Chair for Computer Science 10 (Media Computing and Human-Computer Interaction)

RWTH AACHEN UNIVERSITY

Stathunt – Supporting Novice Researchers Seek Statistical Procedures

Bachelor's Thesis
submitted to the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Malte Adrian Xiang Rui Meng

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Ulrik Schroeder

Registration date: 13.05.2020
Submission date: 12.06.2020

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Meng, Malte Adrian Xiang Rui	354529
Name, Vorname/Last Name, First Name	Matrikelnummer (freiwillige Angabe) Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

StatHunt – Supporting Novice Researchers Seek Statistical Procedures

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, 12.06.2020	
Ort, Datum/City, Date	Unterschrift/Signature

*Nichtzutreffendes bitte streichen
*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen 12.06.2020	
Ort, Datum/City, Date	Unterschrift/Signature

Contents

List of Figures

Abstract

This thesis outlines the creation of StatHunt, an interactive web-application aimed to help researchers identify correct statistical procedures for their experiments. To explain the design rationale behind StatHunt, the underlying problems it is trying to solve are presented. For the design of StatHunt, State Transition Networks are used to model conversation flow of a developed chatbot. The concrete implementation of this conversational flow in Botpress is then detailed. For experiment visualisation the usage of Blocky is explained and implementation information is provided. As a privacy mechanism for researchers to share their data, differential privacy is introduced. Its use in obfuscating data in StatHunt is explained.

Acknowledgements

Firstly, I would like to thank Prof. Dr. Jan Borchers, my thesis supervisor, for his time and effort.

I would also like to thank Prof. Dr. Ulrik Schroeder, for agreeing and taking the time to become the second examiner.

I would also like to thank all of the members at the Media Computing Group for being so welcoming and making my time spent at the Chair so wonderful.

Finally, I would especially like to thank Krishna Subramanian for his time and support. His well-placed insight and critical feedback have turned this thesis into the most valuable experience of my studies.

Thank you!

Malte Meng

Chapter 1

Introduction

Statistics are widely used in research to validate experiment results

Inferential statistics is a branch of mathematics used to infer information about a large population, through analysis of a small sample. In many humanitarian sciences, methods taken from inferential statistics are used to evaluate data from experiments or studies. In HCI (Human Computer Interaction), one of the most commonly used techniques is NHST (Null Hypothesis Significance Testing). In Null Hypothesis Significance Testing, the researcher selects and applies one of many possible tests to prove the validity of their hypothesis. Some commonly used tests are ANOVA (Analysis of Variance), the t-test, Chi-square test and Fisher's test.

Selecting the right statistical test and applying it correctly is a difficult task

Research on significance testing in HCI ranges from questioning the appropriateness of this approach (Kaptein and Robertson [2012]), to recommending alternative methods (Dragicevic [2016]) and investigating problems experienced researchers have when applying inferential statistics (Cairns [2007]). The many problems that researchers have when selecting and applying statistical tests to their experiments show that both are inherently complex. The causes of this, identified by Cairns [2007], are researchers not validating underlying data assumptions, and using inappropriate tests. An additional factor Cairns [2007] identified was a broad lack of statistical education.

1.1 Problem Statement

One of the most commonly used support resources for Statistics is Q&A websites

For novice researchers lacking experience in both experiment design and statistics, finding the correct statistical test and ensuring that required assumptions, like the data distribution, are met, can be daunting. Although many different resources exist to support the decision-making process, there is no standard guideline for how one should proceed. To investigate how researchers and data practitioners approach their search for a statistical test, Hu [2019] performed a group of interviews. One of the predominant discoveries was that the main resource each participant used was Q&A websites, with the most common ones being StackOverflow[1], CrossValidated[2] and ResearchGate[3].

Many researchers don't know what and how to post a good question

To discover how effective this was, 76 questions on statistical tests were taken from CrossValidated by Hu [2019]. The analysis of these question uncovered that the following core issues had a strong effect on the help researchers received:

- Missing information
- Poorly structured information
- Fabricated information
- Unclear formulation

These issues led to respondents making multiple assumptions when providing help. Additionally, respondents would ask for clarification, leading to lengthy back-and-forth communication.

1.2 Our Solution

A web-app is introduced to aid researchers in posting questions

To ensure that respondents on Q&A websites can directly

[1]https://stackoverflow.com
[2]https://stats.stackexchange.com
[3]https://www.researchgate.net

understand and support researchers seeking help, the interactive web-application StatHunt is presented. Through dialogue, a chatbot (conversational agent) guides novice researchers through:

1. Providing their experiment design
2. Uploading their data
3. Formulating their question

As researchers provide their experiment design, StatHunt generates an interactive visualisation of the experiment, inspired by Eiselmayer et al. [2019]'s Touchstone2 experiment visualisation. This allows the user to make changes dynamically. After uploading their data, StatHunt provides the user with the opportunity to either obfuscate the values or variables of their data. This provides them with the privacy they might want when sharing their information. At the final step, the user is provided with a shareable URL containing both the generated experiment design and data. This URL can then be provided when asking for help on Q&A platforms.

1.3 Overview

This thesis will explain StatHunt's design decisions and provide implementation details

In the Background and Related Work section, the research that led to the problem statement will be introduced. In the StatHunt section, the exact specifications that were created to address the problems will be outlined, in addition to the interaction design and workflow. Following this, the technical aspects of StatHunt's final implementation will be explained. This will include introducing the libraries used, the software architecture, and the specific development details for the experiment design and dataset page. In the final chapter of this thesis, the contributions of the work will be summarised and recommendations will be made for future work.

Chapter 2

Background and Related Work

2.1 Seeking Statistical Help

2.1.1 Behaviour of Data Practitioners

To investigate the resource-seeking behaviour of data practitioners for statistics, Hu [2019] interviewed 12 researchers. In exploring procedure, problems, resources and attitudes when it came to that behaviour, Hu [2019]'s interview findings revealed three main themes:

1. Problems in method selection
2. Coping strategies
3. Resource utilisation

Data practitioners apply commonly used or known tests when facing problems finding the right one

Due to different prerequisites required by validity tests, and the lack of a standard, data practitioners face difficulties when trying to determine the correct method for their data. To cope with this, participants of the study would either default to a commonly used method in their field or apply the test they were most comfortable with. In doing so, the correctness of their chosen method was not guaranteed, potentially leading to issues with result validity.

When it came to resource utilisation, researchers used books, looked at methods applied in relevant papers, asked more experienced colleagues, or went on Q&A sites such as StackExchange, Crossvalidated, and ResearchGate.

2.1.2 Q&A Websites

Question analysis on Q&A sites showed problems researchers have when seeking help

Upon discovering that Q&A sites are the only unanimously used resource, Hu [2019] analysed 76 statistical test-related questions from ResearchGate and CrossValidated. The analysed questions were seeking either a recommendation for the correct statistical procedure; validation of a selected statistical procedure, or further information.

The information that researchers provided could be divided into four sections:

- Hypothesis
- Analysis goal
- Experiment design
- Experiment dataset or data visualisation

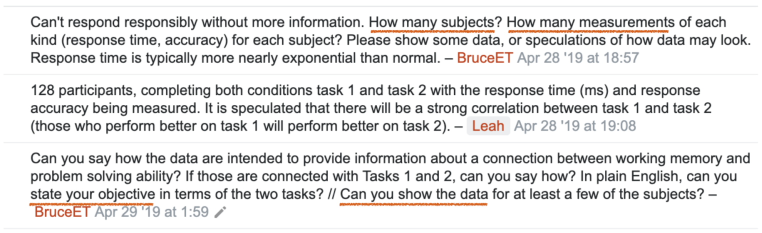
By looking at comments, multiple problems were identified with posted questions.

These problems could be broken down into the following four categories:

1. Missing information
2. Poorly structured information
3. Fabricated information
4. Unclear formulation

Missing Information

Out of the 76 questions, Hu [2019] discovered that 29 questions were missing at least one of four vital pieces of information. This caused respondents to request the information, or provide a solution based on assumptions they had made. In Figure 2.1, *BruceET* had to ask about sample size and dependent variables as the information was not provided.


Can't respond responsibly without more information. How many subjects? How many measurements of each kind (response time, accuracy) for each subject? Please show some data, or speculations of how data may look. Response time is typically more nearly exponential than normal. – BruceET Apr 28 '19 at 18:57

128 participants, completing both conditions task 1 and task 2 with the response time (ms) and response accuracy being measured. It is speculated that there will be a strong correlation between task 1 and task 2 (those who perform better on task 1 will perform better on task 2). – Leah Apr 28 '19 at 19:08

Can you say how the data are intended to provide information about a connection between working memory and problem solving ability? If those are connected with Tasks 1 and 2, can you say how? In plain English, can you state your objective in terms of the two tasks? // Can you show the data for at least a few of the subjects? – BruceET Apr 29 '19 at 1:59

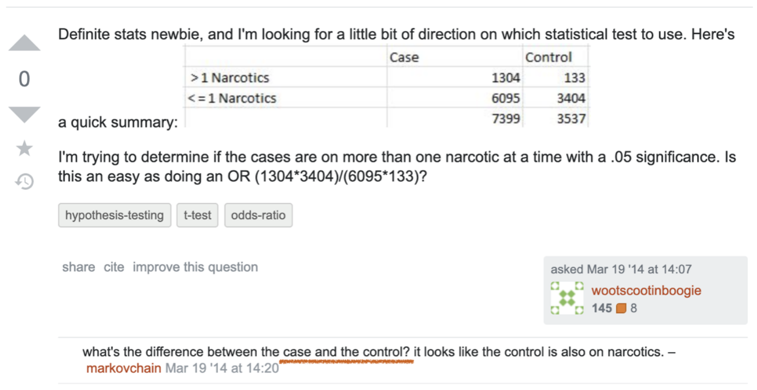
Figure 2.1: Question responses to missing information

Badly Structured Information

In a further 13 questions, the information researchers provided was improperly structured. This again led to respondents asking for further clarification. In Figure 2.2, *wootscootinboogie* used a table to present their variable information, which led to a question asking for further clarification from *markovchain*.

Fabricated Information

Unwilling to share their actual experiment information, researchers also fabricated details of their experiment. This included providing fake variable names, or fake variable properties. In Figure 2.3, *Damien* can be seen doing this. Despite having collected it, some researchers were also unwilling to provide their data due to privacy issues. In Figure 2.4, *user2079355* did this, creating a visualisation that they claim resembles their actual dataset.



0

Definite stats newbie, and I'm looking for a little bit of direction on which statistical test to use. Here's a quick summary:

	Case	Control
> 1 Narcotics	1304	133
<= 1 Narcotics	6095	3404
	7399	3537

I'm trying to determine if the cases are on more than one narcotic at a time with a .05 significance. Is this an easy as doing an OR (1304*3404)/(6095*133)?

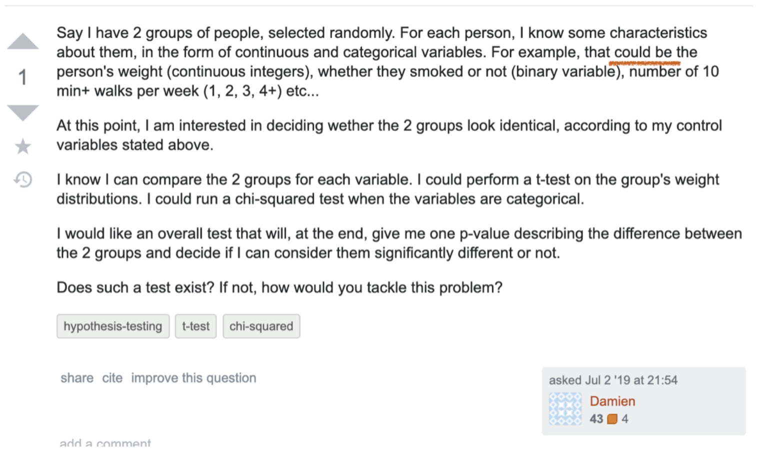
hypothesis-testing t-test odds-ratio

share cite improve this question

asked Mar 19 '14 at 14:07
wootscootinboogie
145 8

what's the difference between the case and the control? it looks like the control is also on narcotics. – markovchain Mar 19 '14 at 14:20

Figure 2.2: Question with badly structured information



1

Say I have 2 groups of people, selected randomly. For each person, I know some characteristics about them, in the form of continuous and categorical variables. For example, that could be the person's weight (continuous integers), whether they smoked or not (binary variable), number of 10 min+ walks per week (1, 2, 3, 4+) etc...

At this point, I am interested in deciding wether the 2 groups look identical, according to my control variables stated above.

I know I can compare the 2 groups for each variable. I could perform a t-test on the group's weight distributions. I could run a chi-squared test when the variables are categorical.

I would like an overall test that will, at the end, give me one p-value describing the difference between the 2 groups and decide if I can consider them significantly different or not.

Does such a test exist? If not, how would you tackle this problem?

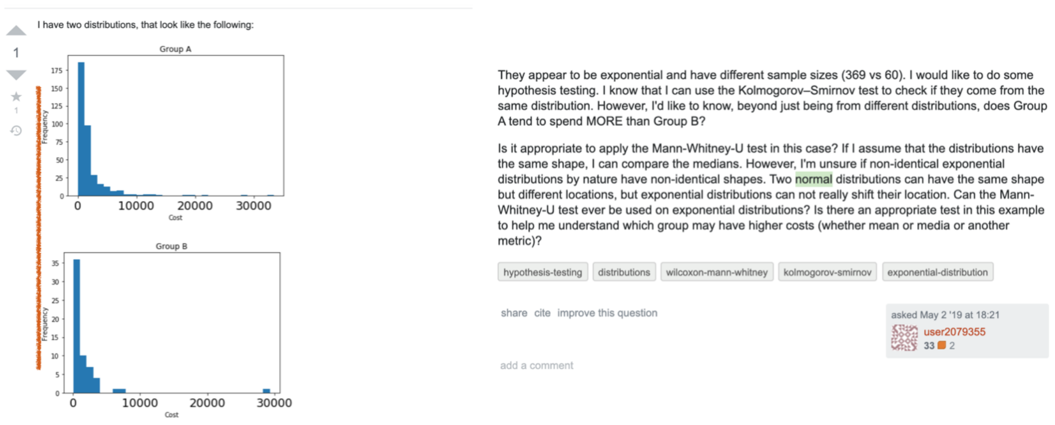
hypothesis-testing t-test chi-squared

share cite improve this question

asked Jul 2 '19 at 21:54
Damien
43 4

add a comment

Figure 2.3: Question with poor variables



1

I have two distributions, that look like the following:

They appear to be exponential and have different sample sizes (369 vs 60). I would like to do some hypothesis testing. I know that I can use the Kolmogorov–Smirnov test to check if they come from the same distribution. However, I'd like to know, beyond just being from different distributions, does Group A tend to spend MORE than Group B?

Is it appropriate to apply the Mann-Whitney-U test in this case? If I assume that the distributions have the same shape, I can compare the medians. However, I'm unsure if non-identical exponential distributions by nature have non-identical shapes. Two normal distributions can have the same shape but different locations, but exponential distributions can not really shift their location. Can the Mann-Whitney-U test ever be used on exponential distributions? Is there an appropriate test in this example to help me understand which group may have higher costs (whether mean or media or another metric)?

hypothesis-testing distributions wilcoxon-mann-whitney kolmogorov-smirnov exponential-distribution

share cite improve this question

asked May 2 '19 at 18:21
user2079355
33 2

add a comment

Figure 2.4: Question with fabricated data

2.2 Support Tools for Experiment Design

One of the most essential features in StatHunt is experiment visualisation. The three main tools developed in HCI to address this are Touchstone, NexP and Touchstone2. Despite Touchstone2 being the main inspiration for StatHunt's visualisation techniques, understanding both the work done by MacKay et al. [2007] in Touchstone and Meng et al. [2017] in NexP is important.

Touchstone, NexP and Touchstone2 provide examples of experiment representation

2.2.1 Touchstone

Studies have proven that even small features of experiment design can impact the results of studies (Gray and Salzman [1998]). Some work has been done on assisting researchers in designing and formally structuring their experiment. In order for HCI researchers to establish a solid foundation, MacKay et al. [2007] introduced TouchStone, a set of tools developed to enable researchers to compare alternative experiment designs. Through a GUI (Graphical User Interface), Touchstone allows users to specify their design information. A help window points to online resources that provide the users with support, if required.

2.2.2 NexP

Despite the completeness of Touchstone, Meng et al. [2017] later proposed and developed the alternative NexP. Meng et al. [2017] designed NexP to help more inexperienced researchers understand the process of experiment design. To facilitate this, NexP, contrary to Touchstone, eases the user into the experiment design process by asking them a series of questions. In addition to using more informal language when framing questions, examples are provided. These give the user a point of reference when formulating their hypothesis or choosing factors for their experiment. Over the course of a user study, Meng et al. [2017] discovered that novices considered this approach more intuitive and easy-to-use when compared to Touchstone.

2.2.3 Touchstone2

Most recently in tools that aid experiment design development, Eiselmayer et al. [2019] introduced Touchstone2. Aimed at tackling what Baker and Penny [2016] identify as a reproducibility crisis in science, Touchstone2 not only allows users to create their own experiment design, but allows them to share it in their interactive workspace. The design is different to that of NexP as Eiselmayer et al. [2019] provide a reworked visual environment for manipulating designs and their parameters. By representing experiments as a collection of nested bricks containing important parameters, users create their experiments by dragging these bricks onto a workspace where they snap together.

2.3 Data Obfuscation

Obfuscation is a method used to change data for privacy protection while preserving its statistical value as well as possible

To address privacy concerns that a user might have when it comes to sharing experiment data, StatHunt provides users with the option to obfuscate it. This consists of changing the individual values of the data, while trying to minimise the effect that the change has on different statistical parameters. A common method for changing data values is running every value through a function that applies a random amount of noise.

To ensure that such a function provides the dataset with a sufficient amount of privacy, we use the following definition provided by Dwork and Roth [2013]:

An algorithm A is ϵ-differentially private with $\epsilon \in \mathbb{R}$ and $\epsilon > 0$, if for all datasets D_1 and D_2 differing on a single element and all subsets $S \subseteq \text{Im}(A)$

$$\Pr[A(D_1)] \in S \leq \exp(\epsilon) \times \Pr[A(D_2) \in S]$$

Where $\Pr[x \in M]$ is the probability of x being in M and $\exp(x)$ is e^x.

Since the Laplace mechanism is an ϵ-differential private mechanism, StatHunt uses it to obfuscate the experiment data. The Laplace mechanism takes a dataset of numerical values and adds noise taken from a Laplace distribution with a mean of 0.

2.4 Chatbots

Chatbots provide users with an intuitive method of providing information or using services

Chatbots, also referred to as conversational agents, are software agents that can communicate with users via written text. A chatbot can act as an interface over which users can perform tasks that would otherwise have to be done in a graphical user interface. With a growing use of messaging platforms such as WhatsApp, Telegram, WeChat and Slack, the barrier to entry for users talking to a well-designed chatbot is low. Due to the many development tools one can use to create complex chatbots (Microsoft Bot Framework[1], IBM Watson[2]), the technical know-how required to make them is low. In research, the trend towards using chatbots has not gone unnoticed, with Følstad and Brandtzaeg [2017] predicting the implications the swing towards natural language interfaces can have for HCI. Since the majority of usability research is currently being done on hardware design or graphical user interfaces, Følstad and Brandtzaeg [2017] even recommend a shift from design being seen as an interpretative task to being seen as an exploratory one.

Multiple research papers have also introduced different chatbots. To try to increase their efficiency, Ranoliya et al. [2017] designed and implemented a chatbot to answer university related questions. Introducing a new methodology to create chatbots that can handle more complex tasks, Fast et al. [2018] developed Iris, a chatbot aimed to help users perform open-ended data science tasks.

[1]https://dev.botframework.com
[2]https://www.ibm.com/watson

2.5 Existing Support Solutions for Inferential Statistics

2.5.1 StatPlayground

StatPlayground, as proposed by Subramanian and Borchers [2017], is an interactive web-app aimed at helping researchers improve their statistical know-how. In StatPlayground, users are asked to select one of a few pre-defined datasets. They can then control different parameters of the dataset through interactive visualisations and observe the effects that these have on the statistics. By changing the selected statistical test, users can determine what effects applying different tests can have on the outcome of experiment data. Taking this exploratory approach, StatPlayground can help novice researchers better understand statistical procedures, data, and identify relationships between statistical concepts. With increased statistical literacy, researchers will be able to make more informed decisions on which statistical test could be a right fit for their experiment.

2.5.2 Statsplorer

Statsplorer, introduced by Wacharamanotham et al. [2015], is a tool developed to help novice researchers learn and apply statistical tests. After selecting a variable from experiment data, Statsplorer selects an appropriate visualisation and presents it to the user. Statsplorer then tests different data assumptions and uses visualisations to help the user select a statistical test and interpret the results of analysis.

Chapter 3

StatHunt

3.1 Specifications

3.1.1 Feature Recommendations

In the outlook of their work, Hu [2019] provide recommendations for an expansion of Q&A websites to tackle researcher problems. This proposed system would determine at which stage the researcher was and detail the information they would need to provide. In designing Stathunt, the decision was made to follow the recommendations in the best possible way. In accordance with this, the following seven core features were used as a baseline:

Seven core features were taken from Hu [2019]'s system recommendation and used as a baseline for StatHunt

1. Tell the data scientist what experiment information they need to provide
2. Structure the provided information in an easily understandable form
3. Prompt the user to provide data if possible
4. Provide charts for possible data visualisation
5. Fabricate a dataset and variables
6. Notify researchers when comments are made on their question

7. Help formulate the researcher's question and help generate relevant question tags

3.1.2 Limitations and Final Features

The original intention was to develop an extension to current, existing Q&A websites. However, this was not a possibility. The reason for this is that while one can program user actions using the StackExchange API, developing an integrated extension is not possible. As the next best alternative, StatHunt was developed as a web-application, implementing therelevant Q&A features using the StackExchange API.

In the design of StatHunt, four core features will be included

After further investigation, the features to be included were altered. With Feature 4. being recommended to present the user's data in an effective way, posting the entire dataset would be more valuable to respondents. By adding in data obfuscation as a feature, users that would be unwilling to share their own data due to privacy concerns could still do so. As Feature 6. is already implemented in StackExchange, it was removed. The revised features that StatHunt should include were:

1. Getting the required experiment information from the data scientist

2. Structuring the provided information in a easily understandable format

3. Obtaining the user's dataset and allowing them to obfuscate it

4. Helping to formulate the user's question and generating relevant question tags

3.2 Interaction Design

3.2.1 Chatbot

StatHunt utilises a chatbot to guide the user and receive their input

The main interaction technique implemented for use in StatHunt was a chatbot. Using the chatbot made applying a question-answer scheme, similar to what can be found in NexP, straightforward. Additionally, instead of having examples under the questions in plain text, the chatbot can provide help and examples when asked to do so. Inexperienced users have the benefit of receiving support, whereas experienced ones are not overloaded with redundant information.

3.2.2 Experiment Representation

Taken from the analysis performed by Hu [2019], required pieces of information StatHunt had to gather were:

- Hypothesis
- Goal of Analysis
- Independent Variables
- Dependent Variables
- Sample Size
- Procedure
- Within- / Between-group design

Interactive blocks provide the user with an alternative method to provide experiment information

To represent this information in a structured matter, a similar approach to Touchstone2 is utilised. A structure of nested bricks was created, containing labels indicating to the user what information the brick required. The brick workspace is split into two sections, as seen in Figure 3.1 The section annotated with a **1** is the toolbar, which features template blocks. Researchers can drag the three blocks out of the toolbar into section **2**, the workspace, where they

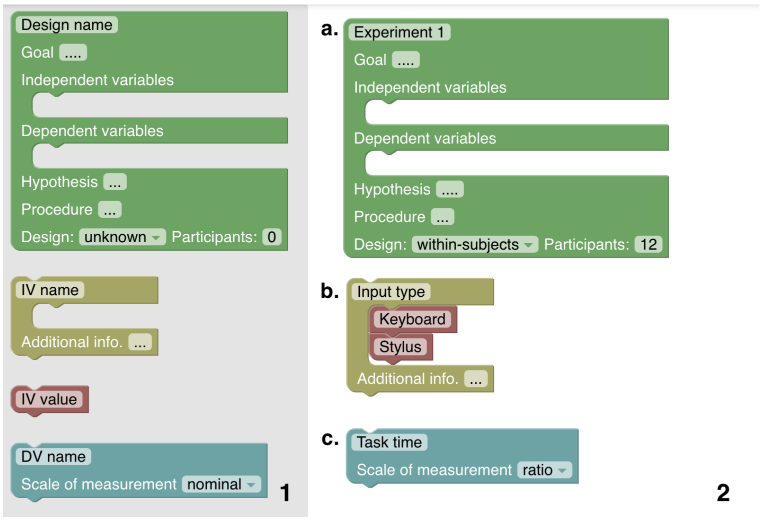
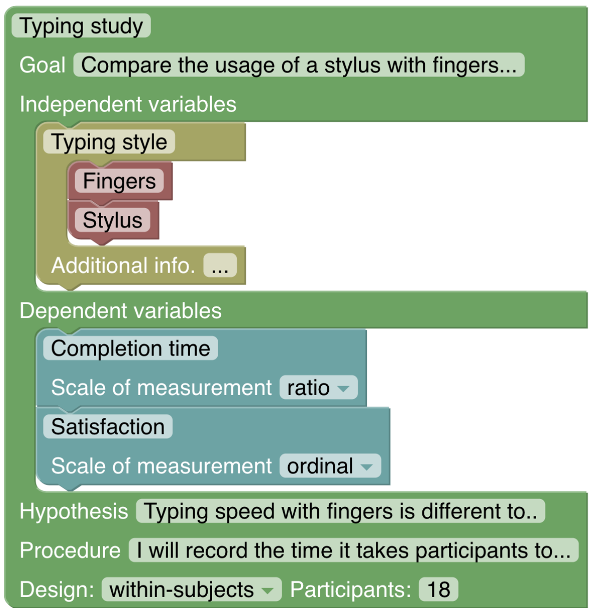


Figure 3.1: Experiment design workspace and toolbar

can edit the individual values on the blocks and connect them. With **a.** being the main design block, users can input their experiment goal, hypothesis, procedure, design, and sample size by selecting fields and editing them. Block **b.** represents the independent variables and features an editable name, while providing the ability to snap multiple red value blocks to the independent variable. Block **c.** is used to represent the dependent variables and features an editable name in addition to allowing the user to select the scale of measurement (nominal, ordinal, interval, or ratio) through a drop-down menu .

Snapping the yellow blocks into the gap of the main design block allows the user to add as many independent variables as required. Likewise, dependent variable blocks can be snapped into the gap underneath the label Dependent Variables. An example of completed experiment representation can be seen in Figure 3.2.



Compare the usage of a stylus with fingers...

Typing speed with fingers is different to..

I will record the time it takes participants to...

Figure 3.2: Sample of a completed design brick

3.3 Workflow

3.3.1 Experiment Information

The first window that opens in StatHunt is the experiment design. Here, the user is greeted by the chatbot and given the option to either use the interactive visualisation to create their experiment design, or provide the information via the chatbot. After the user completes their experiment design, they can either tell the chatbot they are done, or click on the dataset tab in the menu bar of the app, which brings them to the next step.

On the experiment design page, the user provides their experiment information using the chatbot or design blocks

The State Transition Network used to model dialogue for the window can be seen in Figure 3.3. To make sure that users do not have to go through unnecessary points of dialogue, they are provided with three separate options. The user can choose between starting a linear walk-through, providing a specific piece of information, or getting help. After selecting the linear walk-through, the chatbot asks the user to provide all pieces of experiment information se-

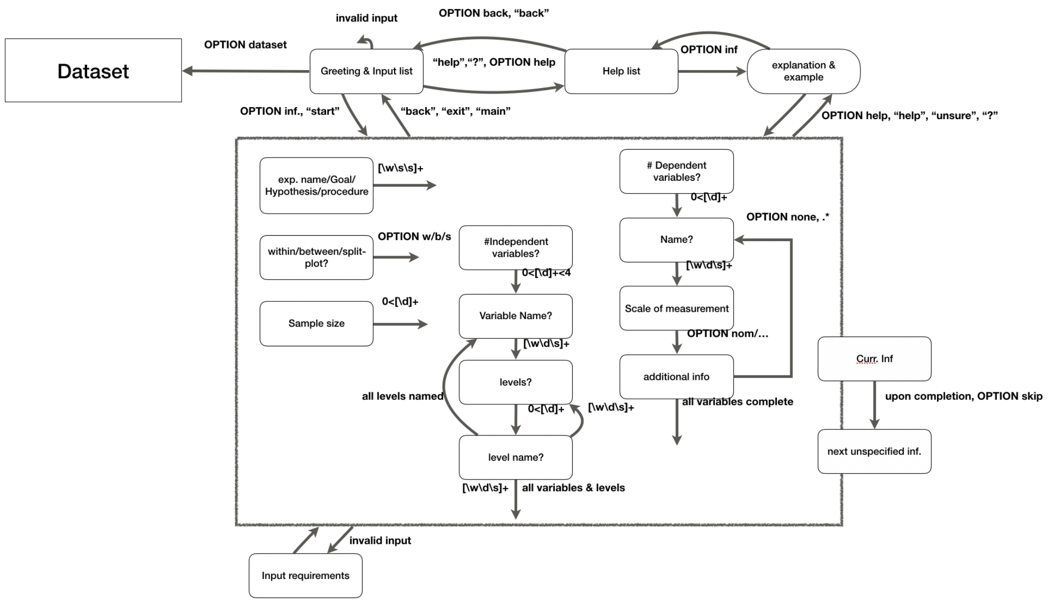


Figure 3.3: State Transition Network for experiment design

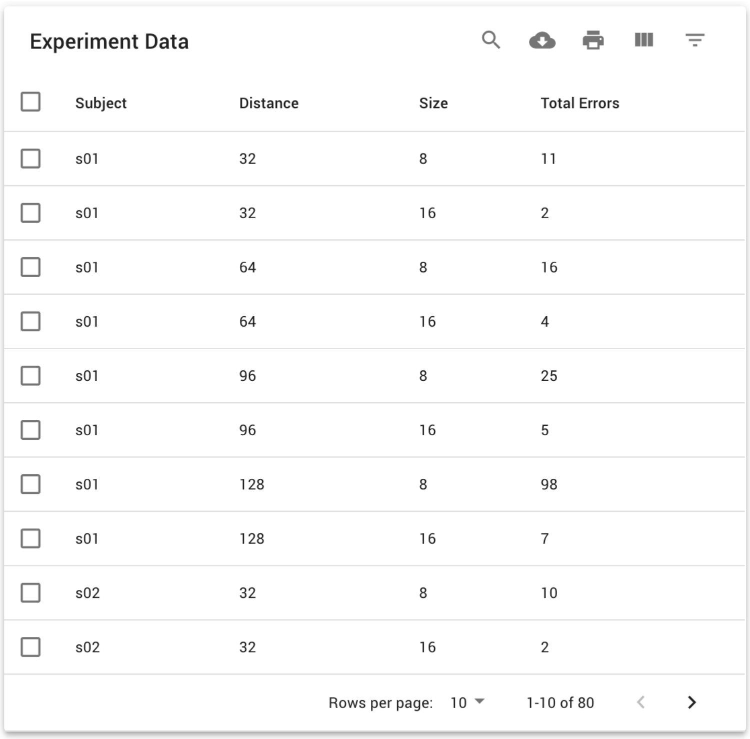
quentially. When typing in a specific piece of information, the user is questioned about it. After providing the information, the user is then brought back to the overview. If the user wants to just seek help, they are brought to a list of the different types of information they need to provide. When selecting one of these pieces of information, the user is provided with an explanation and an example. After these are given, the user is returned to the help list. If the user types "back", they are returned to the general overview.

The user is asked directly to provide all simple types of information. If they are unsure how to do so, they can type "help" and are immediately given an explanation and example. For the dependent and independent variables, the dialogue is slightly more complex. As a single experiment can have multiple independent variables, the user is first asked to state the amount. For each variable, the user then has to provide the name of the variable and how many levels that variable has. For each level, the user then has to provide the level name. Similar to the independent variables, one experiment can have multiple dependent variables. After asking the user to provide the amount of dependent variables the experiment has, the user is then asked to provide the scale of measurement and any additional information they might deem relevant.

3.3.2 Dataset Upload and Obfuscation

After completing the initial steps, the user is brought to the second pages, where they are prompted by the chatbot to upload their experiment data as a CSV file. After uploading the information, they can view and interact with their data over a table view as seen in Figure 3.4. Through the chatbot, users are then given options to obfuscate the variables and their values. Once the user is satisfied with the applied obfuscation, they can move on to the next step through the chatbot, or by pressing on the 'post question' tab. The tool-

Through a table-view, users can see and sort their uploaded data


Experiment Data

	Subject	Distance	Size	Total Errors
☐	s01	32	8	11
☐	s01	32	16	2
☐	s01	64	8	16
☐	s01	64	16	4
☐	s01	96	8	25
☐	s01	96	16	5
☐	s01	128	8	98
☐	s01	128	16	7
☐	s02	32	8	10
☐	s02	32	16	2

Rows per page: 10 1-10 of 80

Figure 3.4: Table view of experiment data

bar at the top of the table provides the user with different functionalities. The user can download a CSV version of their edited dataset, remove certain columns from the

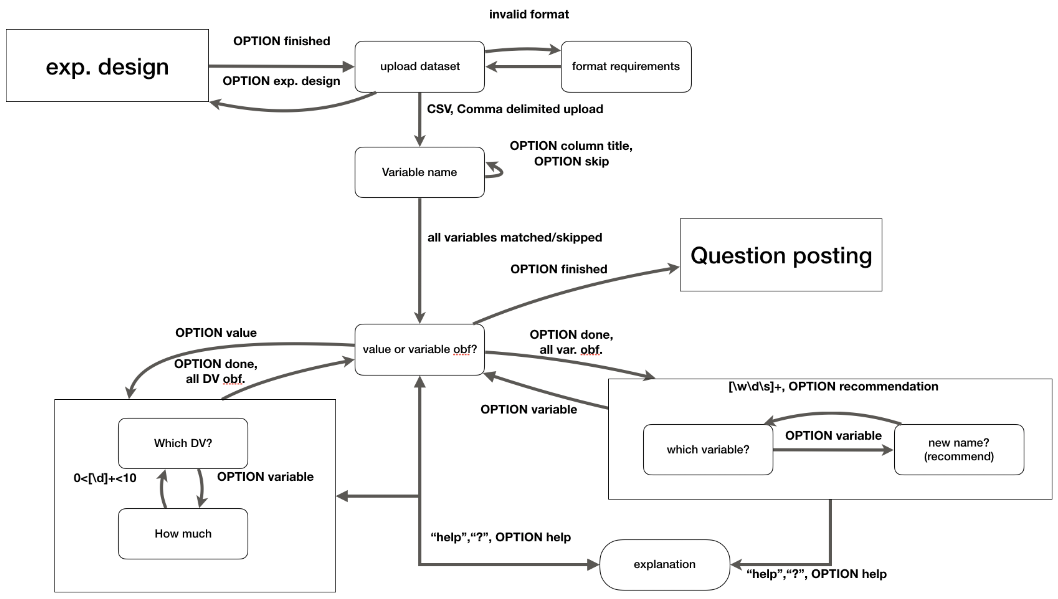


Figure 3.5: State Transition Network for dataset window

view, or filter based on column values. Using the search feature, the user can find specific rows. If the user has certain parts of the dataset that they are unwilling to share even when obfuscated, they can select them using checkboxes and delete them. To obfuscate the dataset, the user has to use the chatbot. As a model for the conversation we use the State Transition Network in Figure 3.5. If the user has entered the window through the chatbot, they are asked to upload their dataset. Once they have done so and confirmed it, the user is brought to an overview with several options. The user can then choose between obfuscating a column's values, obfuscating the variable names of a column, or moving on to the final question posting step of StatHunt.

Using the chatbot, users can obfuscate the data values or variables

For value obfuscation, the user is asked to specify the column name of the variable they want to obfuscate. The data in that column then goes through the obfuscation algorithm and the table re-renders with the new values.

For variable obfuscation, the user is also asked to specify the column they want to obfuscate. The chatbot then asks the user what they want to rename the variable to. After

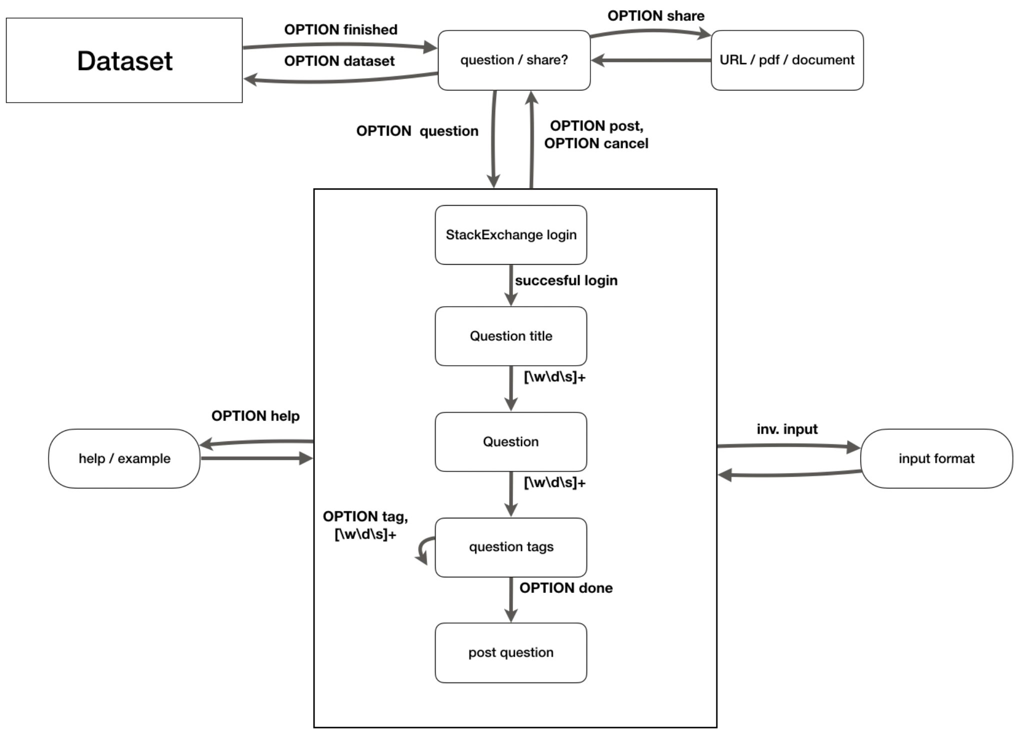


Figure 3.6: State Transition Network for question posting window

asking the user what they want to replace, each of the value names in the table then re-renders after StatHunt applies those changes. Once the obfuscation is applied, the user is returned to the overview. After applying obfuscation to each relevant column, the user can then move on to the final step.

3.3.3 Information Sharing and Question Posting

Once the user has completed all the changes they wanted to make in their dataset, they are brought to the final window of StatHunt. Here, the user can either get a URL to paste into their question, or go through the question posting process in StatHunt.

Once all information is added, users can get a URL to share it

The model used for the conversation in this window can be seen in Figure 3.6. Upon entry, the chatbot asks the user whether they want to use StatHunt to post their question on StackExchange, or just share their information. On choosing the latter, StatHunt generates a URL over which the obfuscated data and the visualised experiment design are viewable. The user is then given both options again. If the the former is chosen, the chatbot requests the user authenticate via StackExchange.

Once authenticated, the chatbot requests the question title and question format from the user. When the user types "help", the chatbot provides examples of properly formatted questions. In the final step, the chatbot asks the user to provide tags for their question, while recommending ones that are related to the experiment design. The question is then posted to StackExchange and the user is brought back to the overview.

Chapter 4

Implementation

4.1 Technologies

To develop the front-end of StatHunt, the Javascript framework React[1] was used. React applications are built using multiple components, each in control of their own logic and rendering. Due to this, projects developed with React are inherently modular. Since StatHunt is open-source, this modularity can aid future developers in their understanding of the code.

The three main frameworks used to build StatHunt were React, FastAPI and Botpress

For the back-end, FastAPI[2], a high-performance Python framework, was used. Developed as a micro-framework, FastAPI is a library that is primarily used to build API's. As such, it does not include things like database integration and other features that were not necessary for this project. Additionally, since FastAPI has automatic documentation generation, all written code will be properly documented.

As the core of user interaction was the chatbot, a framework that was as high-performing and flexible as possible was needed. Botpress[3] is not only a framework that fits these criteria, but also a complete development platform. By running Botpress as a server, one has access to an admin-

[1]https://reactjs.org/
[2]https://fastapi.tiangolo.com/
[3]https://botpress.com/

istration panel where one can create, test, and deploy chatbots. Through an interactive flow diagram, one can create complex dialogue with conditional transitioning between messages and the launching of Javascript code.

4.2 Architecture

The front-end of StatHunt is a React app, the back-and a FastAPI server and the chatbot is hosted with Botpress

StatHunt is split into three major components. The first component is the React front-end, which renders the user interface and handles user input. The second component is the FastAPI back-end, which stores experiment information and data in addition to providing a way for the front-end and bot server to communicate. The third component is the bot server. This handles all internal bot logic, manages user chat sessions, and provides the front-end with a loadable chat interface.

4.2.1 Authentication and Session Management

To ensure that user information is correctly allocated, StatHunt generates a unique ID on the first visit

When a user opens StatHunt, the front-end tries to find a user ID inside the browser's local storage. If no such ID is present (if a user is visiting the site for the first time), the front-end sends a request to the back-end, requesting one. The back-end then generates the user ID using Python's uuid library to ensure that the ID is unique and returns it.

After receiving the ID, the front-end stores it in the local browser. Using the generated ID, the front-end initialises the chatbot and connects it to the bot server. The bot server then registers a new user session with the given ID. For all successive requests applied to the back-end from either the front-end or the bot server, the user ID is appended. Through this, the server can clearly identify who is making the request and handle it appropriately.

As long as the browser's local storage remains uncleared, the user's current progress in StatHunt will automatically be loaded on startup.

4.2.2 Data Passing

Internal communication is handled using HTTP requests, with the user ID serving as an identifier

As the user is conversing with the chatbot, the information provided has to be visible in the front-end. Due to the way Botpress handles chatbot integration into websites, there is no direct way to send information to the front-end. Additionally, as the bot server is hosted and the front-end is rendered on the user's computer, a shared storage does not exist. Since the front-end and bot server have both been initialised with their user ID, however, the implemented back-end can act as common storage.

When the user makes changes in the front-end, these changes are taken, serialised into JSON, and then sent to the back-end. The data is sent as a HTTP request using the browser's built-in fetch function. An example of this can be seen in the following code snippet:

```
fetch('http://x.x.x.x:8000/post_design/' +
  localStorage.uid,
  {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(data)
    })
```

When information needs to be passed from the bot server to the back-end, the data is serialised to JSON and then sent to the back-end. The request is not done through fetch, but instead with the Javascript library axios[4]. An example of this can be seen below:

```
axios.post('http://x.x.x.x:8000/dataset/rcolumn/' +
    event.target,
    {
        'column': temp.column_name,
        'ncolumn': temp.ncolumn
        'values': temp.ncolumn_data
        })
```

Once the bot server makes changes to the data, the front-end can retrieve it in a similar fashion. A HTTP request is made to the back-end requesting the specific information:

```
fetch("http://localhost:8000/exp_design/" +
```

[4]https://github.com/axios/

```
    localStorage.uid)
    .then((response) => {
        return response.json()
    })
    .then((response) => {
        this.data = response
        this.syncWorkspace()
    });
```

4.2.3 Storage Model

The user's information is stored as a dictionary in the back-end

The user's experiment information and dataset are stored in a dictionary on the back-end server. After user ID generation, the back-end uses the user ID as a key to its entry in the dictionary. User data is split into two separate parts. Once the user requests an ID, the first part, the experiment information, is initialised as an object:

```
user_id = str(uuid.uuid1())
data[user_id] = {}
update[user_id] = True
data[user_id]['design'] = {
    'hypothesis': "",
    'goal_of_analysis': "",
    'procedure': "",
    'sample_size': "",
    'exp_design': "",
    'iv': [],
    'dv': [],
}
```

Once a file is uploaded by the user, the back-end stores it using the user ID. Using pandas[5], a python library for data science, the CSV file is parsed into a dataframe and stored:

```
async def create_upload_file(user_id: str,
    file: UploadFile = File(...)):
        data[user_id]['dataset'] = file.file
        data[user_id]['dataframe'] = pd.read_csv(
    file.file)
```

[5]https://pandas.pydata.org

4.3 Experiment Design

The implementation of experiment design is split into two parts. The first is the implementation of the chatbot and its dialogue. The second is the implementation of the Blockly workspace. To ensure that the information being displayed in the front-end consistently reflects the changes being made in both the chatbot and interactive workspace, a synchronisation mechanism was also implemented.

4.3.1 Chatbot

In Botpress' content management system, the flow creator was used to implement the planned dialogue. A dialogue flow consists of multiple state nodes that are linked to each other. As a user converses with the chatbot, the dialogue flow moves from one state to another, ensuring that a single state is always active.

The chatbot dialogue was developed using Botpress' content management system

Each state features an On Enter, On Receive, and Transition property. State actions can consist of content that the chatbot sends to the user via chat-interface or the execution of Javascript code. When a state is reached in the dialogue, all actions that are saved inside the On Enter property are called. Likewise, when user input is received, all actions saved in the On Receive property are called. After the actions in On Receive are called, the Transition property decides which the next state will be. The dialogue flow then activates the next state.

Experiment Information

For all basic pieces of experiment information the user provides to the chatbot, the implementation is similar. When the states are activated, the message included in the On Receive property is the question posed to the user. For the experiment design, the message that the bot sends is of the dropdown type, which sends a message and provides the user with a list of options in the chat interface. In the case of

When the user provides non-variable information, they are immediately sent to the back-end the hypothesis, analysis goal, sample size, and procedure, the On Receive property sends text.

Once the user types in their information, the chatbot calls the `saveVar` function depicted below:

```
const saveVar = async (type, value) => {
if (user.design == undefined){
        user.design = {};
        user.design["iv"] = [];
        user.design["dv"] = [];
    }
    user.design[type] = value;

    axios.post('http://x.x.x.x:8000/exp_design/', {
        user_id: event.target,
        variable: type,
        value: value
        })
}
```

`saveVar` first updates the user's design within the chatbot's local storage, then sends the information to the back-end. The back-end then updates the data saved under the user ID with the new information.

The disconnected state nodes for the hypothesis and experiment design can be seen in Figure 4.1.

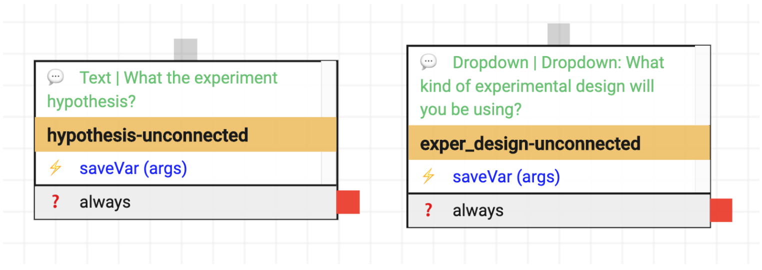
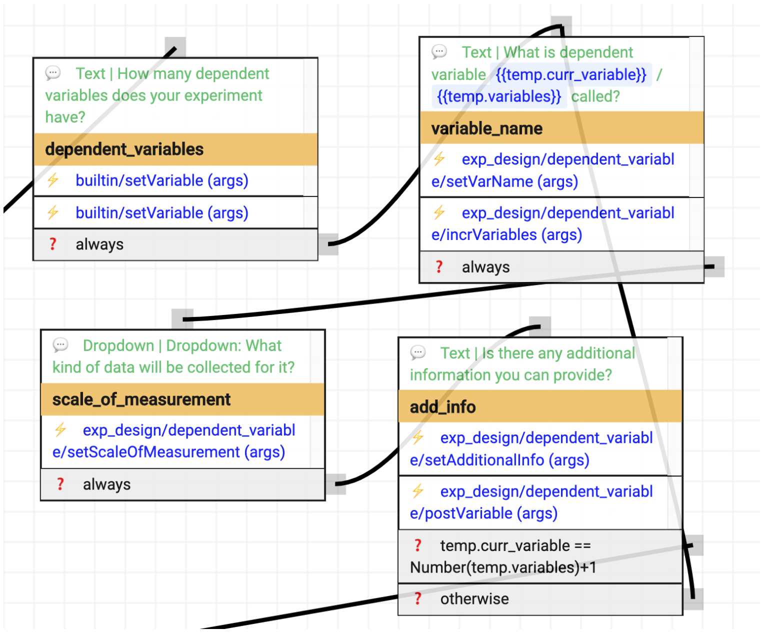

Text | What the experiment hypothesis?
hypothesis-unconnected
saveVar (args)
always
Dropdown | Dropdown: What kind of experimental design will you be using?
exper_design-unconnected
saveVar (args)
always

Figure 4.1: Hypothesis and experiment design nodes in Botpress

Dependent Variables

The dialogue for supplying dependent variable information is repeated until all variables are provided

For the dependent variables, the implementation is slightly more complex. This is due to a single experiment having multiple variables, each of which contain a scale of measurement property and possible additional information. The chatbot asks the user for the amount, then loops over the conversation required to acquire all relevant bits of information. The connected state nodes for this can be seen below in Figure 4.2.


Text | How many dependent variables does your experiment have?
dependent_variables
builtin/setVariable (args)
builtin/setVariable (args)
? always

Text | What is dependent variable {{temp.curr_variable}} / {{temp.variables}} called?
variable_name
exp_design/dependent_variable/setVarName (args)
exp_design/dependent_variable/incrVariables (args)
? always

Dropdown | Dropdown: What kind of data will be collected for it?
scale_of_measurement
exp_design/dependent_variable/setScaleOfMeasurement (args)
? always

Text | Is there any additional information you can provide?
add_info
exp_design/dependent_variable/setAdditionalInfo (args)
exp_design/dependent_variable/postVariable (args)
? temp.curr_variable == Number(temp.variables)+1
? otherwise

Figure 4.2: Dependent variable flow in Botpress

The first state node for the dependent variables asks the user to provide the amount of dependent variables. After receiving the amount, the On Receive property saves the this information in temporary storage and sets a `curr_variable` counter to 1.

In the variable_name state, the user receives a message asking them to provide the name of the current independent variable (taken from the `curr_variable` counter). Upon receiving the name, the following `setVarName` function is called:

```
const saveVar = async (value) => {
    if (user.design == undefined){
        user.design = {};
        user.design["iv"] = [];
        user.design["dv"] = [];
    }
    while(user.design["dv"].length < temp.variables)
    {
        user.design["dv"].push({});
    }

    user.design["dv"]
        [temp.curr_variable-1]
        ["name"] = value;
    console.log(user.design)
}
```

Using internal variables the bot server counts the amount of iterations the user has gone through

The function first checks if the user has already provided information on their experiment design. If not, it initiates it with empty values. Additionally, the `curr_variable` counter is incremented. It then makes sure that the array storing the dependent variable objects has the proper length. Finally, it assigns the name of the variable to the current independent variable object.

The next active state is scale_of_measurement. Here, On Enter sends the user a message asking for the data type, while providing the different possibilities in the form of a dropdown menu. Upon receipt, the function `setScaleMeasurement` is called:

```
const setScaleOfMeasurement = async (value) => {
    user.design["dv"]
        [temp.curr_variable-2]
        ["scale_of_measurement"] = value;
}
```

Due to the incrementation of the `curr_variable`, it's value is decreased by two, in order to access the right object.

The final state add_info asks the user for any additional information they want to provide. After receiving an answer, the On Receive property calls the function `setAdditionalInfo`, which saves the information. Once this is done `postVariable` is called:

```
const postVariable = async () => {
    axios.post('http://localhost:8000/exp_design/dv'
    , {
        user_id: event.target,
        name: user.design['dv']
            [temp.curr_variable-2]
            ['name'],
        measurement: user.design['dv']
            [temp.curr_variable-2]
            ['scale_of_measurement'],
        add_info: user.design['dv']
            [temp.curr_variable-2]
            ['additional_info']
        })
    }
```

The function takes the current dependent variable and sends it to the back-end. In the Transition property, the `curr_variable` counter is compared with the total amount of variables. If all the variables have been added, the dependent variable loop is left. If all variables have not been added, the `variable_name` state is reactivated and the user can provide information for further variables.

Independent Variables

Since an experiment can have multiple independent variables, and each has multiple different levels, the dialogue needs to loop over both. The state nodes and transitions for the implementation can be seen in Figure 4.3.

For the independent variables, the bot server needs to additionally loop over the levels

The first activated state independent_variables asks and obtains the amount of factors that the user's experiment contains. It then saves that amount in the temporary storage and initialises the counter `curr_fact` as 1.

The next state factor_name asks the user for the name of the current factor. Once received, On Receive calls the function `setVarName`:

```
const setVarName = async (value) => {
    if (user.design == undefined){
        user.design = {};
        user.design["iv"] = [];
        user.design["dv"] = [];
    }
    while(user.design["iv"].length < temp.factors){
        user.design["iv"].push({});
    }
```

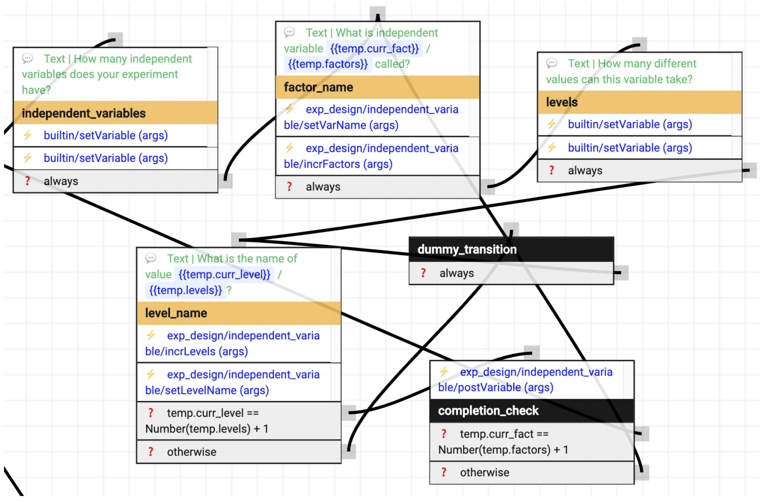

Text | How many independent variables does your experiment have?
independent_variables
builtin/setVariable (args)
builtin/setVariable (args)
always
Text | What is independent variable {{temp.curr_fact}} / {{temp.factors}} called?
factor_name
exp_design/independent_variable/setVarName (args)
exp_design/independent_variable/incrFactors (args)
always
Text | How many different values can this variable take?
levels
builtin/setVariable (args)
builtin/setVariable (args)
always
dummy_transition
always
Text | What is the name of value {{temp.curr_level}} / {{temp.levels}} ?
level_name
exp_design/independent_variable/incrLevels (args)
exp_design/independent_variable/setLevelName (args)
temp.curr_level == Number(temp.levels) + 1
otherwise
exp_design/independent_variable/postVariable (args)
completion_check
temp.curr_fact == Number(temp.factors) + 1
otherwise

Figure 4.3: Independent variable flow in Botpress

```
    user.design["iv"][temp.curr_fact-1]["name"] =
    value;
}
```

This variation of `setVarName` is identical to the one introduced in the previous section. Instead of expanding the storage to fit the amount of dependent variables, and saving the variable name to the current dependent variable, it does so for the independent variables. Additionally, On Receive increments the `curr_fact` counter.

Levels, the following state, asks the user to provide the number of levels the current factor has. Once provided, On Receive stores the value in addition to setting a further counter called `curr_level` to 1.

In the `level_name` state, the user is then asked what the name of the current level is. On Receive first increments the `curr_level` counter, then calls the following `setLevelName` function:

```
const setLevelName = async (value) => {
    if(temp.curr_level == 2){
        user.design["iv"][temp.curr_fact-2]["levels"
    ] = []
    }
    user.design["iv"][temp.curr_fact-2]["levels"].
    push(value);
    console.log(user.design);
}
```

If this is the first level that is being provided for the variable (if `curr_level` is 2), the function then initialises the array. Following this, it adds the provided level name to the levels array for the variable.

If there are still additional levels that have not been provided, the Transition property activates the dummy_transition state, which then reactivates the level_name state. Otherwise, the completion_check state is activated. On Enter then directly calls the following `postVariable` function:

```
const postVariable = async () => {
    axios.post('http://x.x.x.x:8000/exp_design/iv',
    {
        user_id: event.target,
        name: user.design['iv']
            [temp.curr_fact-2]
            ['name'],
        levels: user.design['iv']
            [temp.curr_fact-2]
            ['levels'],
        })
}
```

This sends the current independent variable name and levels to the back-end for storage. If all independent variables have been sent, the independent variable flow is exited. Otherwise, the factor_name state is activated for the user to add their next independent variable.

4.3.2 Blockly

To implement the visual representation Google's Blockly library was used. Originally developed for the creation of visual programming languages, it has been used to create custom blocks, render the interactive experiment design interface, and create a toolbar.

Block Creation

Using Blockly's developer tools, the different blocks used in StatHunt were created

To create each of the four custom blocks, Blockly's developer tools were used. These tools allow the creation of blocks by drag and dropping different types. After creation, custom-made blocks were exported as Javascript code to use in the front-end.

The custom blocks created using the developer tools can be seen in the following figures. Figure 4.4 is the brick created to generate the experiment design block seen in Figure 3.1. The labels in the design brick were created by connecting

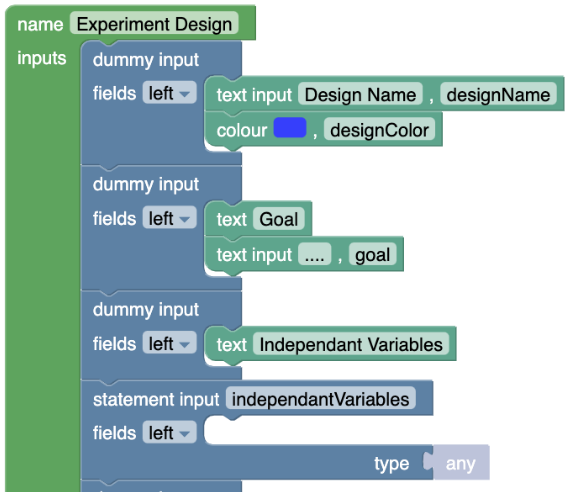
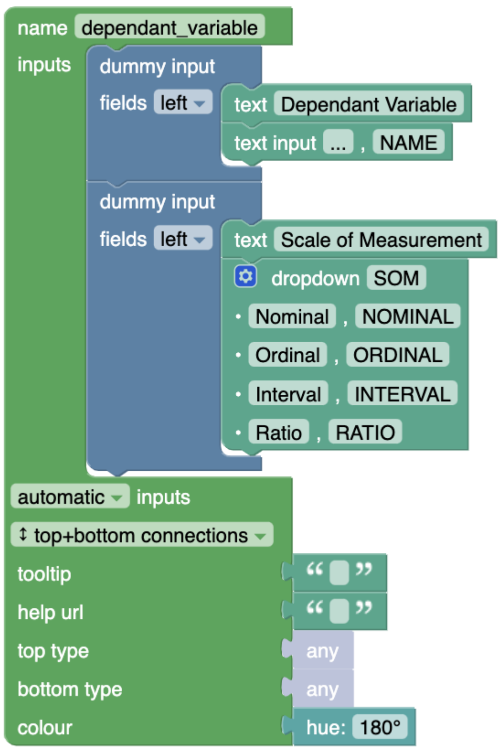


Figure 4.4: Blockly part of experiment design generator brick

the turquoise text bricks to the blue dummy input bricks. For text input like the hypothesis, procedure, and goal of analysis, the turquoise text input bricks were connected to the dummy input. Since the experiment design is restricted to one of three options, a dropdown brick outlining the different possibilities was also connected. For the sample size (labeled participants), a numeric input brick which only accepted natural numbers was implemented. To create a place for the independent variable and dependent variable blocks to snap to, the statement input brick was snapped to the dummy input. To ensure that the experiment blocks were not stack- or connectable, the connection option was set to no connections.

In Figures 4.6 and 4.5, the bricks used to generate the independent and dependent variable blocks are featured. Since a single experiment can have multiple dependent and independent variables, top and bottom connections were enabled. For the dependent variable block, a text input



```
name dependant_variable
inputs  dummy input
        fields left ▾   text Dependant Variable
                        text input ... , NAME
        dummy input
        fields left ▾   text Scale of Measurement
                        dropdown SOM
                        • Nominal , NOMINAL
                        • Ordinal , ORDINAL
                        • Interval , INTERVAL
                        • Ratio , RATIO
automatic ▾ inputs
↕ top+bottom connections ▾
tooltip       " "
help url      " "
top type      any
bottom type   any
colour        hue: 180°
```

Figure 4.5: Blockly dependent variable generator brick

brick was used to handle the name. A dropdown brick was used for the scale of measurement. Since the information needed for the independent variables consists of multiple levels, each with their own values, the statement input brick was used. This allows multiple level bricks to be attached. The name field for the independent variable block is added through a text input brick, as is the additional information field.

To implement the generated blocks into the front-end, the developer tools were used to generate Javascript code. The

The JavaScript generated by the Blockly developer tools were pasted into the front-end

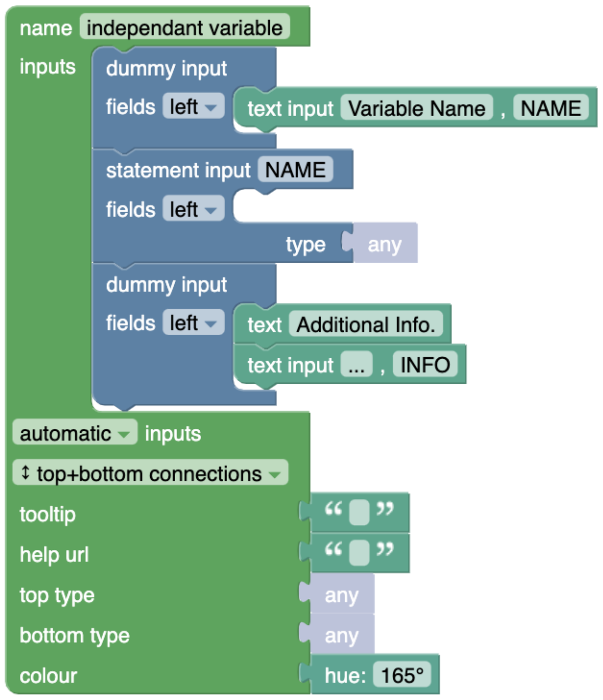


Figure 4.6: Blockly independent variable generator bricks

code was then added into the React app and saved to the blockly library's Blocks object, making it accessible through the front-end. An example of how adding the independent variable works can be seen below:

```
Blockly.Blocks['independent_variable'] = {
    init: function () {
        this.appendDummyInput("name")
            .appendField(
                new Blockly.FieldTextInput("IV name"),
                 "NAME");
        this.appendStatementInput("variables")
            .setCheck(null);
        this.appendDummyInput("add_info")
            .appendField("Additional info.")
            .appendField(
                new Blockly.FieldTextInput("..."),
                 "INFO");
        this.setPreviousStatement(true, null);
        this.setNextStatement(true, null);
        this.setColour(60);
        this.setTooltip("");
        this.setHelpUrl("");
    }
};
```

Workspace Integration

To integrate the Blockly workspace (featuring the toolbar and custom blocks) the sample BlocklyComponent, created by the developers of Blockly, was used. By passing the component, one can control how it is initialized, and whether it is scrollable or read only. The code used to render it was:

A pre-made Blockly component for React was used for integration into the front-end

```
<BlocklyComponent ref="blocklyComponent" readOnly={
    false} move={{
    scrollbars: true,
    drag: true,
    wheel: true
    }} initialXml={``}>
    <Block type="experiment_design" />
    <Block type="independent_variable" />
    <Block type="variable" />
    <Block type="dependent_variable" />
</BlocklyComponent>
```

To include the custom Blocks in the toolbar, they were added as child components to the imported Blockly-Component. By passing it the reference prop of Blockly-Component, it can be accessed throughout the rest of the component. Using this stored reference, it is possible to read what blocks are currently in the workspace, in addition to editing and even deleting them.

Storing Experiment Data

Through the reference object BlocklyComponent, the workspace and all of the blocks contained in it are accessible. Using this, it is possible to go through the workspace, locate the relevant blocks, and parse their information into a proper format. With this formatting, the information can be sent to the back-end where it is loaded into the shared storage.

The function `sendDesignData` in the React component

Experiment information is parsed from the Blockly workspace and sent to the back-end that renders the Blockly workspace is the main handler for this. The first part of the function's code can be seen below:

```
var block = this.refs.blocklyComponent
.primaryWorkspace.getTopBlocks()[0]

if (block === undefined) {
    this.syncWorkspace()
    block = this.refs.blocklyComponent
    .primaryWorkspace.getTopBlocks()[0]
}

var data = {}

data['hypothesis'] = block.getInput('hypothesis')
.fieldRow[1].getValue()
data['goal_of_analysis'] = block.getInput('goal')
.fieldRow[1].getValue()
data['procedure'] = block.getInput('procedure')
.fieldRow[1].getValue()
data['sample_size'] = block.getInput('dss')
.fieldRow[3].getValue()
data['exp_design'] = block.getInput('dss')
.fieldRow[1].getValue()
```

Using the `blocklyComponent` reference, the function retrieves the experiment design block from the workspace. If the block has not been initialised yet, `syncWorkspace` is called. This initialises the workspace from the back-end. The data object, which is used to store all the experiment information, is then initialised. All information, excluding the variables, is then added to the object in a similar fashion. The relevant input object is retrieved through the tags defined in the bricks seen in Figure 4.4. After this, the value is retrieved from the specific field and saved.

To parse the dependent variable `sendDesignData` runs the following code:

```
var dv = []
var conn = block.getInput('dependentVariables').
   connection
var name
while (conn.targetBlock() != null) {
    name = conn.targetBlock().getInput('name')
    .fieldRow[0].getValue()
    var measurement = conn.targetBlock().getInput('
   scale_of_measurement')
    .fieldRow[1].getValue()
    dv.push({ 'name': name, 'measurement':
```

```
    measurement })
    conn = conn.targetBlock().nextConnection
}
data['dv'] = dv
```

Since the dependent variables are individual blocks that snap onto the experiment design block, their values can not be directly retrieved. After initialising the array `dv`, the connection for the dependent variable blocks is saved as `conn`. From the dependent variable block (that is snapped onto the connection stored in `conn`), the variable name and scale of measurement are retrieved and added to the `dv` array. The variable `conn` is then updated to the next connection. Once there are no more blocks connected to `conn`, the `dv` array is saved to the `data` object.

To access the dependent and independent variables, the Blockly connection object is used

For independent variables the following code is run:

```
var iv = []
conn = block.getInput('independentVariables').
    connection
while (conn.targetBlock() != null) {
    name = conn.targetBlock().getInput('name').
    fieldRow[0].getValue()
    var levels = []
    var vconn = conn.targetBlock().getInput('
    variables').connection
    while (vconn.targetBlock() != null) {
        levels.push(vconn.targetBlock().getInput('
    name')
        .fieldRow[0].getValue())
        vconn = vconn.targetBlock().nextConnection
    }
    iv.push({ 'name': name, 'levels': levels })
    conn = conn.targetBlock().nextConnection
}
data['iv'] = iv
```

Similarly to the dependent variable parsing, the connection for the independent variable blocks is saved as `conn`. The variable name is retrieved from the target block of the connection `conn` and an array, `levels`, is initialised. The connection for the value blocks is then initialised as `vconn`. Information from the value block connected to `vconn` is then added to the levels array, after which `vconn` is updated to the next value connection. Once there are no more values, the variable name and levels array are appended to the iv array. Finally, conn is updated to the next independent variable connection.

After all the independent variables have been added to the iv array and it has been saved into data, the final piece of code is run:

```
this.data = data
fetch('http://localhost:8000/post_design/' +
    localStorage.uid, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(data)
})
```

The data object is saved into the component storage before being sent to the back-end, where it is then finally saved into the shared storage.

4.3.3 State Synchronisation

Since the back-end is used as the shared storage for a user's experiment data, the front-end also needs to be able to retrieve the information and render it. In this way, any changes the chatbot makes to the stored experiment information can be loaded into the workspace. Due to the user being able to edit the design in the Blockly workspace, StatHunt needs to decide between reading and writing the data.

On startup, the component that renders the workspace runs the following code:

```
fetch("http://x.x.x.x:8000/exp_design/" +
    localStorage.uid)
        .then((response) => {
            return response.json()
        })
        .then((response) => {
            this.data = response
            this.syncWorkspace()
        });

window.myInterval = setInterval(() => {
    this.updateData()
}, 500)
```

Using fetch, the experiment design data is taken from the back-end and saved into the component storage's data object. Then the experiment blocks in the workspace are initialised using the data with the `syncWorkspace` function. Finally, a scheduler is set that calls the `updateData` function every half-second.

In regular intervals the front-end checks if the bot server has made changes to the shared storage

When called, `updateData` handles the update of either the storage in the back-end or the rendering of blocks in the workspace to reflect that storage. It runs the following code to handle this:

```
fetch("http://x.x.x.x:8000/update/"
+ localStorage.uid)
    .then((response) => {
    return response.json()
    })
    .then((response) => {
    if (response) {
        fetch("http://localhost:8000/exp_design/"
        + localStorage.uid)
        .then((response) => {
            return response.json()
        })
        .then((response) => {
            this.data = response
            this.syncWorkspace()
        });
    }else{
        this.sendDesignData()
    }
})
```

First, the fetch request is sent to the back-end asking whether or not the bot server has recently made a change. If this is not the case, `sendDesignData` is called to save the current state of the workspace in the back-end.

If the user has recently provided information through the chatbot, the data from storage is taken and rendered onto the workspace. This is done in the `syncWorkspace` function, which runs the following code:

The syncWorkspace function retrieves information from the back-end and renders it into the Blockly workspace

```
    if (this.refs.blocklyComponent.primaryWorkspace.
    getTopBlocks().length === 0) {
        var initXml = `<xml><block type="
    experiment_design"></
        block></xml>`
        Blockly.Xml.domToWorkspace(Blockly.Xml
        .textToDom(initXml),
        Blockly.getMainWorkspace());
      }
```

To begin, `syncWorkspace` checks whether the workspace is empty. If this is the case, XML is used to define an experiment design block. The design block is then used to initialise the workspace using the `domToWorkspace` function provided by Blockly.

```
var block = this.refs.blocklyComponent.
    primaryWorkspace
.getTopBlocks()[0]

var goal = block.getInput('goal')
goal.fieldRow[1].setValue(this.data['
    goal_of_analysis'])

var hypothesis = block.getInput('hypothesis')
hypothesis.fieldRow[1].setValue(this.data['
    hypothesis'])

var procedure = block.getInput('procedure')
procedure.fieldRow[1].setValue(this.data['procedure'
    ])

var design = block.getInput('dss')
design.fieldRow[1].setValue(this.data['exp_design'])
design.fieldRow[3].setValue(this.data['sample_size'
    ])
```

After fetching and saving the main experiment design block, all experiment information is written into it in a fashion similar to its retrieval in the `sendDesignData` function. Instead of `getValue`, the function `setValue` is called and the data is taken from the component's data object.

For the dependent variables, the creation and appending of objects was handled using Blockly's connection objects.

```
while (block.getInput('dependentVariables').
    connection.targetBlock() != null) {
    block.getInput('dependentVariables').connection.
    targetBlock().dispose(true)
}
for (var i = 0; i < this.data['dv'].length; i++) {
    var dvblock = this.refs.blocklyComponent
    .primaryWorkspace.newBlock('dependent_variable')
    dvblock.initSvg()
    dvblock.render()
    dvblock.getInput('name').fieldRow[0]
    .setValue(this.data['dv'][i]['name'])
    dvblock.getInput('scale_of_measurement')
    .fieldRow[1].setValue(this.data['dv'][i]['
    measurement'])

    block.getInput('dependentVariables').connection.
    connect(dvblock.previousConnection)
}
```

The code first loops through all dependent variable blocks and removes them from the workspace. Then, for each of the dependent variables sitting in the data object, a new dependent variable is initialized and saved as `dvblock`. To ensure it renders in the workspace, the SVG representation of the block is initialised, after which the render function is called. Using the information stored in the `data` object, the variable name and scale of measurement are set. Finally, the block is attached to the main experiment design block.

To render the independent variables, `syncWorkspace` runs the following code:

```
while (block.getInput('independentVariables')
.connection.targetBlock() != null) {
    block.getInput('independentVariables')
    .connection.targetBlock().dispose(true)
}
```

To first clear out the current independent variables, the function loops through the independent variable connection of the experiment design block and removes them.

```
for (i = 0; i < this.data['iv'].length; i++) {
    var ivblock = this.refs.blocklyComponent
    .primaryWorkspace.newBlock('independent_variable
    ')
    ivblock.initSvg()
    ivblock.render()
    ivblock.getInput('name').fieldRow[0]
    .setValue(this.data['iv'][i]['name'])

    var ivconnection = ivblock.getInput('variables')
    .connection
    for (var j = 0;
     j < this.data['iv'][i]['levels'].length;
      j++) {
        var valblock = this.refs.blocklyComponent
        .primaryWorkspace.newBlock('variable')
        valblock.initSvg()
        valblock.render()
        valblock.getInput('name').fieldRow[0]
        .setValue(this.data['iv'][i]['levels'][j])

        ivconnection.connect(
            valblock.previousConnection)
    }

    block.getInput('independentVariables')
    .connection.connect(ivblock.previousConnection)
    }
}
```

After this, a new independent variable block is initialised and saved as `ivblock`. Similar to the dependent variable block, the SVG is initialised and the render function is called. Then the independent variable name is taken from the data object and set.

Since the different values are connected to the independent variable block, the relevant connection is saved as `ivconnection`. For each related level found in the data object, the level block is rendered, its name set, and connected to the independent variable block.

Finally, the independent variable block is connected to the experiment design block, or the previously added independent variable.

4.4 Dataset

After data is uploaded to the front-end, it is sent to the back-end

The implementation of dataset obfuscation is inherently simpler than that of the experiment design. The table is rendered in the front-end, offering functions like filtering, sorting, and searching through the data. Meanwhile, the manipulation of the dataset through obfuscation occurs completely in the back-end. To upload the user dataset, the DataWindow component in the front-end renders a file upload button. Once a file is selected by the user that is different to the one currently uploaded, the input field calls the components `fileUpload` function:

```
fileUpload() {
    var data = new FormData()
    data.append("file", document.getElementById('
    file-upload').files[0])
    fetch('http://x.x.x.x:8000/uploadfile/' +
    localStorage.uid, {
        method: 'POST',
        body: data,
    })
        .then(response => response.json())
        .then(success => {
            console.log("File Succesfully Uploaded")
        })
        .catch(error => console.log(error)
        );
}
```

`fileUpload` takes the file that is currently being saved in the file-upload input and appends it to a FormData object. This is then sent to the back-end using fetch. It is logged upon successful completion.

4.4.1 Datatable

To implement the datatable shown in Figure 3.4, the React library, mui-datatables[6] was used. After loading the dataset from the back-end, the information is rendered in the table view. With the regular reloading of the data, it is ensured that any changes made to it through the chatbot are reflected.

The data saved in the back-end is retrieved and rendered in the front-end using mui-datatables

When the component that renders the datatable is about to be loaded, the following `componentWillMount` function is called:

```
componentWillMount() {
    if(localStorage.dataset === null ||
    localStorage.dataset === undefined){
        localStorage.dataset = '{}'
    }
    this.updateDataset();
    window.datasetInterval = setInterval(() => {
        this.updateDataset()
    }, 1000)
}
```

[6]https://github.com/gregnb/mui-datatables

If the user has not loaded the data before, the browser's local storage will be empty. To ensure that the empty datatable is still rendered instead of displaying an error, is is initialized as an empty object. A timer is then set to schedule repeated calls of the `updateDataset` function. This uses the following piece of code to retrieve and save the dataset into the browser's local storage:

```
fetch("http://x.x.x.x:8000/dataset/"
+ localStorage.uid)
    .then((response) => {
        return response.json();
    })
    .then((response) => {
        if (localStorage.dataset !== response
        && response !== '') {
            localStorage.dataset = (response);
        }
    })
```

To retrieve the data from the back-end, the fetch function is used. After obtaining a response, the data is parsed in JSON form and save it to the browser's local storage.

Since mui-datatables requires the column names and rows to be provided in the form of a string array and a two-dimensional array, the data needs to be re-formatted. In the render function of the component, the following code is used:

```
var dataset = Object.entries(
    JSON.parse(localStorage.dataset))
var rows = []
var columns = []

if(dataset.length !== 0){
    for (var i = 0; i < Object.entries(
    dataset[0][1]).length; i++) {
        var row = []
        for (var x in dataset) {
            row.push(dataset[x][1][i.toString()])
        }
        rows.push(row)
    }
    for (var y in (dataset)) {
        columns.push(
            dataset[y][0]
        )
    }
}
```

Using `Object.entries`, the `dataset` object is serialised and turned into an array containing key value entries in the form of an array. To save the columns, all key entries are appended into `dataset` to the columns array. Each individual row is passed through. A value entry is taken for each column and appended into the rows array.

When rendering the datatable, these two are passed as props into the mui-datables component.

```
<MUIDataTable
            title={"Experiment Data"}
            data={rows}
            columns={columns}
        />
```

4.4.2 Dataset Obfuscation

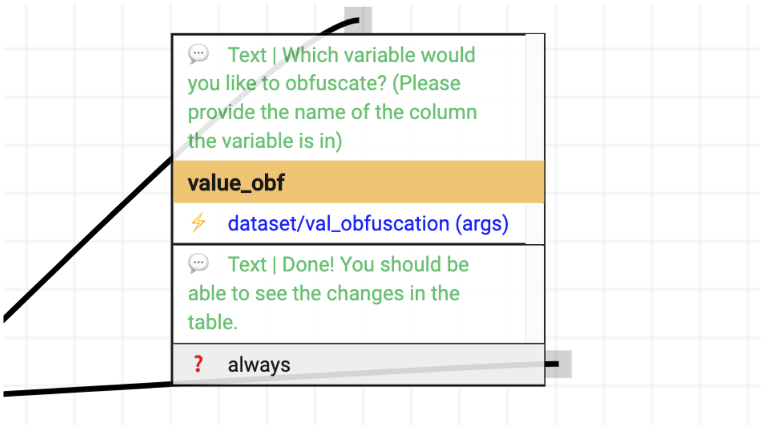
To obfuscate, the user has two separate options: obfuscating the variable names, or obfuscating the values. After the user decides which of the two they want to perform, they enter their information into the chatbot. The bot server then sends this information to the back-end, where the data is obfuscated. When the table on the page retrieves the dataset, the newly obfuscated data is loaded and rendered.

Similarly to the experiment design, the conversation handling uses Botpress' content management system.

Value Obfuscation

For value obfuscation, dialogue implementation was simple. The state node used to handle this can be seen in Figure 4.7. When the user enters the state, the On Enter parameter sends the user a question asking for the column they want to obfuscate. After providing it, On Recieve calls the following `val_obfuscation` function:

```
const val_obfuscation = async (column) => {
    axios.post('http://x.x.x.x:8000/dataset/
    obfuscate/'
```



Text | Which variable would you like to obfuscate? (Please provide the name of the column the variable is in)

value_obf

dataset/val_obfuscation (args)

Text | Done! You should be able to see the changes in the table.

always

Figure 4.7: Botpress value obfuscation node

```
    + event.target, null, {
        params: {
            'column': column
        }
        }).then(function (response) {
        console.log(response);
        })
        .catch(function (error) {
        console.log(error.response.data);
        });
    }
```

Using the response the user has set as the value column, the column name is passed on to the back-end. Following this, On Receive sends the user a confirmation. Then the previous state is reactivated.

When the back-end receives the request and column name from the bot server the following function is called:

```
async def obf_data(user_id: str, column: str):
    df = data[user_id]['dataframe']
    if(column not in df.columns):
        return 'Not a valid column'
    laplace = Laplace()
    laplace.set_epsilon(1)
    if df[column].dtype == 'int64':
        laplace.set_sensitivity(df[column].mean()
    *0.1)
        df[column] = df[column].apply(laplace.
    randomise).round(0).astype(int)
    elif df[column].dtype == 'float64':
        laplace.set_sensitivity(df[column].mean()
    *0.1)
        df[column] = df[column].apply(laplace.
    randomise)
return 'Column has been obfuscated'
```

From diffprivlib[7], a library for differential privacy created by IBM, the laplace object is imported and then initialised. Preferences for the generated noise are applied to the object by calling the `set_epsilon` and `set_sensitivity` methods. Using the pandas apply object, the additive laplace noise is applied to each value in the column. If the column values are whole numbers, the newly obfuscated values are rounded and the type is enforced onto the edited column.

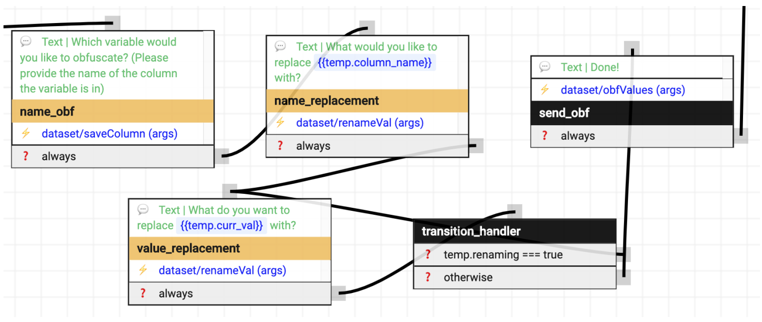
To obfuscate a column's values, the back-end applies laplace noise using the diffprivlib library

Variable Obfuscation

Dialogue implementation for variable obfuscation is slightly more complicated than for value obfuscation, as the user needs to provide replacements for the variable name and each associated value.

To obfuscate the variables, the dialogue loops through each variable that needs to be replaced

The flow created for variable obfuscation in Botpress can be seen below in Figure 4.8. In the first activated state



Text | Which variable would you like to obfuscate? (Please provide the name of the column the variable is in)
name_obf
dataset/saveColumn (args)
always

Text | What would you like to replace {{temp.column_name}} with?
name_replacement
dataset/renameVal (args)
always

Text | Done!
dataset/obfValues (args)
send_obf
always

Text | What do you want to replace {{temp.curr_val}} with?
value_replacement
dataset/renameVal (args)
always

transition_handler
temp.renaming === true
otherwise

Figure 4.8: Botpress variable obfuscation blocks

`name_obf`, the user is asked for the variable name. After receiving it, the On Receive property calls the `saveColumn` function which runs the code shown below:

[7]https://github.com/IBM/differential-privacy-library

```
axios.get('http://x.x.x.x:8000/dataset/column/' +
    event.target, {
    params: {
        'column': column
    }
    })
    .then(function (response) {
        temp.column_data = (response.data.values);
    })
    .catch(function (error) {
        console.log(error);
    })
temp.column_name = column
temp.ncolumn_data = []
temp.renaming = true
console.log(temp)
```

The function first requests the column values, which are returned by the back-end as an array. This value is then saved to the bot server's temporary storage. Furthermore, the column name is saved, and a new array to store replacement variable names is initialised. The temporary variable renaming is set to true, which later allows the determination of when the user has completed the renaming process.

After this, the name_replacement state is activated, asking the user what they want to replace the variable name with. The function `renameVal` then calls the following code (with title being set as 'true' and the replacement name):

```
change_title = title === 'true'
if(change_title){
    temp.ncolumn = new_name
}else {
    temp.ncolumn_data.push(new_name)
}
if(temp.column_data.length === 0){
    temp.renaming = false
    return
}
temp.curr_val = temp.column_data[0]
temp.column_data = temp.column_data.slice(1)
```

Because of the value of title, the replacement name is saved as `ncolumn`. The value that needs to be replaced next is saved into temporary storage as `curr_val` and is removed from the `column_data` array.

After this, the state value_replacement is called, which asks the user what they want to replace the current variable value with. Once received, the function rename is called again, this time with title being set as 'false'. Due to this, the function appends the replacement value to the `ncolumn_data` array. If all values have been taken out of `column_data`, there are no more column variables to replace and renaming is set to false.

In transition_handler, the temporary variable renaming is checked. If the value is set to true, there are still more variables to replace and value_replacement is reactivated. Otherwise, the user has provided their replacement for each of the variables and send_obf is activated.

send_obf sends the user a confirmation that obfuscation is complete and calls `obfValues` which runs the following code:

```
axios.post('http://x.x.x.x:8000/dataset/rcolumn/'
+ event.target,
    {'column': temp.column_name,
     'ncolumn': temp.ncolumn,
      'values': temp.ncolumn_data})
    .catch(function (error) {
        console.log(error);
    })
```

The new column name and new column data are sent to the back-end. The back-end then uses the following code to replace the relevant values:

```
old_values = data[user_id]['dataframe'][info.column
    ].unique()
mapping = {}
for i in range(len(info.values)):
    mapping[old_values[i]] = info.values[i]
data[user_id]['dataframe'][info.column] = data[
    user_id]['dataframe'][info.column].map(mapping)
data[user_id]['dataframe'] = data[user_id]['
    dataframe'].rename(columns={info.column: info.
    ncolumn})
print(data[user_id]['dataframe'][info.ncolumn])
```

With `info` being the object passed from the bot server to the back-end, the relevant column's unique values are saved. Then, the dictionary `mapping` is created, which uses the old values as keys and the new ones as entries. Column values are re-written through the mapping using the dataframe’s rename method. Finally, the column title is changed to the new one.

Chapter 5

Summary and Future Work

5.1 Summary and Contributions

To help inexperienced researchers receive support in statistical test selection, StatHunt, an interactive web-application was designed and implemented. In order to ensure that StatHunt tackles the problems help-seeking researchers face on Q&A websites, the recommendations provided by Hu [2019] were used as a baseline.

In the section Background and Related Work, the problems researchers have when posting to Q&A questions were outlined. After this, tools for representing experiment design were introduced, some background information on chatbots was provided, and a definition and approach to data obfuscation was explained. Finally, we briefly discussed two pre-existing tools that were built to support users with inferential statistics.

In the section StatHunt, the features of a system to aid researchers in posting questions were defined. The conversational design of a chatbot that can guide users through providing their experiment information, obfuscating their data, and posting a question, was introduced. Additionally, the use of bricks to create a structural representation of experiment data was presented.

In the Implementation section of this work, the development of StatHunt was detailed. The use of Botpress to model the conversation of the previously introduced State Transition Networks was outlined. The implementation of the Blockly workspace and creation of custom bricks is also explained. Additionally, the approach taken to obfuscate the data with IBM's diffprivlib was applied.

5.2 Future Work

Since StatHunt was built with expandability in mind, there are a few recommendations that can be made considering future development.

As one of the first recommendations made by Hu [2019], the ability to fabricate a dataset would be a valuable addition. Doing so would allow researchers that have not yet collected their experiment data, but may already have some assumptions, outline their information in a clearly structured way.

Additionally, though outlined and designed, the support for question formulation and tag generation has not been implemented yet and would also prove useful. By aiding the user in the formulation of their questions, respondents could better provide help. Through the generation and addition of relevant tags, respondents would have an easier time finding questions, while researchers with similar problems could easily find the answered question for reference.

Bibliography

Monya Baker and Dan Penny. Is there a reproducibility crisis?, may 2016. ISSN 14764687.

Paul Cairns. HCI... Not As It Should Be: Inferential Statistics in HCI Research. Technical report, 2007.

Pierre Dragicevic. Fair Statistical Communication in HCI. In *Modern Statistical Methods for HCI*, pages 291–330. Springer, Cham, 2016. doi: 10.1007/978-3-319-26633-6_13.

Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–487, 2013. ISSN 15513068. doi: 10.1561/0400000042.

Alexander Eiselmayer, Chat Wacharamanotham, Michel Beaudouin-Lafon, and Wendy E. Mackay. Touchstone2: An Interactive Environment for Exploring Trade-offs in HCI Experiment Design. In *Conference on Human Factors in Computing Systems - Proceedings*, pages 1–11, New York, New York, USA, may 2019. ACM Press. ISBN 9781450359702. doi: 10.1145/3290605.3300447. URL http://dl.acm.org/citation.cfm?doid=3290605.3300447.

Ethan Fast, Binbin Chen, Julia Mendelsohn, Jonathan Bassen, and Michael S. Bernstein. Iris: A conversational agent for complex tasks. In *Conference on Human Factors in Computing Systems - Proceedings*, volume 2018-April, pages 1–12, New York, New York, USA, apr 2018. Association for Computing Machinery. ISBN 9781450356206. doi: 10.1145/3173574.3174047. URL http://dl.acm.org/citation.cfm?doid=3173574.3174047.

Asbjørn Følstad and Petter Bae Brandtzaeg. Chatbots and the New World of HCI. *Interactions*, 24(4):38–42, jul 2017. ISSN 15583449. doi: 10.1145/3085558.

Wayne D. Gray and Marilyn C. Salzman. Damaged Merchandise? A Review of Experiments That Compare Usability Evaluation Methods, 1998. ISSN 07370024.

Yue Hu. *Statistics in the Wild: How Practitioners Choose Statistical Procedures*. PhD thesis, 2019.

Maurits Kaptein and Judy Robertson. Rethinking statistical analysis methods for CHI. In *Conference on Human Factors in Computing Systems - Proceedings*, pages 1105–1113, 2012. ISBN 9781450310154. doi: 10.1145/2207676.2208557.

Wendy E. MacKay, Caroline Appert, Michel Beaudouin-Lafon, Olivier Chapuis, Yangzhou Du, Jean Daniel Fekete, and Yves Guiard. Touchstone: Exploratory design of experiments. In *Conference on Human Factors in Computing Systems - Proceedings*, pages 1425–1434, 2007. ISBN 1595935932. doi: 10.1145/1240624.1240840. URL `http://psychwextor.unizh.ch/wextor/`.

Xiaojun Meng, Pin Sym Foong, Simon Perrault, and Shengdong Zhao. NexP: A beginner friendly toolkit for designing and conducting controlled experiments. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10515 LNCS, pages 132–141. Springer Verlag, 2017. ISBN 9783319676869. doi: 10.1007/978-3-319-67687-6_10.

Bhavika R Ranoliya, Nidhi Raghuwanshi, and Sanjay Singh. Chatbot for university related FAQs. In *2017 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2017*, volume 2017-Janua, pages 1525–1530, 2017. ISBN 9781509063673. doi: 10.1109/ICACCI.2017.8126057.

Krishna Subramanian and Jan Borchers. Statplayground: Exploring statistics through visualizations. In *Conference on Human Factors in Computing Systems - Proceedings*, volume Part F1276, pages 401–404. Association for Computing Machinery, may 2017. ISBN 9781450346566. doi: 10.1145/3027063.3052970.

Chat Wacharamanotham, Krishna Subramanian, Sarah Theres Völkel, and Jan Borchers. Statsplorer: Guiding novices in statistical analysis. In *Conference on Human Factors in Computing Systems - Proceedings*, volume 2015-April, pages 2693–2702, 2015. ISBN 9781450331456. doi: 10.1145/2702123.2702347. URL `http://dx.doi.org/10.1145/2702123.2702347`.

Index

Typeset June 12, 2020