of the group and its rules.	String

IsEmpty()	Returns True if the group allows all data entries through, false otherwise.	Boolean
ruleIsEmpty(index)	Returns True if the rule of column number 'index' is empty, False otherwise.	Boolean
getSetOfRuleIndices()	Returns the set of non-empty rules' corresponding indices.	Set(int)

Rule API:

Note: Rule's functions are non-functional. These functions must be overloaded by its children. Future developers must take note of this when using Rule as a parent class.

Function	Description	Return(s)
isEmpty()	Returns True if the rule is empty, False otherwise.	Boolean
setIsEmpty()	Sets the rule's empty state depending on the information within the group.	-
Filter(data)	Returns the filtered list of entries.	List(list(String))
getColNum()	Returns the column number of the column this rule influences.	Int
getColName()	Returns the name of the column this rule influences.	String
printGroup()	Returns a print describing the rule.	String
printTitle()	Returns a shorter print summarizing the rule (for graph display purposes)	String
includedInGroup(entry)	Returns True if the entry passes this rule's criteria, False otherwise.	Boolean
isEqualTo(rule)	Returns True if this rule is functionally equal to the input rule, False otherwise.	Boolean

RuleNum API:

RuleNum uses all the functions in Rule's API.

RuleText API:

RuleText uses all the functions in Rule's API, and this function:

Function	Description	Return(s)
getIncludedElems()	Returns the set of elements that this rule passes for.	Set(String)

TestLogic API:

TestLogic was originally designed to perform stratified testing. However, the shift from stratified testing to correlation discovery and correction led to many of the functions being either commented out or re-purposed to fit the new functional requirements.

Function	Description	Return(s)
setMinSampleLength(int)	Sets the minimum sample size. Note: This function is currently unused.	-
setRemoveOutliers(Boolean)	Sets the remove outliers mode to True or False as per the input state.	-
getTestType()	Returns the type of test received by Test Logic in a string form. The string's corresponding type is defined in Typedef.	String
getValidationPrint()	Performs validation tests on the data, and returns a print representing the results of the validation tests.	String
performTest()	Performs the test defined by the statement TestLogic has. Note: This function is overloaded in Correlation Report, and is non-functional in TestLogic.	-

Correlation Report API:

Correlation Report, the spiritual successor of **Test Logic**, extends on the functionality of **Test Logic** to also perform testing for correlations and correct test statistics to account for the possible influence the confounding variable could have on the test statistics.

Function	Description	Return(s)
getCorrectingColIndex(int)	A list of integers is given to Results Explorer. Given the index of the list, return the corresponding integer.	int
getCorrelationTitles(index)	Returns a list of titles for the correlation reports of every column index except the input index in string form.	List(string)
getCorrelation(index)	Returns the correlation of the column of given index against the test statistic(s). This is either a list containing the regression line's info as 3 floats, or a list containing a p-value (float) and a dictionary representing the variable's different category counts (A counter-type	List(float) / List(float, Counter)

	Dictionary object) If the test type is a 2-sample test, a tuple of two such objects, one for each sample, is returned.	
getCorrelationPrint(hold, correcting)	Returns the correlation print associated with the potential confounding variable of column 'correcting' against sub-groups split by variable of column 'hold'	String
performTest()	Performs the test defined by the statement CorrelationReport has, generating the correlations for future internal use.	-
getVals(hold, correcting, correctionEnabled)	Gets the graph values of the test statistics for different groups, split by the variable of column 'hold'. If correctionEnabled is true, the graph test statistics are also corrected by the variable of column 'correcting'.	List(String, List(float), List(float))
getVarVsTestStats(colNum)	Gets the list of coordinates of entries' variable (of column index 'colNum') against their corresponding test statistics, as two lists of values (XVals, YVals). Only works if the 'colNum' column is numeric.	List(float), List(float)
getDiffCategoryMeans(splitCol, correctCol)	Gets the lists of variable means for the variable of column index 'correctCol'. Each list is split according to the variable of column index 'splitCol'. If column 'correctCol' is of type text, a Counter representing the distribution of the variable categories within each category is returned instead.	List(List(String, float)) OR List(List(String, Counter))

HNStats API:

HNStats is a module housing the collection of functions. The functions used in the latest version of the program are listed here.

Function	Description	Return(s)
isNumeric(token)	Returns True if the token can be converted into a numeric value successfully, False otherwise.	Boolean
evaluate(LHS, RHS, operator)	Evaluates the params as if they were an infix math expression in the order (LHS,	float

	operator, RHS), returning the expression's result as a value.	
checkSampleOverlap(testData, group1, group2)	Checks the samples (obtained by filtering the testData using group1 and group2) for overlap, and returns a string of the sample overlap report. Also returns a Boolean. True if samples show full overlap or zero overlap, False if partial overlap.	String, Boolean
removeOutliersInCol(entries, colNum)	Performs Grubb's outlier test using the variable in column 'colNum', removing entries that are found to be outlying. Returns the entries remaining after the test is performed.	List(list(string))
performTest()	Performs the test defined by the statement TestLogic has. Note: This function is overloaded in Correlation Report, and is non-functional in TestLogic.	-

Parser API:

The **Parser** serves only one function: To parse infix expressions and convert them into Reverse Polish Notation (RPN) form, as an ordered list of string tokens.

Function	Description	Return(s)
parse(string)	Given an infix math expression, return a list of tokens representing the ordered RPN form.	List(string)