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

RWTH AACHEN UNIVERSITY

Visual and Functional Aids to Support the Statistical Analysis Workflow

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

by
Ilya Zubarev

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Axel Mayer

Registration date: 05.09.2018
Submission date: 05.03.2019

Eidesstattliche Versicherung

______________________________	______________________________
Name, Vorname	Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

____

____

____

selbständig und ohne unzulässige fremde Hilfe 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.

______________________________	______________________________
Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 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.

§ 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.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

______________________________	______________________________
Ort, Datum	Unterschrift

Contents

List of Figures

Listings

Abstract

Organisational, psychological and practical aspects of software development process have been extensively analyzed over the several last decades, and have in the recent years received a lot of public spotlight, with the technological start-up boom and increased public interest in the internal workings of the larger companies, brought in by their outstanding financial growth.

It is more so remarkable, than, that the day-to-day of the ”data science”, which often gets credited for facilitating this state of affairs, has long remained comparatively very scarcely explored.

Following the wave of newfound interest in the practices of *exploratory programming* and suggestions on classification of its needs and issues, we have created a *Hypothesis Manager* — a proof of concept extension for RStudio, one of the popular interactive developer environments of the field.

With it we aim to improve the user's ability to asses and navigate the context of their ongoing work, decreasing the time required to resume the active programming process after the attention shift, and promoting their ability to reuse their old code without having to resort to excessive duplication.

Acknowledgements

First of all I would like to thank Krishna Subramanian for his guidance and patience with me during the work on this thesis, and Prof. Dr. Axel Mayer, for his involvement was instrumental to my understanding of the practical dimension of the problems I was trying to solve, and has on multiple occasions allowed me to change the perspective to the benefit of the project.

Secondly, I would like to thank my colleagues at m3connect GmbH for not firing me over the erratic attendance and decreased productivity during the later stages of work on the thesis.

Thirdly, I would like to thank Ray Becker for providing me with the necessary weekly dose of social interaction, as well as a valuable second opinion on the practical aspects of the topic.

Lastly, I would like to dedicate this work, however underwhelming it might have ended up being, to my late grandmother.

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Source code and implementation symbols are written in typewriter-style text.

```
myClass
```

The whole thesis is written in Canadian English.

Chapter 1

Introduction

Exploration of new possibilities and *exploitation* of old certainties are the two basic strategies of adaptation exhibited by complex systems, from the genetic scale of organic life, to communal organizations and artificial decision-making agents [March, 1991]. Exploratory behaviour is characterized by terms like ″variation″, ″experimentation″, ″risk taking″ and ″discovery″; exploitative — by ″refinement″, ″efficiency″, ″choice″ and ″execution″.

Examining the field of information technology through the lens of these strategies, we can categorize software engineeringas being ″exploitative″, with its problem scope being primarily focused on implementation, and innovation coming from refinement of known and tried technologies, techniques and concepts, rather than from active discovery.

Software Engineering here is taken in the generalized sense, as systematic software development procedure, performed in accordance with the provided software specification

Conversely, the *exploratory programming* can be defined as a workflow with the following two properties [Kery and Myers, 2017]:

1. The programmer writes code as a medium to prototype or experiment with different ideas.
2. The programmer's goal is open-ended, and evolves through the process of programming.

It has first been described by Shiel [1983], as Xerox PARC management attempted to conceptualize the actuality of the AI research that was being conducted at the facility: it became obvious that contemporary software engineering practices were failing to apply to the inherently exploratory workflow. To the day exploratory programming remains characteristic for a broad variety of the data science fields [Kery and Myers, 2017], from computational statistics to computer vision to scientific visualization.

In **Chapter 2** we explore this diversity and its characteristic issues, based on the prior and ongoing research.

However, while these issues can be considered universal, addressing them in a universal context does not seem feasible, due to a variety of specific usage scenarios and tools that are being employed in every particular field of data science.

We have chosen *significance testing* as a jumping-off point for our search for solutions to these issues, as it exhibits several lucrative traits, like broad cross-disciplinary importance, relative platform-independence, and low set-up and exploration costs. In comparison, it would be impossible to address computer vision while ignoring the proprietary and fairly restricted platform of MATLAB, and any computation-intensive fields would have required organization of the relevant infrastructure, at least in its mock-up version, while we can satisfy most of our needs with a Read-Eval-Print Loop (REPL, also commonly referred to as ″[language name] console″) and a small collection of language-specific packages and, in rare cases, system libraries.

Chapter 3 showcases our attempt at creating a mental and data model for representation of the source code in significance testing.

Chapter 4 presents the inner workings our proof-of-concept solution — Hypothesis Manager, an add-in for RStudio IDE for R language.

R language is an open-source multi-paradigm programming language with a C-similar syntax and a strong functional

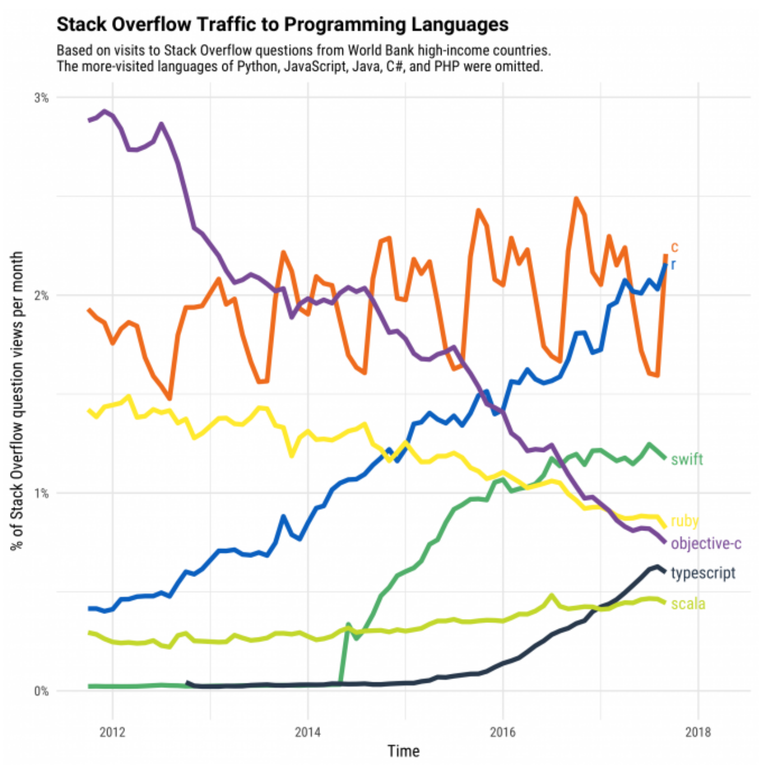
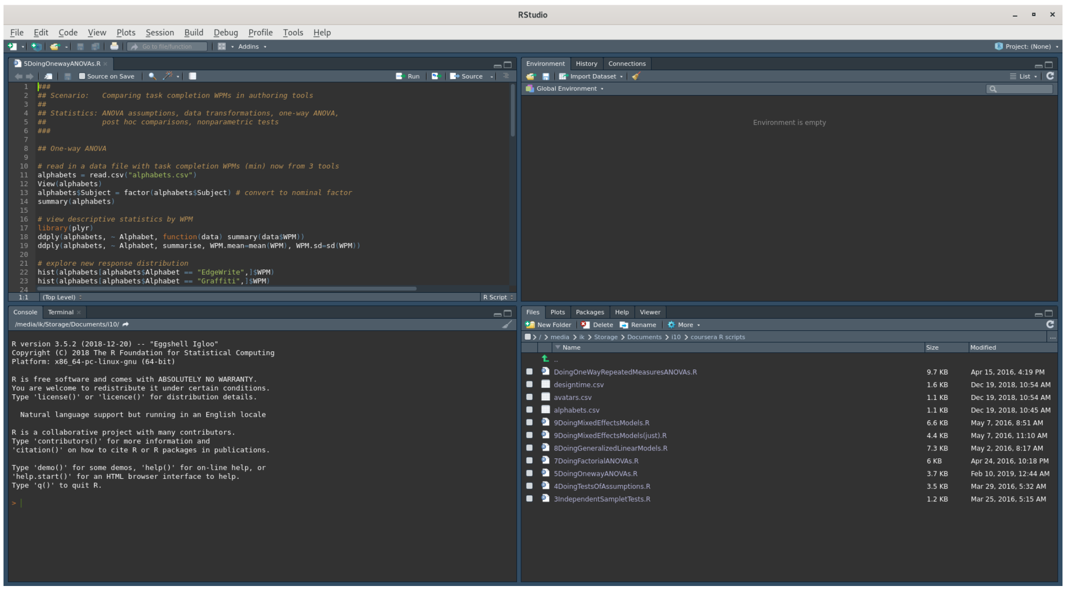


Figure 1.1: R popularity comparative to other languages [Robinson, 2017]

programming affinity [R core team, 2018]. Like its predecessor S, R is purposely designed for statistical analysis, providing for example a built-in support for specialized data storage types like matrices and datasets, complete with element-wise operations; as well as standard library packages for statistical modelling and graphics — something that in Python would only be available with external (albeit popular) packages like `numpy` and `pandas`. The language is a popular (see Figure 1.1) and its popularity is steadily growing [Robinson, 2017], possibly due to proliferation of ”big data” across the industries.

RStudio is the most popular (according to the AlternativeTo.net aggregation) Interactive Development Environment for R language, incorporating a script editor with optional interactive notebook functionality, a debugger, a REPL with session overview, an integrated graphics and R



```
###
## Scenario:   Comparing task completion WPMs in authoring tools
##
## Statistics: ANOVA assumptions, data transformations, one-way ANOVA,
##             post hoc comparisons, nonparametric tests
###

## One-way ANOVA

# read in a data file with task completion WPMs (min) now from 3 tools
alphabets = read.csv("alphabets.csv")
View(alphabets)
alphabets$Subject = factor(alphabets$Subject) # convert to nominal factor
summary(alphabets)

# view descriptive statistics by WPM
library(plyr)
ddply(alphabets, ~ Alphabet, function(data) summary(data$WPM))
ddply(alphabets, ~ Alphabet, summarise, WPM.mean=mean(WPM), WPM.sd=sd(WPM))

# explore new response distribution
hist(alphabets[alphabets$Alphabet == "EdgeWrite",]$WPM)
hist(alphabets[alphabets$Alphabet == "Graffiti",]$WPM)
```

```
R version 3.5.2 (2018-12-20) -- "Eggshell Igloo"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Name	Size	Modified
..		
DoingOneWayRepeatedMeasuresANOVAs.R	9.7 KB	Apr 15, 2016, 4:19 PM
designtime.csv	1.6 KB	Dec 19, 2018, 10:54 AM
avatars.csv	1.1 KB	Dec 19, 2018, 10:54 AM
alphabets.csv	1.1 KB	Dec 19, 2018, 10:45 AM
9DoingMixedEffectsModels.R	6.6 KB	May 7, 2016, 8:51 AM
9DoingMixedEffectsModels(just).R	4.4 KB	May 7, 2016, 11:10 AM
8DoingGeneralizedLinearModels.R	7.3 KB	May 2, 2016, 8:17 AM
7DoingFactorialANOVAs.R	6 KB	Apr 24, 2016, 10:18 PM
5DoingOnewayANOVAs.R	3.7 KB	Feb 10, 2019, 12:44 AM
4DoingTestsOfAssumptions.R	3.5 KB	Mar 29, 2016, 5:32 AM
3IndependentSampletTests.R	1.2 KB	Mar 25, 2016, 5:15 AM

Figure 1.2: RStudio overview screenshot

help viewers, and a version control system interface.

RStudio is also extendable with plug-ins (referred to as ”add-ins”) with rich visualization capabilities and provides an API with direct access to the IDE’s working context, both of which are instrumental to our project.

Particularities of R language and RStudio design and their influence on the development process are also examined in Chapter 4.

Chapter 5 explains how Hypothesis Manager addresses the issues of exploratory programming, and presents the design for user evaluation, that we, unfortunately, were not able to perform due to failure to recruit participants.

Chapter 6 summarizes this work and details some avenues for possible future improvement of the Hypothesis Manager, that we saw in the process of the development, but conceded in preference to more immediately important ones.

Chapter 2

Related work

Since its definition by Shiel [1983], exploratory programming has maintained a stable interest of the scientific community: Google Scholar lists approximately 1850 results for the keyword, distributed throughout the decades. It is important to note, however, that a number of those results, particularly from the earlier years, are mainly dedicated to Smalltalk, as it was a manifestation of Xerox PARC internal work on the concept, with exploratory programming in general being referenced, but only so.

Kery and Myers [2017] readdress exploratory programming on the conceptual level, contemplating "[the] lack of tool support for experimentation, including a lack of support for recording and sensemaking of exploration history, and a lack of support for exploration by groups of people". While Kery et al. [2017] focus on the exploration history aspect with Variolite, Patterson et al. [2017] independently suggest a schematic for language representation in collaborative data science environment,

2.1 Issues of Exploratory Programming

Through interviews, observation and collected code samples Subramanian et al. [2019] have identified several specific issues that arise for data scientists in the process of exploratory programming.

2.1.1 Code hoarding and duplication

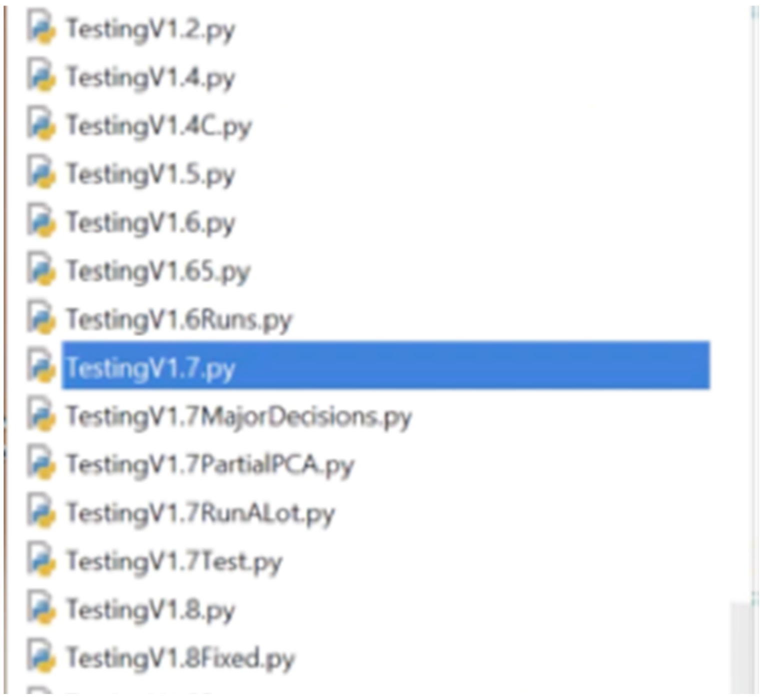
Exploratory processes naturally involve juxtaposition and rejection of alternative prospects. While some of these prospects remain irrelevant for the rest of the exploration, it is equally as natural for them to resurface due to increased credibility after their alternative has been rejected further during the analysis.

In application to exploratory programming, it means that programmers build their workflow around immediate access to multiple prior stages of the development of their ongoing project.

Version Control Systems (VCS) are a commonplace solution for similar issues in the "exploitative" software development, however as Kery et al. [2017] have found, even data scientists that have the relevant experience and actively use VCS in other projects, tend not to abstain form them in the exploratory scenarios. Their reasoning on this gets somewhat contradictory — interviewees state that they do not need backtrack, while clearly exhibiting backtracking in their code — and some of the argumentation against VCSs' usage exposes an ill-informed perception of these tools in general, putting emphasis on the collaboration or pointing out the quality of life complications that are resolvable with a dedicated VCS GUI or an extension for their text editor or IDE of choice.

The fact remains, however, that exploratory programming necessitates preservation of the older versions of the code, and, in absence of the Version Control Systems, it ends up getting preserved as is. Depending on the application, language and environment it may be a different script file (see figure 2.1) or a side-by-side copied code chunk, which is either commented out or informally labeled as obsolete.

The code base that Subramanian et al. [2019] have collected



TestingV1.2.py
TestingV1.4.py
TestingV1.4C.py
TestingV1.5.py
TestingV1.6.py
TestingV1.65.py
TestingV1.6Runs.py
TestingV1.7.py
TestingV1.7MajorDecisions.py
TestingV1.7PartialPCA.py
TestingV1.7RunALot.py
TestingV1.7Test.py
TestingV1.8.py
TestingV1.8Fixed.py

Figure 2.1: Infromal versioning example [Kery et al., 2017]

has examples of R language script files of multiple hundred lines, that up to 75% consist of the repetition of the same 3 to 40-line code chunks with a single function argument being different between any two duplicates.

2.1.2 Intermodal and intercontextual code transfer

Informal version archive is not the only way code duplication occurs in exploratory projects. As it occurs in the software development at large, some of the code gets to carry over from project to project as a personal toolbox of the particular author.

And, perhaps uniquely to the realm of data science, executable code is often not the only form of code that exists within a particular research. Exploration may take place entirely in the REPL environment, while being stored in the script file for posterity , or samples may also get to be

Sometimes together with the unfiltered and unedited print-outs, which turn the script file practically invalid included into the reports.

This results in code being independently iterated in the several locations at a risk of becoming divergent.

Interactive language notebooks do alleviate this problem to a degree, providing the universal modality for storage, experimentation and (to a certain degree) reporting, but their availability and functionality depends on the exact programming language and flavour of data science: for some applications interactive notebooks can only serve explanation or reporting purpose, since code execution would only available in the form of script files.

2.1.3 Context retention

Extensive size of the code hoards, coupled with the necessity of regular context switches makes it difficult for data workers to retain a mental model of the working code file and turns reconstruction of this mental model into an additional chore, whether or not informal versioning is employed within the file.

As data workers also admit to be actively reusing the code from their past projects, this issue can surface even if the ongoing work is maintained in a fashion that minimizes the hoarding or context switching.

DeLine et al. [2006] address a problem of an unfamiliar code navigation as a more generalized scenario, and on the basis which is not immediately related to the exploratory programming per se. However, their solution of this problem, and its later practical implementations like ”minimap” of *Sublime Text* editor[1] served an important inspiration to our work.

[1]We were not able to prove any direct connection between DeLine et al. [2006] and development of Sublime Text, or its authors, Will Bond and Jon Skinner. The likeness of their approach to the design of the navigational feature is, however, remarkable.

Chapter 3

Data model

The specifics of why our add-in required an intermediate data representation will be explored in the next chapter, together with the rest of the add-in's inner workings, however it is important to state that they have, and this constraint gave us an opportunity to develop a model that would be used throughout the rest of this report as a way of thinking about the entities that we are operating with.

While this model is inherently an answer to the specific obstacle we had faced, and therefore is tailored to the R language and significance testing workflow to a certain degree, we propose that already in its current stage it can be used in the broader context.

There are also certain similarities with the collaboration-centered model proposed by Patterson et al. [2017], but they are superficial, as our approach demanded bigger variety in types of entities and a greater granularity of expression representation.

As a final disclaimer we should add that the labels used for different types of entities are a product of early development stages, and their common meaning does not necessarily represent exactly what they have ended up being attached to later on. These situations will be highlighted to minimize the ambiguity.

3.1 Functions and Variables

Or, in a more proper terminology, *expressions* and *objects*. While "variables" and "objects" can be used interchangeably with difference between the terms being relevant only in the context of the inner logic of R language, "expressions" cover a broad range of language statements with "functions" being just a single kind. For the purposes of our model, however, the more exotic scenarios (examples and more detail are explored in section 4.2) are ignored, so that in practice expressions end up covering only function calls and operator usage.

> In every computer language variables provide a means of accessing the data stored in memory. R does not provide direct access to the computer's memory but rather provides a number of specialized data structures we will refer to as objects. These objects are referred to through symbols or variables. — R core team [2018]

> When a user types a command at the prompt (or when an expression is read from a file) the first thing that happens to it is that the command is transformed by the parser into an internal representation. The evaluator executes parsed R expressions and returns the value of the expression. All expressions have a value. This is the core of the language. — R core team [2018]

We assume the function object to have the following properties:

- unique identifier
- name
- location in the script file
- full signature, if available

- function's (or its namesakes') origin package, if available
- list of arguments
- depth within the call tree
- breakpoint identifiers, if relevant

Listing 3.1: Serialized function instance

```
{
    ''id'': ''f-d208439a-0b45-40d9-a43a-dfb4f0dbb721'',
    ''name'': ''aov'',
    ''lines'': [31, 31],
    ''signature'': ''m = aov(WPM ~ Alphabet, data=alphabets) # fit model'',
    ''packages'': [ ''package:stats'' ],
    ''arguments'': [
        ''h-a5fd4a79-b442-4bcc-b07c-9f0760c2abc6'',
        ''v-a5f30b8b-c0c8-4d4b-a543-05f4738fe4ce''
    ],
    ''depth'': 1
}
```

Argument list can contain constant values as well as other entities' identifiers, including those of the nested expressions.

Nested expressions precede their "parent" in the function object collection, and can be identified by the depth parameter of a greater value.

Breakpoint is assigned if an existing variable is known to have been partially modified as a result of an expression. Complete overwrites of the same variable name do not generate a breakpoint, as these objects are considered distinct.

Variable object have the following properties:

- unique identifier
- name
- precursor variables' identifiers list, if applicable

- column identifiers list, if applicable
- origin function identifier
- internal type definition, one of:
  - data — for datasets
  - model — for variables that can be explicitly identified as (statistical) models
  - column — for column names, as R language allows them to be referenced independently from the dataset as a whole
  - formula — rare case when R formula is assigned to the variable before being used in a modeling function
  - constant — for values that fit neither of the other types
- value, is auxiliary and can be nullified if its data type does not support serialization
- generation

Listing 3.2: Serialized variable instance

```
{
    ''id'': ''v-6846c0cc-6103-42dd-8677-a193e631353e'',
    ''name'': ''m'',
    ''precursors'': [
        ''v-a5f30b8b-c0c8-4d4b-a543-05f4738fe4ce''
    ],
    ''columns'': [],
    ''origin'': ''f-b7f11c3b-01a0-46d0-8ca4-6bb385175897
    ''type'': ''model'',
    ''value'': null,
    ''generation'': 1
}
```

Origin function is one that had its output assigned to the variable's name. Precursor variables are ones that are used as arguments in that function. I.e. statistical model would have its dataset marked as a precursor, dataset that is read from the file would have no precursors if the file name has not been retrieved from the separate (”constant”-type) variable, otherwise this variable would be marked as a precursor for the dataset.

Variable’s generation is incremented upon the highest generation value among its precursors. We have considered fixating the datasets’ generation to a single base value, but in practice that would have meant applying cascading changes to its precursors and those precursors’ other ”children”, which might have lead to inconsistencies, as several datasets may be derived from the same arrangement of ”constant” lists with, for example, randomization resulting in additional intermediate variable for one of them. Therefore, in absence of a practical need for such behaviour, the idea got scrapped.

Multiple variables can share a name, as it is common in the duplicated code, where informal index suffixes are applied independently to several models within each step of the exploration (see listing 3.3).

For the purposes of the model these variables are distinguished by their unique identifiers. It is also taken as granted, that once the name has been reassigned, the old variable can not be referenced anymore.

Listing 3.3: Variable reassignment example

```
# -> PC2 ------------------------------------------------
# first up-down component
m0 <-lmer(PC1 ~ mat + cluster + mat*cluster +
                (1|pop) + (1|pla) + (1|position),
          data=up.2)
m1 <-lmer(PC1 ~ mat + cluster +
                (1|pop) + (1|pla) + (1|position),
          data=up.2)
m2 <-lmer(PC1 ~       cluster +
                (1|pop) + (1|pla) + (1|position),
          data=up.2)
m3 <-lmer(PC1 ~ (1|pop) + (1|pla) + (1|position),
          data=up.2)

anova(m0,m1) # INTERACTION
anova(m0,m2) # MATING SYSTEM
anova(m0,m3) # GENETIC CLUSTER
summary(m0)

# -> PC3 ------------------------------------------------
# first left-right component
m0 <-lmer(PC1 ~ mat + cluster + mat*cluster +
                (1|pop) + (1|pla) + (1|position),
          data=left.2)
m1 <-lmer(PC1 ~ mat + cluster +
                (1|pop) + (1|pla) + (1|position),
          data=left.2)
m2 <-lmer(PC1 ~       cluster +
                (1|pop) + (1|pla) + (1|position),
          data=left.2)
m3 <-lmer(PC1 ~ (1|pop) + (1|pla) + (1|position),
          data=left.2)

anova(m0,m1) # INTERACTION
anova(m0,m2) # MATING SYSTEM
anova(m0,m3) # GENETIC CLUSTER
summary(m0)
```

3.2 Hypotheses

Semantic information that is enclosed in the function and variable entities is limited to the variable type and, from a certain point of view, the function's origin library, as it can allow to suggest its purpose (keeping in mind that dedicated packages may include popular utility functions).

We describe the majority of the semantics in the additional entity type, called "hypothesis". Details on the emergence of this concept are included into the section 4.4.1.

Unlike the base two entity types, hypotheses are heavily predicated on the context if significance testing. It may be possible to extrapolate them into more generic "explorations", but that would likely require introduction of several alternating schematics.

Hypothesis entity consists of:

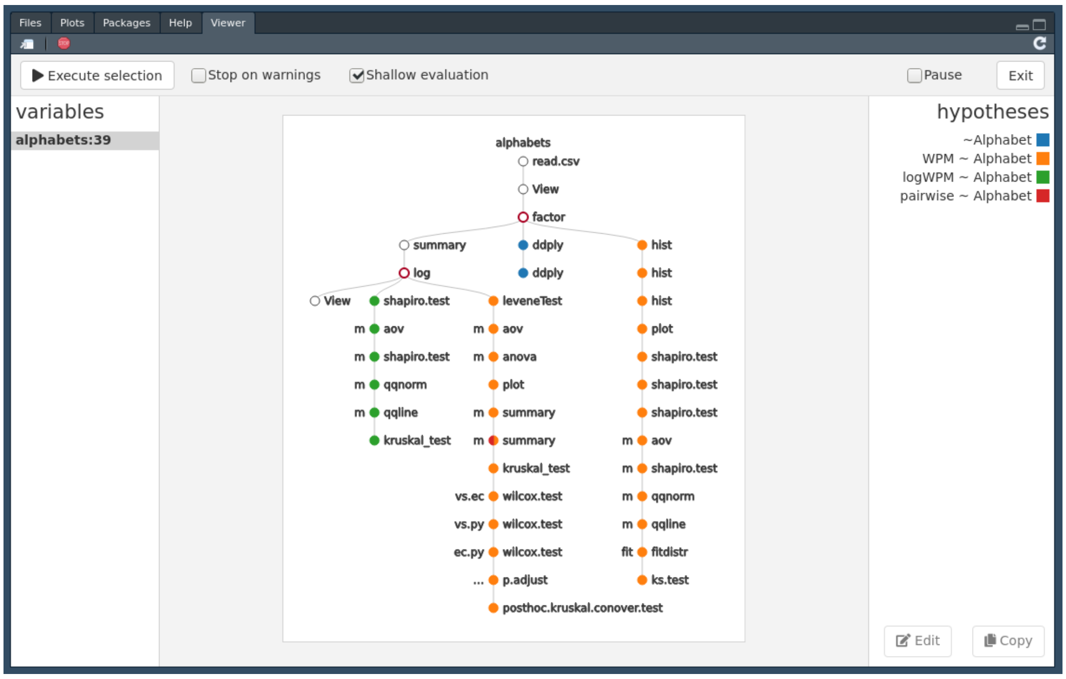
- unique identifier
- name
- arrangement of references to the "column"-type variables, distinguishing a dependent and a list of independent (or "control") dataset columns
- list of functions' identifiers for the functions that were categorized as exploring this hypothesis
- list of "model"-type variables' identifiers that are created from these functions, if any
- list of "formula"-type variables' identifiers, relevant to this hypothesis, if any

Listing 3.4: Serialized hypothesis instance

```
{
    ''id'': ''h-d665d82c-3d46-4d47-9497-5ef018eb310c'',
    ''name'': ''logWPM ~ Alphabet'',
    ''columns'': {
        ''dependant'': ''v-5fed8f3f-0711-44b9-913b-0374a14d1d7b'',
            ''control'': [
                ''v-5ec29826-ef56-4dce-9f24-7cdf145c25bc''
            ]
        },
        ''functions'': [
            ''f-0614e121-b80f-4c46-81d8-9abd555664ee'',
            ''f-30f527c3-b43f-480c-8da8-3ffd9c4a516e'',
            ''f-55a48aa0-84c3-41ac-b236-bc4f58b07d64''
        ],
        ''models'': [
            ''v-72a92c36-e876-42a7-adc2-fa63a21374bf''
        ],
        ''formulas'':[]
}
```

We have consciously decided not to distinguish the exact interrelation between the control columns, which may differ, as formula syntax of R language includes multiple of R base operators and allows for limited constant usage.

Hypothesis exploration is not limited to creation and subsequent use of model variables. If dataset columns are plotted against each other, or control columns are used to create a subset to examine the dependent column's values, these functions are categorized as exploring the hypothesis as well.



☐ Stop on warnings ☑ Shallow evaluation ☐ Pause

Figure 4.1: Hypothesis Manager

Chapter 4

Hypothesis Manager

Although may not be apparent from the first sight, ”RStudio IDE” itself is a Webkit-based client for the ”RStudio-server” application (available as a stand-alone commercial product), that manages the execution of R code, session storage and other utility aspects of working with the R language. Its add-ins, therefore, inherently have to have *some* server code, even if no client-server interaction is going to be present.

Client-server interactions can be managed two-fold: `rstudioapi` package can be used to mimic or trigger IDE's own functionality, from setting of the text cursor position and saving the changes in the open files to selecting the visual theme of the IDE. Developers, however, are not allowed further "under the hood" of the application, which makes it impossible to interact with the text editor input events or extend the basic visuals of the IDE. Limitations of this happenstance and their influence on the design of the add-in will be elaborated upon in the designated section.

Since RStudio overhauls the base R language approach to representing graphics and help pages, these can also be considered interactive to a degree, however the details on internals of this overhaul are not documented and due to a very limited scope of use, it can hardly be considered extendable.

Part of the visuals which developer is, on the other hand, in the significantly more independent control of is the "Viewer" pane of the RStudio, intended for the display of "Shiny applications".

4.1 Shiny

Shiny, distributed as an eponymous R package, is another part of a broader RStudio project, providing a web framework for R, complete with a server, an assortment of tools — page layouts, sophisticated inputs (for instance, replacing the basic select and multiselect inputs with the selectize.js counterparts[1]), visualization built-ins for geospatial or graph data — and a front-end JavaScript library to facilitate the client-server interactions.

[1]See `https://selectize.github.io/selectize.js/` for the detailed list of selectize.js features

It aims to enable R programmers to build web applications "without requiring HTML, CSS, or JavaScript knowledge" [RSt]. These applications can be executed in the IDE on a one-off basis, included into the *R Markdown* interactive notebook cells, or deployed on a dedicated hosting, provided by the RStudio team, directly from the IDE.

If not extended further with the help of packages like `htmlwidgets`, Shiny application is a textbook example of a "thin client application". Extension would allow developer to include custom design elements and front-end code and business-logic, that can still receive the data via the default Shiny event system.

The server part of the Hypothesis Manager is an example of such extended Shiny server, with the additional custom output used along with a number of the default "thin" Shiny client inputs and outputs. It also handles the utilization of `rstudioapi` for manipulations with a script editor pane of the IDE, and parsing of the source code.

4.2 Parser

We distinguish *syntactic* and *semantic* stages in the process of parsing the code parsing process.

Ultimately, syntactic parsing is handled by R language built-in `parse()` function that turns valid plain-text into the *expression*-type object, but due to reasons that would later be elaborated upon, we could not settle with putting the entire raw script file through that mechanism at once, introducing some original logic and code into that process.

Expression object may contain nested expressions as well as 2 other types of values:

1. symbol - character strings, representing function or variable names;
2. atomic - unnamed constants of atomic types: booleans, numbers, strings and null-values;

It is important to stress that, as Listing 4.1 illustrates, there is no distinction between function calls, operators (including assignment), control structures and block statements within the control structures.

Listing 4.1: `pryr::call_tree` evaluation

```
> call_tree(quote(
+     for (i in 7:ncol(dataParents)) {
+         dataParents[, i] = as.numeric(dataParents[, i])
+     }
+ ))
\- ()                    # expression
  \- 'for                # symbol - name of expression
  \- 'i                  # symbol - regular variable
  \- ()
    \- ':
    \-  7                # atomic
    \- ()
      \- 'ncol
      \- 'dataParents
  \- ()
    \- '{
    \- ()
      \- '=
      \- ()
        \- '[
        \- 'dataParents
        \- 'MISSING      # symbol - missing parameter
        \- 'i
      \- ()
        \- 'as.numeric
        \- ()
          \- '[
          \- 'dataParents
          \- 'MISSING
          \- 'i
```

Semantic parsing assumes expression objects as input and matches them with one of the designed case scenarios based on its contents, here partially grouped for better transparency of reasoning:

- assignment operators: `<-` and `=`
- Syntax that can indicate hypothesis presence
  - formulas: `~`
  - index-based multi-item retrieval: `[`
  - name-based item retrieval: `$`
- Syntax that is obstructs function detection
  - parentheses: `(`
  - control structures: `if`, `for`, `while`, `repeat`
  - block statements: {
- expressions
- symbols
- atomic values

4.2.1 Adjustable scanning window

Expression objects, perhaps, obviously so, do not preserve indentation, comments or distinction between the expressions, separated by semicolon and a newline character, which makes it inconvenient to attempt to associate expressions with their location in the file, which is necessary for several aspects of visualization and interactivity of the add-in.

Which has lead to the aforementioned manual overhaul of the parts of the parsing process: file lines are being iterated through, `parse()` function is being called on each line individually, with two specific parsing exceptions being intercepted as they signify that expression should have had continued on the next string. In these cases, the second margin of the window starts to expand until either the exception-less pass gets performed, or a different kind of exception interrupts the process altogether. The parsing will then resume from the next line after the successfully productive window.

In this fashion, besides having a way to map the function with its position in the file, it also becomes possible to locate the problematic lines of code, if such happen to occur in the file, track the progress of the parsing process, which becomes useful for script hoards of multiple hundreds, or thousands, lines in length, and to preserve the function signature in its original form, later used to make the particular instance of the function more recognizable among the others.

4.3 Limitations

As it may be evident from the glance inside the R language internals we have provided, Perl paradigm "There's more than one way to do it" applies to R in full extent.

This is further articulated by that R packages may introduce custom operators, that may introduce complicated behaviour to the visually basic code, for example:

- `magrittr` introduces a pipe operator `%>%` enabling user to chain the data processing commands similarly to Java streams or JavaScript `Array.prototype` functions

  ```
  car_data <-
    mtcars %>%
    subset(hp > 100) %>%
    aggregate(. ~ cyl, data = .,
              FUN = . %>% mean %>% round(2)) %>%
    transform(kpl = mpg %>% multiply_by(0.4251))
  ```

  In base R, this operations would have to be written as a nested arrangement of function calls:

  ```
  car_data <- transform(
    aggregate(
      . ~ cyl,
      data = subset(mtcars, hp > 100),
  ```

```
      FUN = function (x) round(mean(x, 2))
    ),
    kpl = mpg*0.4251
  )
```

- `zeallot` package introduces another operator `%<-%`, which allows for Python-like multiple assignment for vectors and list objects:

  ```
  c(duration, wait) %<-% head(faithful)
  ```

While existence of the latter package have been of a great relief during the work on the add-in, due to developer's personal preferences and habits of working with the Python language, attempting to process its usages as intended would have severely complicated the parsing process, possibly making the text-based implementation impossible, since making even broad assumptions on the nature of the returned values would require knowing positional output of the invoked function.

It is important to note that while `magrittr` pipe operator does not have these issues, processing it a general-purpose expression does not have any practical downsides either

Additionally, while this package can be found used internally by multiple popular R packages, like `keras` and `vctrs`, both being in the 16th percentile according to rdocumentation.org, it does not exhibit such popularity itself — 96th percentile with zero recent downloads.

Finally, extraneous operators may become ambiguous, as, for example, `%<-%` operator also exists in the `igraph` package, which is as, if not more, popular as `zeallot`: 99th percentile, but >3000 monthly downloads according to rdocumentation.org.

Under these considerations, for our parser we have decided to go off of the R code base that was available to us with one preliminary exception: no support for custom modeling languages, like those used in `lavaan` or `rjags`.

4.3.1 Parsing Loop

As it has been mentioned, while `rstudioapi` provides a lot of control on the working context of the RStudio, allowing, among other things, to switch the projects and modify the open files, it does not allow to intercept the input events that are happening in the Ace text editor on its front-end.

While ideally we would have liked to plug the parser trigger directly into the Shiny server event system, due to these limitations it had to be mimicked with a periodically firing event and an extra manual check for change of the script editor context (file name or file contents' hash being different from to previous successful iteration).

As a consequence, parsing is getting initiated practically with each keyboard stroke which is more often that it would have been desirable. With smaller files in can lead to barrages of "false negative" error messages, and for bigger files that can take minutes to parse these interruptions may be flow-breaking.

To mitigate this issue to a degree, we have introduced a manual pause switch, but it is evident that either additional logic for its automatic engagement, or a more sophisticated file difference identification (and, perhaps, partial parsing) mechanism is required for a more comfortable usage.

4.3.2 Shiny server blocking behaviour

While we were not able to find a documented explanation for this quirk of Shiny server, questions on the topic can be found across the project's GitHub[2] and in Stack Overflow communities[3]. At the moment of writing, it can only be asserted: Shiny server blocks the REPL interface of the RStudio and commands get stacked until the server is shut down.

[2]`https://github.com/rstudio/shiny/issues/652`

[3]`https://stackoverflow.com/questions/24020636/access-use-r-console-when-running-a-shiny-app`

We have investigated ways to bypass this issue with multithreading and multi-session execution, and found those unsuccessful, since session information proved to be vital for RStudio client-server contact retention.

However, while direct access to REPL is unlikely to be regained until relevant changes are introduced to RStudio or Shiny by their maintainers, we could still provide a proxy for execution of the script editor context chunks.

4.4 Visualization

Before there were RStudio and direct address of exploratory programming design concerns, this project was initially conceived with an intention to introduce a graphical programming tool for StatWire [Maas, 2017] R programming environment.

With a shift away from graphical programming and necessity to represent the code one-to-one, and towards a supplementary modality with visualization granularity and metaphor in general left to our arbitration, we were also enabled to operate with the concepts from the realms of semantic and methodical.

4.4.1 Metaphor

Particularly, we were now interested in how users' workflow was manifesting itself through the code.

At first, perhaps out of the habits of our day-to-day practices, we were expecting to use variables and their mutations and transitions as a main indicator of the workflow routines and milestones.

However, upon maturing the data model and add-in backend enough to reliably produce data for visualization, it became apparent that in common practice, data scientists rarely, if ever, actively use more than two generations of the

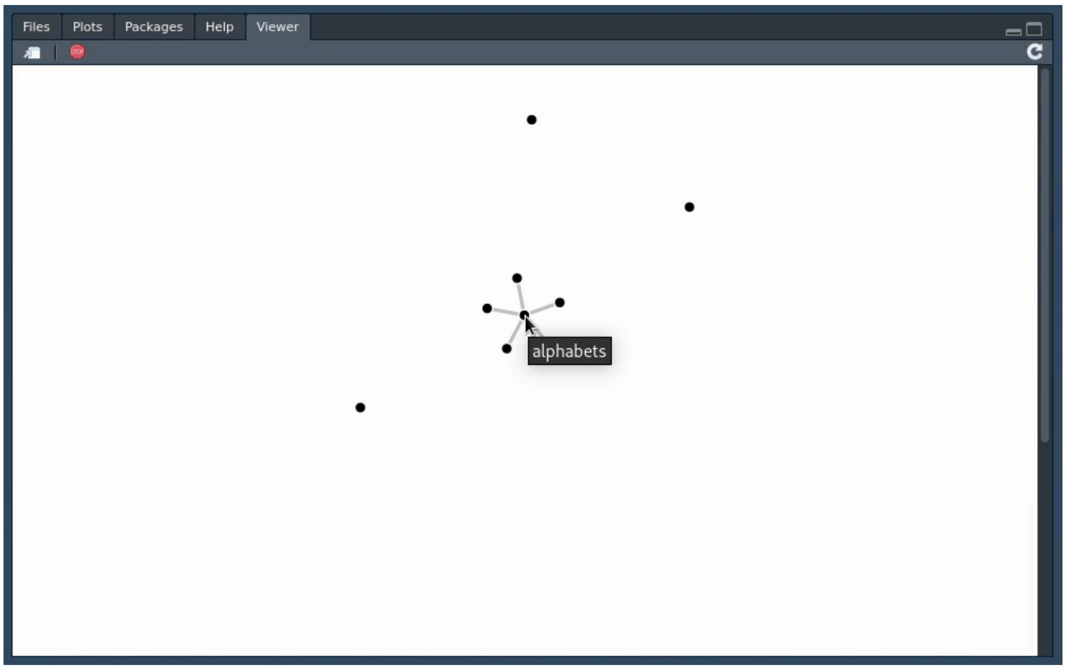


Figure 4.2: First attempt at data visualization

data variables, with model variables being derived either directly from the initial dataset, its subsets, or ”first generation” models, number of datasets being, perhaps obviously, mostly limited to one per script, and, in cases where there are more, almost never getting compared to one another.

Figure 4.2 depicts a basic force-directed graph that we have started off with, with variables as its nodes and their mutual relationships as arcs. Five nodes around the central one represent all the extra variables used within the same file that was later visualized in figure 4.1.

This has prompted us to iterate on the initial visual metaphor, giving a meaningful place to functions that are executed on the datasets and their statistical models, which, while being included to the data model from the very beginning, had yet to find an appropriate visual representation.

London tube map became an important inspiration for this iteration of the visual metaphor, although final version arguably much more resembles the Git branches visualization.

Additionally, this has helped to articulate the idea of hypothesis, visually — as a way to distinguish the similar function sets, and conceptually — as a unit of exploration that could (or so our thinking went) be condensed into a function and extracted as a reusable subroutine.

This duplication reduction functionality was at the initial point perceived as the main interactive feature of the add-in, and a "solution" to the problem of duplicate hoarding in general. However, multiple reassessments of its viability throughout the development have remained inconclusive. More so when new, more inconspicuous approaches (4.5.3, 4.5.4) of influencing user's workflow were introduced.

4.4.2 Implementation

To provide ourselves with necessary degree of freedom in visualization, we extend the base `shiny` capabilities with aforementioned `htmlwidgets` and add a d3.js library for front-end. D3 provides a wide range of tools for data visualization, complete with a custom Document Object Model navigation toolkit for simpler element manipulation and direct association between graphics and original data.

Specifically, we utilize the d3.js capabilities for hierarchy generation and graph construction. While it leaves an avenue for an overhaul in future (see 6.2.2), the source code functions are arranged in a tree, with dataset origin being a root node.

Available dataset variables and hypotheses are listed in the side panels of the graph, former being a selection menu to switch between the graphs for each specific dataset (if multiple are available) and latter — a legend for the color-coding of the graph notches, and an entry point for hypothesis editing functionality, which will be explored later (4.5.3) together with the rest of the interactive features.

A single function node consists of a circular notch, used as visual focal point and an anchor for the graph arches, and a function name.

The notches are colored according with the hypotheses they represent, with multiple colors forming a pie chart, and hy-

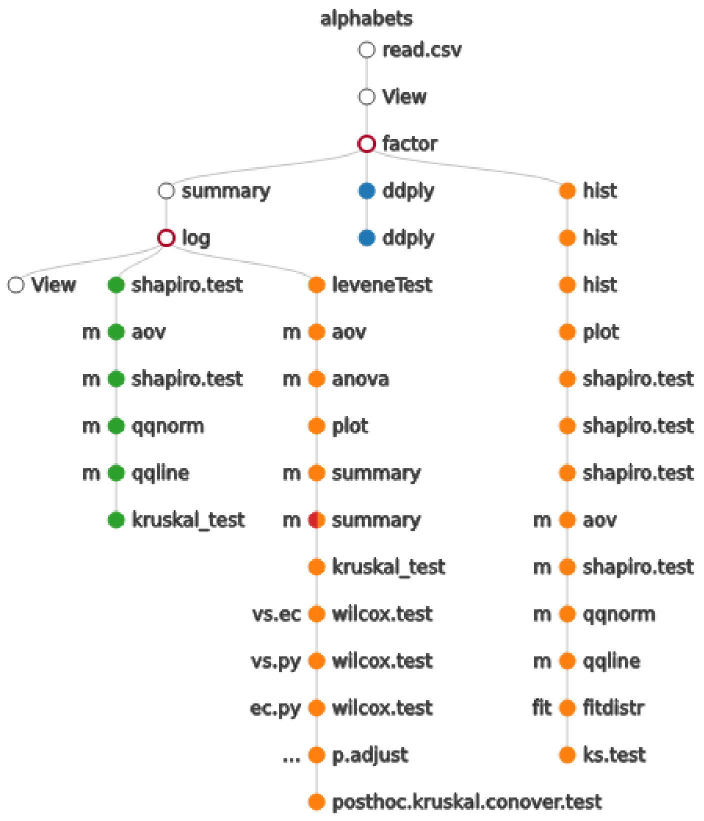


Figure 4.3: Single variable graph

pothesis-less notches having white background and a light-grey rim (see figure 4.4).

Node-hypothesis attribution builds upon the direct function-hypothesis link, captured in the data model, by also tracking the model variables and marking the functions where they were called as belonging to a hypothesis. In these cases, names of these model variables are also visualised on the opposite side from the function name. If multiple models are used in a single function, their names are replaced with an ellipsis.

Functions are then attached to each other in the order of their occurrence in the source code, with two notable deviations:

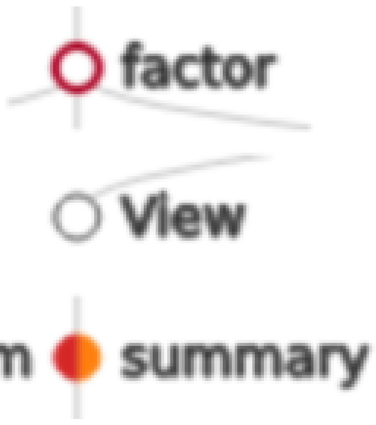


Figure 4.4: Close-up of different kinds of notches

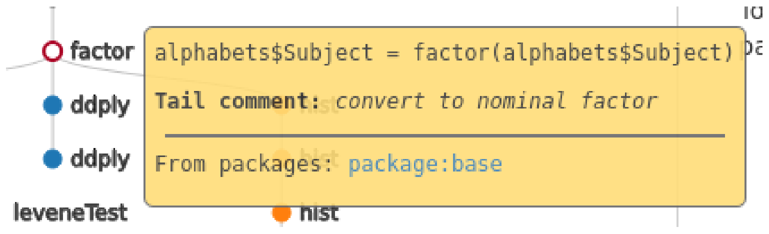
1. If the function was recognized to modify the dataset, it serves and a *breakpoint*, becoming an intermediate root for all the later functions to get attached to
2. Hypotheses spawn their own branches under the common root node, with functions that do not belong to any hypothesis being grouped together as if there was one

If the hypothesis is being explored both prior and past the breakpoint, it spawns branches separately with each root node, since we can not ensure that the exploration has not been disturbed by changes in the data.

4.5 Interactivity

Since we were aiming to improve the workflow at large, we could not limit ourselves with just visualizing the code. Coming back to the tube map metaphor, active code *navigation* was one of the immediate inspirations brought with introduction of this idea.

This is also the place where the introduction of adjustable scanning window (see 4.2.1) had a direct implication: knowing the lines that correspond to each function,



```
alphabets$Subject = factor(alphabets$Subject)
Tail comment: convert to nominal factor
From packages: package:base
```

Figure 4.5: Graph node tooltip

`rstudioapi` now allowed us to set the cursor to the position that corresponds to the the graph node. After some additional consideration it has also been decided that creating a cursor selection for the entire span of the expression would serve as a better indication, due to being more visible and also allowing user to execute the function immediately finding it in the code.

4.5.1 Tooltips

After the initial trials, it has also became apparent that even though direct association between the node and the expression in code allows to facilitate the mental connection between the two, function name on its own is a sub-par indicator of the exact instance of the expression (for heavily duplicated code can hold a number of almost identical function calls), and that the meaning behind the function can be obscure for somebody who has no prior knowledge of the exact package this function is derived from.

On the other hand, putting the full function signature onto the graph would heavily impede its readability, and partially defy the purpose behind the addition of the new modality into the environment.

The compromise was found in introduction of the interactive tooltips, that, while overlaying the graph, did not hinder its usefulness, but can provide an extensive amount information with the natural action of putting the cursor over the object of interest.

While these tooltips can be greatly extended in future if found necessary, at the point of writing, they contain:

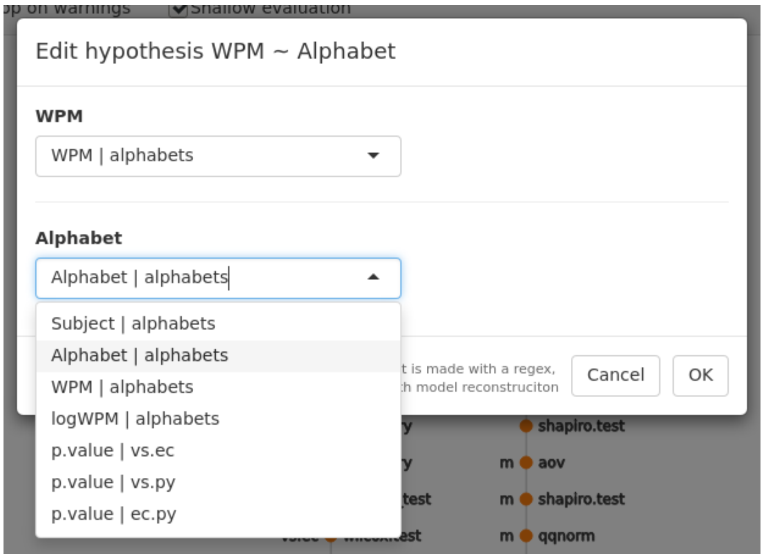
- Full function signature
- Optional separate citation of the end-of-the-line comment
- Links to the R help pages for the function (or its namesakes)

While, as it has been mentioned before, help pages can not be triggered through any of the RStudio APIs, we have found a way to reliably present them by sending the output of the default R `help` command (or `?` operator) to the print output.

4.5.2 Modes of operation

The notion of the text-only parsing has already been brought up elsewhere in the text. During the development process we have experienced somewhat of the distortion of perception: having the moderately vast code base, but lacking any context for what the contents of this code base were initially written for, we perceived this approach to the R code as commonplace, whereas the data scientist would have also had, and actively employed, the dataset file that was actually being explored in the script they are writing.

Besides enabling the code execution, availability of the dataset file can also improve the parsing process, as it alleviates the ambiguity of dataset column name references, which are not necessarily being made with a direct conjunction with the dataset variable reference and can, therefore, be confused with constants. Reading the dataset form the file, on the other hand, allows to generate all of the column variables from the very beginning of the processing and ensure that no column can spontaneously occur in the unforeseen context.



Edit hypothesis WPM ~ Alphabet

WPM: WPM | alphabets

Alphabet: Alphabet | alphabets

Figure 4.6: Hypothesis editor modal dialog

It is interesting to note that the name ”Run” in the context of RStudio is used to refer to the current line/selection execution, which is also bound to a widely used Ctrl/Cmd+Enter hotkey. At the same time full script execution — a commonplace default for the IDEs — doesn’t have a hotkey and is called ”Source”, which, while remaining meaningful, contradicts naming conventions.

From that, and from the fact that most of the R code in our possession is used to output data into the console or into the graphics viewer, we infer that automatic execution of the entirety of the script would be undesirable, and have limited it to the assignment expressions, which enhances the parser’s understanding of the script’s context and would allow for a less constrained operation with the variables further down the line, but would not produce any additional output.

4.5.3 Hypothesis editing

Current state of variable and hypotheses lists by the sides of the graph are a product of several iterations on the add-in’s

UI. Hypothesis list in particular have always been present, as it served as a legend for the graph color-coding, but was not initially meant to be interactive.

After one of these iterations we have received a suggestion to allow user to shift the hypotheses in place, practically as a test of strength to see if the targeted code manipulations we wanted to make with the duplicate extractor, explained in the latter part of the 4.4.1, are possible.

Hypothesis editor is a Shiny default modal dialog that allows user to replace the columns for the hypothesis, or add the new names. If opened from the hypotheses list, as it was initially, changes (if made and confirmed) will affect all of that hypothesis's functions. Upon which, as soon as the change in the source code get registered, the visualization will update itself.

4.5.4 Drag-select and hypothesis injection

While changing the hypothesis indiscriminately for all of its functions may be viable for some of the usage scenarios, this is still a very crude tool.

Making it finer, however, would have required introduction of an individual node selection mechanism, which the add-in was lacking up to that point.

Moreover, as node selection became a candidate for implementation, it became apparent that a certain additional limitation mechanism will have to be implemented to restrict user from editing several hypotheses at once. We concluded that this limitation would have had to be implemented identically regardless of the complexity of the selection mechanism, which enabled us to make a try at implementing the more user-friendly drag selection, since failure to do so would not have had impeded the overall progression with the feature development.

Additional limitation had to have been introduced for the suggested visual node copy mechanism to work: aside

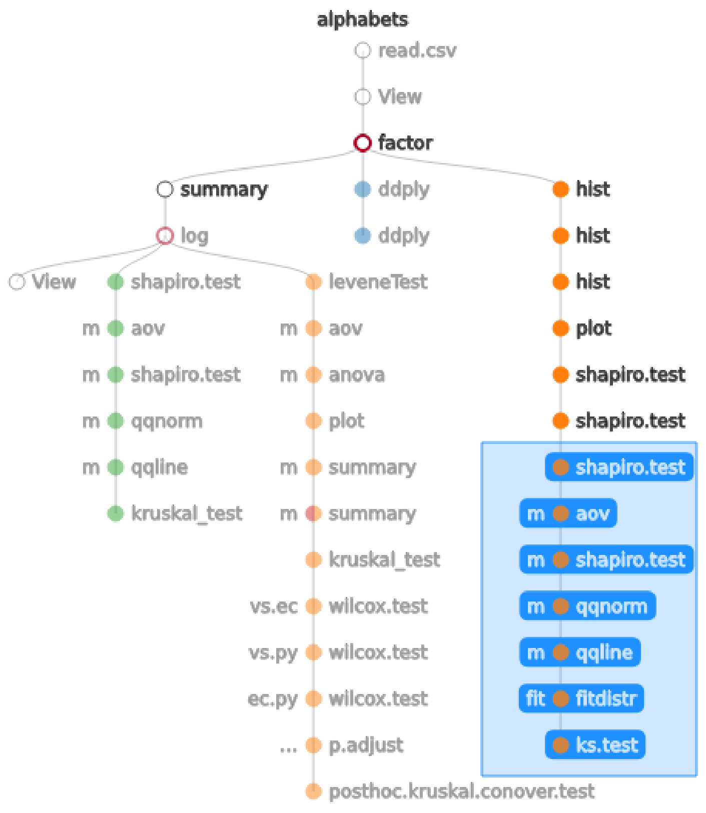


Figure 4.7: Drag selection

from having to belong the same hypothesis, only the nodes within the same code segment between the breakpoints could be selected. Figure 4.7 illustrates this, as functions on the left branch that are separated by the `log` function breakpoint are unavailable for selection despite belonging to the same ”orange” hypothesis.

The selection limitations are being updated as the cursor is being dragged through the canvas, but generally the breakpoint limitation relies only on the initial position of cursor, a dominant direction of its displacement (sign and module difference of its axis coordinates difference) and their relation to the nearest breakpoint. And the hypothesis limitation, on the top of that, checks whether or not any hypothesis-marked node have already been selected. If they are, other hypotheses' nodes within the ”breakpoint region” become disabled as well, however breakpoint node itself and non-hypothesis nodes never do.

When the nodes are selected, Edit and Copy buttons become available for the user. These operations use Hypothesis editor as their interface, but the server-side changes in processing. Selection edit operation is different from the mass-edit only in that is doesn't change all of the functions, but only the selected ones, while copy operation also has to factor in the next closest breakpoint position, and will insert the copied code just before it.

Chapter 5

Evaluation

Reiterating form 2.1, with our add-in we were addressing the following problems that arise in the exploratory programming workflow:

- Backtracking and safekeeping leading to the accumulation of "hoards" of functionally similar code
- Data-scientists having to reintroduce themselves to these hoards of code after switching back to code after dedicating time to other aspects of their work, like reporting or data accumulation

We leave the issue of intercontextual and intermodal code duplication aside since is not confined within a single environment we could address it in.

We attempt to disincentivise the hoarding behaviour and code duplication by providing the tools that allow for easier experimentation, presented and described in 4.5.3 and 4.5.4. While the core problem of backtracking while the exact set of operations that will be used for exploration of each of the hypotheses is being developed is left unaddressed, we propose that as soon as (if ever) user turns from this process to iteration through the data column relationships, being able to switch them at once and come back, again, at once either through the undo operation or another switch would be able to address the core of the duplication issue.

Hypothesis Manager's visualization also provides a mind-map of the user's code, which, while not necessarily aligning exactly with their intention, would help to navigate it without having to read or do a text search through the script file.

5.1 Study Design

While we would have liked to test both of our allegations, hoarding is a persistent behaviour that could only be impacted with a long-term treatment, that we couldn't see to be implemented in the confines of the thesis timeframe. We see an optimal way to study it in releasing the add-in R package into the open source and collecting the code samples and screen recordings of work from volunteers over time, similar to how Subramanian et al. [2019] have gathered their data.

Therefore, we have have decided to focus our study on the second aspect.

We were going to compare the quality of the users' recognition of the scripts' context and internal logic and structure, by asking them to read and interpret R code samples in pairs of equivalent size and structural complexity.

In each pair, one script were to be read with RStudio default interface and functionality, whereas for the other the participant would be permitted to use the Hypothesis Manager. The interpretations were going to be recorded, and recordings were to be graded for fidelity by an invited R expert. The application of Hypothesis Manager to the scripts in the pairs would have been alternated between the participants to provide an even representation within each pair. Participants were also to be inquired on their experience with R language and significance testing to provide a frame of reference for their answers and even the expectations for the results.

Additionally, we considered taking metrics on speed of the participants' preparation to present their interpretations, and utilized means of navigation within the RStudio, either with or without the Hypothesis Manager enabled.

5.2 Failure to find participants

Over the period of three weeks we have advertised our study with multiple online platforms and communities with the theoretical maximum cumulative outreach of approximately 23 thousands. This includes, in chronological order, first our personal acquaintances on the social networks and whomever the word of mouth could have reached, then the r/rstats subreddit[1] — an online community dedicated to R language with approximately 22 thousands subscribed users at the time of the ad submission, then Köln R User Group[2] — a local, mostly offline meetup community of 904 (again, at the time of the ad submission).

Of course, it would be preposterous to suggest that more than a hundredth of these people have actually seen our advertisement, but we were still able to receive a more than average positive reception on the r/rstats, and to find two volunteers for the study with the caveat that it would have had to be performed via the remote control software (which, while being inconvenient, would not violate the study design).

Scheduling the session, however, became an issue, since one of these volunteers have not answered the request, and the other had to withdraw due to personal issues.

At the time of submission of this report, we have just started to receive delayed responses to some of our initial inquiries, including a new volunteer, but the circumstances do not allow us to act on this opportunity to add to this report's value.

[1]A link to the posting: `https://www.reddit.com/r/rstats/comments/as9x4t/im_working_on_rstudio_shiny_plugin_that_aims_to/`

[2]`https://www.meetup.com/KoelnRUG/`

Chapter 6

Summary and future work

6.1 Summary and contributions

We have created a prototype for the programming product — interactive RStudio add-in — that addresses the issues of exploratory programming in the context of significance testing.

Aside from the prototype as a whole, we also consider following our contributions:

- As a backbone for the add-in, we have introduced a model for the exploratory code representation, based around a concept of the hypotheses that are being tested by the user.
- To generate the data according to this model, we have developed a semantic wrapper and utility extensions for the R language syntax parser.

Although the graphical user interface of the Hypothesis Manager also contains original solutions and design decisions, with the lack of a proper user evaluation we hesitate to highlight any of those.

6.2 Future work

6.2.1 Evaluation

As we were not able to conduct the evaluation, correcting this should take priority for the continuation of this project.

Furthermore, if R users recruited for the evaluation show interest in seeing the add-in reach maturity, it may be possible to recruit some of them for repeated usability studies in the benefit of the graphical interface of the add-in. We were not focusing on the UI/UX development in the context of this thesis due to being primarily concerned with prototyping, but the lack of study subjects would not have allowed us to successfully do that to begin with.

6.2.2 Improvements

In the process of the development we had numerous moments where the choice between several features or improvements had to be made. As a general rule, we were first prioritizing the creation of a Minimal Viable Product, then making target enhancements that would advance the quality and reliability of the parser, or fix the issues that became apparent from the experiments with the code base.

While some of these were later re-prioritized and implemented (see 4.5.4 for an example of such a feature), a lot of quality-of-life improvements had stayed in the "To Do" list.

Before going into the specifics, we would also like to make a broad disclaimer, that the general code quality, especially on a project with a single developer, regardless of their discipline or experience with the tools, can always be improved. In a similar sentiment, we understand that the visual metaphor we have settled with can benefit from more user studies and be iterated further. This is in part why the strong focus was put on creating a comprehensive code data model, which could, as was our intention, allow for independent refinement of various parts of the project.

Parser-Server decoupling

On the subject of independence, introduction of code scanning window (see 4.2.1) had forced a compromise on our intention to keep the shiny server clearly separated from the parsing algorithm. Peculiarities of R language exception handling mechanism necessitated a merger of the visual progress updates into the scanning window loop, which is inherently a part of the parser.

We believe that this merger should be deconstructed, as, ideally, parsing algorithm should exemplify a *generator* function. Although generators are not a part of R language paradigm, there is evidence [Rudolph, 2018] that this behaviour is possible to replicate.

Partial parsing

On the subject of the parser, as it was mentioned in 4.3.1, the time spent on re-parsing of the code could be reduced if the code-expression association were to be applied to skip the unchanged parts of the script file. Details of this particular improvement are yet to be considered: for instance, whether the entirety of the code after the modified chunk should be re-parsed in case the changes have altered its meaning, or if this behaviour could somehow be conditioned to only apply to the affected parts of the script.

Scopes and user-defined functions

Wrapping up with the parser, user-defined function, both named and anonymous, have ended up being altogether ignored by the algorithm. While not a particularly frequent occurrence within the project code base, handling them properly is an important stepping stone for further development of the add-in, as it will likely involve an introduction of *scope* entities to the data model.

Scopes could then be used to better contextualize other parts of the code, like control statement blocks and lambda-like anonymous functions.

RStudio REPL block

The issue of Shiny server blocking the REPL interface of the RStudio has been thoroughly explored in 4.3.2. If relevant changes to the RStudio get implemented, or a bypass to this problem is found, the add-in would cast away its only hindering aspect.

Non-tree visualization hierarchy

Mentioned in 4.4.2, the visualization graph nodes are bound by their tree hierarchy, while direct model comparisons like `stats:anova` require several hypotheses' branches to be able to converge and introduce graph loops.

While in the code base, instances of such code still allow for branches to be associated with one another due to the happenstance of proximity of their placement, introduction of the different hierarchy or overall graph generation mechanism would remove this concern altogether.

Better hypothesis editor view and non-regex replacement

Current implementation of Hypothesis Editor (see 4.5.3) lacks functionality that would allow to modify the hypotheses' formulas more that with interchange of the column names. Extending it to be able to use the full extent of the formula syntax while preserving the current functionality of selecting columns seems challenging, but would increase the power of this tool manyfold.

Code selection synchronization

There is no principal limitation for manually made source code selection ranges not to be represented in the add-in visualization, as `rstudiapi` provides the list of them, among other parts of source editor context. And client-server transmission of this information can be added into the same event loop as code parsing, since this script editor code contents is retrieved from the very the same context.

Enabling multiple custom outputs per application should also be an easy process, since multiple of base shiny inputs and outputs are generated without any additional commands aside of the element creation.

However, for unknown reasons, we were not able to make this client-server interaction work in the dedicated time frame, so it has been relegated to the backlog.

Deeper reflection

It should be theoretically possible to find which exact internal function is getting invoked in the case of generic functions like `summary`, at least when plug-in is in the dataset-enhanced mode. This information would make tooltips' help page links more precise.

Obtaining the reflection information in R language, however, seems to require the execution of the functions, and, to that end, tweaking of the parser internal rule of only executing the assignment statements.

6.2.3 Features

There also were some full pieces of new functionality that have been considered, but set aside for resource considerations.

Semantic labeling of graph nodes

Replacing the function name labels with their generalised semantic purpose was a suggestion that went in completely opposite direction to what lead to the introduction of the tooltips (see 4.5.1), but that may be viable in conjunction with the tooltips, that can serve as a counter-balance, still providing the full information on the function.

The thinking behind it was that some of the function names are generic or ambiguous, and having them replaced with short full-text descriptions like ″model creation″, ″hypothesis testing″, ″potting″ would not impede the recognition of the function place in the script, as it already very poor, but improve the representation of the workflow in general.

Duplicate extractor

The feature that was kept in the ″Future work″ from the very beginning of the development of this thesis, it would rely on several of the aforementioned improvements to be successfully introduced.

The details on its implementation are far from being clear, however. For example, how the extracted subroutine should be represented in the visualization, should it separated as the independent context and just referenced by name in the overall graph similar to a library function, or should it be extended for each of the occurrences? In the latter case, how should the navigation behave, should it go into the function body or into the particular instance of the function's invocation?

Bibliography

Rstudio. URL `https://www.rstudio.com/`.

Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. Code thumbnails: Using spatial memory to navigate source code. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pages 11–18. IEEE, 2006.

Mary Beth Kery and Brad A. Myers. Exploring exploratory programming. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–28, 2017. doi: https://doi.org/10.1109/VLHCC.2017.8103446.

Mary Beth Kery, Amber Horvath, and Brad A. Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, pages 1265–1276, 2017. doi: http://dx.doi.org/10.1145/3025453.3025626.

Johannes Maas. *StatWire: Visual Flow-Based Programming for Statistical Analysis*. PhD thesis, RWTH Aachen University, 2017.

James G. March. Exploration and exploitation in organizational learning. *Organization science*, 2(1):71–87, 1991.

Evan Patterson, Robert McBurney, Hollie Schmidt, Ioana Baldini, Aleksandra Mojsilovič, and Kush R. Varshney. Dataflow representation of data analyses: Toward a platform for collaborative data science. *IBM Journal of Research and Development*, 61(6):9–1, 2017.

R core team. R language definition, 2018. URL `https://cran.r-project.org/doc/manuals/r-release/R-lang.html`.

David Robinson. The impressive growth of r, 2017. URL `https://stackoverflow.blog/2017/10/10/impressive-growth-r/`.

Konrad Rudolph. Python-like generators in r, 2018. URL `https://gist.github.com/klmr/d10623a0b4c7e1e9a6523eebee4913d1`.

Beau Shiel. *Power tools for programmers*. Morgan Kaufmann Publishers Inc., 1983.

Krishna Subramanian, Ilya Zubarev, Simon Völker, and Jan Borcheres. Supporting data workers to perform exploratory programming. In *CHI*, 2019.

Index

Typeset March 5, 2019