PythonVis: Software Visualisation in Virtual Reality for Program Comprehension

Mattias Larsson
KTH Royal Institute of Technology
Stockholm, Sweden
matlar4@kth.se

ABSTRACT
This paper presents PythonVis, a novel Virtual Reality (VR) software visualization prototype for program comprehension. The motivation for PythonVis is to leverage the affordances of VR and the debugger tool to support software developers' comprehension of novel software. An experimental study with follow up interviews was conducted using 10 participants, comparing PythonVis to a desktop setup. The results indicate that PythonVis could be useful for getting a better overview over a whole code base. Limitation are addressed and further studies are suggested.

Author Keywords
Code visualisation; program comprehension; virtual reality; user experience; python; computer education

INTRODUCTION

Program comprehension, the process of understanding how a software system works, is a prevalent field of research, as developers spend a substantial amount of time reading and learning about other developers' software [21][23]. A common praxis of program comprehension is reading source code in an *Integrated Development Environment* (IDE) and a majority of developers use a debugger when exploring an unknown code base [11].

Virtual Reality (VR), as a medium, offers a number of affordances that could be of use in the context of program comprehension. The larger field of view offered by VR allows for more information to be available to users [24], along with a larger screen space, which has shown been shown to facilitate learning [6]. Furthermore, VR has been shown to leverage certain cognitive mechanisms, such as spatial memory and motion, to positively affect comprehension, perception, navigation and remembrance [5][4].

A few studies have investigated program comprehension in VR. Those studies include displaying the UML diagram of a Java code base through spatial metaphors [15], displaying Java code on a virtual computer monitor [3] or comparing VR to a computer monitor when investigating a Java code base [19]. These studies tackled the problem of program comprehension by visualizing the codebase, with mixed results. However, since many developers use a debugger when familiarising themselves to new code [11], it is interesting to investigate that functionality in a VR program comprehension context.
This paper introduces *PythonVis*, a novel VR software visualization prototype. PythonVis aims to give developers a tool for program comprehension by implementing the debugger in a VR context. The ambition is to leverage the VR affordances, and the debugger, to improve developers ability to understand unfamiliar code.

The paper is structured as follows:
First, a literature review of previous studies on program comprehension and virtual reality. This is followed by a thorough explanation of PythonVis and the experiment used to evaluate it. Finally, the results from the experiment is presented along with a thematic analysis. Lastly the results are discussed and implications for further research and developmed of PythonVis is suggested.

THEORY AND RELATED WORK

This section will begin with a presentation of the key concepts *program comprehension* and *spatial metaphors*. Following this, a number of affordances and potential hazards of using VR will be presented. Finally, a number of studies which investigated program comprehension in VR is explored.

Program Comprehension and The Spatial Metaphor

In their 2014 paper, Maalej and Tiarks defines *program comprehension* as *"the activity of understanding how a software system or a part of it works"*. They analysed 28 developers and surveyed thousands more to get an understanding of how program comprehension is practiced. Their findings indicate that while developers' approach to program comprehension is very individual, a common practice is to use an IDE to read the source code, and to execute the code using a debugger (running the code step by step using breakpoints to investigate the code at runtime). Furthermore, they highlight how software development is an ever growing industry and because of this, program comprehension is a field of research in constant need of more studies as new programming languages and development practices emerge [11].

One way to facilitate program comprehension is to visualise a program in 3D space and when doing this, a number of *spatial metaphors* can be employed. Averbukh et al. defines such a metaphor as *"(...) a mapping from concepts*

and objects of the simulated application domain to a system of similarities and analogies" [1]. A widely used metaphor is the *city metaphor* in which a virtual city is presented where each building represents a program's classes and each district represents its packages [19]. A few studies investigating program comprehension using different spatial metaphors in VR have been done in the past; to be presented in an upcoming section.

VR Affordances

In their 2015 paper, Elliot et al. discusses how VR can leverage certain cognitive mechanisms to its benefit when investigated in a software engineering context. One example of this is how VR affects *spatial memory* and how having the ability to allocate code to a physical space helps with comprehension and navigation of the code base. Furthermore, they illustrate how the manipulation of physical objects can have positive effects on users' perception and retention, and how this also applies in a virtual world with virtual objects [4].

This is in accordance with the findings of Riva et al. who highlights VR's ability to improve the *visual-spatial ability* among users, as well as improving their *executive functioning* [17]. The latter, as defined by Shannon and Duckwitz, refers to *"(...) higher-level abilities in the following areas: planning, problem-solving, attention, mental flexibility, initiation, judgment, inhibition, and abstract reasoning."*. These affordances could certainly help users with the exploration and learning of an unfamiliar code base. Finally, while VR is primarily a visual medium, it has the potential to engage more of the senses through sound, haptics and movement. The concept of *Multimodal Learning*, and its benefits is heavily documented (especially in children), tracing all the way back to Montessori and her work [12][9][10].

Program Comprehension in VR

In their 2019 study, Romano et al. compared program comprehension in a traditional programming environment and using the city metaphor when displayed on a standard computer monitor and using a VR headset. The results indicate that the city metaphor improved program comprehension as opposed to more traditional methods, and that participants using the VR version performed tasks faster and found the experience more enjoyable [19]. Oberhauser and Lecon performed a similar study wherein six participants used two established spatial metaphors (terrestrial and universe) in VR to investigate program comprehension. They found that the use of VR provided users with a better comprehension of the program and found the program more fun and engaging to use than traditional means [15].

However, the use of VR for program comprehension also has its downsides and there are a number of potential hazards. For their 2020 paper, Dominic et al. had participants read through eight different Java projects (in an IDE mirrored in VR), which they had to explain the purpose of, along with their expected outputs. Their results indicate that while VR had a positive effect on the perceived productivity of the participant, the unfamiliar virtual environment, and increased *"(...) mental demand, physical demand, and overall task load"* lead to lowered concentration and thus, lower overall accuracy, when compared to participants using a standard computer monitor [3]. Furthermore, due to technical limitations of the VR equipment used in their study, Oberhauser and Lecon found that a number of participants experienced motion sickness due to low frame rate, along with information loss due to limited processing power, affecting the render distance in the application employed [16]. Further potential hazards of VR will be discussed in the following sections.

PYTHONVIS

Instead of using established spatial metaphors like those presented in [19][15], PythonVis was designed to be a VR version of *Visual Studio Code* (VSC), a widely used IDE. The reason for this decision was to have two programs that are as identical as possible with the only difference being the medium itself (VR headset or computer monitor). Since developers are already familiar with the text-based version of their code, this lowers the learning threshold of the new tool. Also, by not mapping the textual code to new metaphors, we can isolate the effect on program comprehension from using a debugger in VR, something that not has been studied previously. The option to mirror VSC in VR was considered but ultimately ignored as it was deemed to similar to the study by Dominic et al. (although they did not test runtime debugging) [3]. PythonVis was developed using the Unity game engine [22].

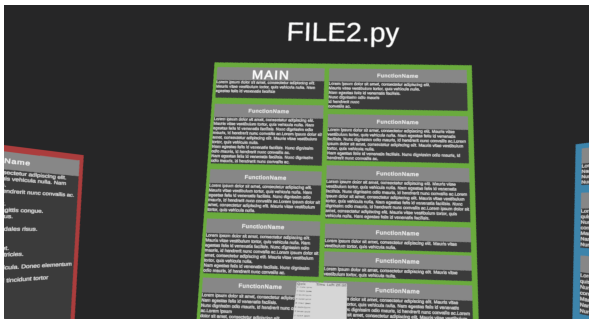

Figure 1. A screenshot of PythonVis before the experiment starts. Notice the small panel for questions at the bottom and the three files in the background.

When PythonVis is launched, the user is presented with three panels with placeholder text displayed; font, font size, font color and background colors were chosen to mimic those used in VSC, which also were in compliance with the findings of Kojić et al. who performed a study on readability in VR [8]. Furthermore, Kojić et al. highlighted the importance of a proper *angular size* (distance to text in relation to font size) for readability. Because of this, the user can point at these panels with their controllers and scroll with their thumb to move the panels closer or further away until the text in the panels feels comfortable to read. The panels contain a number of smaller boxes, one for each function of the program; each box contains the code for that function (if any code exists outside of a function, it is placed in a box titled *MAIN*).

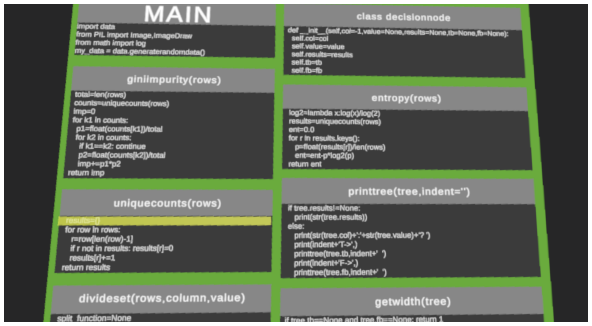

Figure 2. PythonVis. Showing the file *tree.py* at runtime. Notice the yellow bar indicating the current line the program is executing

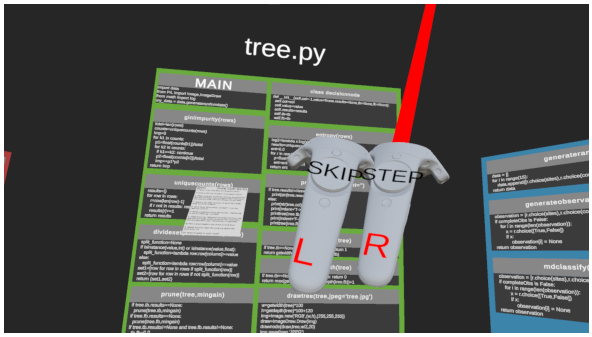

Figure 3. PythonVis. Showing the controllers, questions and laser pointer during the experiment.

A smaller panel with a timer in the top-right corner along with the numbers 1 to 10 followed by placeholder text is floating right in front of the user. This panel can be grabbed by the user and placed anywhere in the virtual space; this is to represent the questions and timer that will be given to the participants during the study. This implementation was made to avoid having the user remove their VR glasses in order to read the questions on a physical paper, thus breaking the immersion of the virtual world. Finally, a red laser can be seen coming out of the right controller. This was added to make it easier to select and adjust the panels but ended up being used by many participants to assist in reading.

When the spacebar is pressed (by the person running the experiment, not the participant), the aforementioned placeholder text is changed to the contents of the code base, the smaller panel in front of the user is filled with the questions and the timer starts. Just like in VSC, a yellow bar appears over the line of code that is currently being executed at runtime. The user can use their right touchpad to take one *step* forward the program, and their left touchpad to *skip* ahead in the program. These features replicate the *continue* and *step into* features of VSC, which will be detailed later. Originally, a feature was added that allowed users to take one step back in the code. However, this feature was removed from PythonVis as it impossible to do so in VSC.
After the spacebar is pressed, users can no longer adjust the distance to the code panels; this decision was made as it added a level of interactivity which could distract participants during the experiment.

METHOD

In this section, the experiment itself is presented, starting with a motivation for the layout of the study along with a presentation of the conducted pilot study. Following this, a presentation of each task which was given to participants is explained in chronological order.

Design Approach

When deciding how to do the study, inspiration was taken from a number of previous studies. A common practice in studies of this nature is to simply have a number of participants using one medium and others using the alternative, as seen in [14][3][15]. Doing it this way, with enough similarities between the applications, the impact on the results can hopefully be isolated to that of the medium. To ensure that the structure of the experiment was satisfactory, a pilot study was conducted before the actual experiments.

The pilot study had one participant using VSC and another using PythonVis to answer the questions which are detailed in the upcoming section. The results of the pilot study were minor changes to PythonVis and certain questions intended for the study being changed for clarity. The pilot study also resulted in the time limit being set to 25 minutes as both participants took between 25 and 30 minutes to complete the questions. A time limit was necessary as a way to make participants progress through the questions and not get stuck analysing the code for too long. Finally, the most important discovery of the pilot study was the need for a tutorial which was then designed for the actual experiments (detailed below).

The actual experiments were conducted over a two-day period in October of 2021 and 10 people participated: Seven males, two females and one who preferred not to say. At the time of the study, six of the participants were students at the Royal Institute of Technology in Stockholm, and the remaining four had full-time jobs as programmers in various fields.

Participant Tasks

Experience Form

Before the experiment, participants answered an *Experience Form* which asked them to rate their experiences with the following areas on a 5-point Likert scale: Python, VSC, VR, HTC-Vive and using a debugger. The results of this form can be found in appendix A.

Virtual Reality Sickness Questionnaire

As VR headsets are known to affect their users (nausea, disorientation, headache etc.), if the participant were to use PythonVis, they were asked to answer the *Virtual Reality Sickness Questionnaire* (VRSQ). Developed by Kim et al., this questionnaire allows users to quantify (on a 4-point Likert scale) the following symptoms commonly induced by the use of a VR headset: general discomfort, fatigue, headache, eye strain, difficulty focusing, fullness of the head, blurred vision, dizziness with eyes closed and vertigo [7]. The same questionnaire

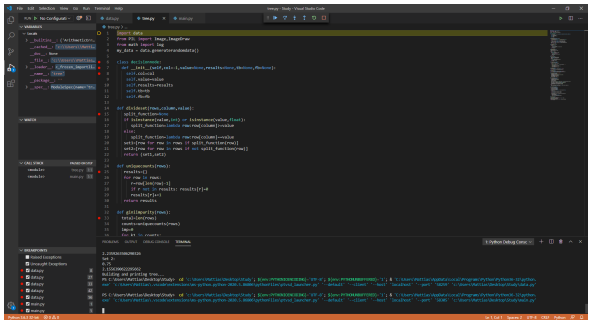

Figure 4. VSC. Showing the file *tree.py* at runtime. Notice the yellow bar indicating the current line the program is executing, the tabs at the top, the red break points and the call stack on the left and the syntax highlighting.

was also answered by the participants using PythonVis directly after the experiment to record any effects the app may have had on the user. The results to the VRSQ can be found in appendix B.

Visual Studio Code

Visual Studio Code, VSC for short, is a open-source code editor developed by Microsoft [13]. It was chosen for this study simply because it is very flexible and widely used (this was also indicated by the responses to the Experience Form). At the top of the program, a number of tabs indicate the different files and a yellow line indicates the line currently being executed. The background color, font, font size, font color were not altered for the purpose of this experiment.

When debugging in VSC, a number of features are available to the user: *continue, step into, step over, step out, restart* and *stop*. For this study, the number of features were limited to only two, *step into* and *continue*. This was partly due to the difficulty of implementing certain features in PythonVis but also as a precaution as to not overwhelm users who might not be familiar with debugging. As previously mentioned, 'step into' simply takes one step forward in the program, executing the next line of code. As the name suggests, 'continue' continues executing the program until it hits the next breakpoint, at which point it pauses. These breakpoints have been placed in the beginning of each new function to ensure that VSC and PythonVis operate in the same way.

The features in VSC outnumber those of PythonVis. In order to keep the two as similar as possible, in an attempt to find the true differences between the two mediums, certain features were prohibited from the participants using VSC. These features include the four other aforementioned stepping features, a call stack, the ability to click on the code to highlight similarly named items and, most importantly, the search function. Finally, VSC has *Syntax Highlighting*, a feature where different terms of the code are given different colors in order to make it more readable. An attempt was made to include this in PythonVis but due to how Unity handles text, it was omitted from the program.

The Code Base

Using the monitor intended for the study (15.6", with standard font settings), Visual Studio Code can display around 40 lines of code. Thus, in order to take advantage of potential benefits VR might have in effective code representation, a code base had to be chosen that was big enough to span over multiple files, and average more than 40 lines of code per file. The code base chosen for this study was taken from the book *'Programming Collective Intelligence: Building Smart Web 2.0 Applications'*, namely chapter 7 *'Modeling with Decision Trees'* [20]. The code base totalled 250 lines of code which was then split up over 3 files: *main.py* (58 lines, 1 function), *tree.py* (115 lines, 11 functions, 1 class) and *data.py* (77 lines, 5 functions).

The code base creates a randomised dataset and using this, builds and trains a *decision tree*; a simpler machine learning method for data classification using a tree structure [20]. After the tree is created, a number of operations are performed on the tree, such as: pruning, printing, entropy calculation etc. The full code base can be found in appendix C.

Experiment

At the start of the experiment, participants were given a short tutorial of the respective tools they were about to use. The tutorial consisted of a short Python program which only prints text to the console. In the program, a number of breakpoints were interspersed throughout the code for the participants to get familiar with using the stepping features of either debugger.

After the tutorial, participants were informed that they would have 25 minutes to answer ten questions in order. They were told that questions one and two were related to the beginning of the program and that question three could take a long time to answer. They were instructed this as a precaution so that they would not step past relevant information in the beginning, or get intimidated by the difficulty of the third question. Furthermore, participants were instructed to provide answers to the questions verbally so that the response time and the answer could be recorded by the author; this method was used for all users as to have consistency between the two different mediums and writing answers down in VR can prove difficult. The questions used in the study were:

1. What is the first thing the program does?

2. What is the first function the program calls?

3. What does the "buildtree" function do? Describe how the functions called relate to each other (e.g. this function then calls this function to get this)

4. What is the main purpose of each file? What does each file handle?

5. What does the function "prune()" in the file "tree.py" return?

6. What parameters does the function "mdclassify" in the file "data.py" receive?

Figure 5. A participant during the experiment. A view of what they see is mirrored on the monitor in the background.

7. What does the "drawtree" function do? Describe how the functions called relate to each other (e.g. this function then calls this function to get this)

8. How many functions are there in each file?

9. Which function does the program spend the most time in?

10. What function is never used?

Questions one and two were designed to require the use of the stepping function. Questions three and seven were designed to have users read through functions line by line. Questions five and six were placed right after each other as they relate to information from two different files, which may give some information about navigation in the two mediums. Questions four and eight relate to getting an overview of the whole code base. Questions nine and ten were placed last as they may be easier to answer after having studied the whole code base for a while. Some of these questions were expected to be easier in PythonVis, some were expected to be easier in VSC.

At the start of the experiment, these questions were placed in front of the user (virtually in PythonVis) along with a timer counting down from 25 minutes. Participants answered the questions in order, at their own pace, and they were informed when half the time had passed and when only five minutes were remaining. Throughout the experiment, audio was recorded to capture any comments or questions made by the participants. The response time for each question (time between reading a question and providing an answer) was also recorded throughout the experiment, using the aforementioned timer.

Semi-structured Interviews

After the experiment, a short semi-structured interview was held with the intent of gathering qualitative data related to the experiment. The questions used for the interviews were:

1. How do you think answering the questions went?
2. Can you explain in broad terms what the program does?
3. Were there any functionality you wish you had access to?
4. Do you think I will find any differences between using VR and not using it?[1]
5. Any other thoughts, comments or questions?

Audio recordings from the experiments and the interviews were transcribed by the author using Microsoft Word, and then adjusted for accuracy and readability. During this process, participants' names were encoded to protect their privacy throughout this report[2]. Finally, overall score and time spent per questions were logged and a thematic analysis was performed, the results of which are presented in the following section.

RESULTS

In this section, the results of the study will be presented. First, the quantitative data from the experiment will be presented, followed by a summary of the themes that were discovered during the semi-structured interviews that followed the experiment.

Experiment

The results are presented in two tables below where each cell represents a question. The contents of a cell indicates the time spent on that question (in the format m:ss), and the color of the cell indicates whether the question was answered correctly or not; green for correct, red for incorrect.

PythonVis

	VR1	VR2	VR3	VR4	VR5
Q1	0:35	0:30	2:54	0:46	0:40
Q2	1:25	0:15	0:30	0:19	1:10
Q3	5:00	8:25	3:06	7:45	8:10
Q4	0:53	0:24	1:30	4:40	4:30
Q5	3:37	2:34	3:00	4:06	2:02
Q6	0:33	0:22	0:16	0:51	0:22
Q7	2:20	1:30	3:04	3:03	2:56
Q8	0:52	1:18	0:17	0:23	0:17
Q9	1:20	1:28	0:53	0:57	0:40
Q10	3:07	8:14	2:50	2:00	1:33

Table 1. Results of the experiment for the participants who used PythonVis.

Out of a maximum score of 10, the participants who used PythonVis got an average score of 6.6 (1.95 standard deviation) and spent an average of 22 minutes answering all of the questions. Note that all participants answered correctly on questions 3, 6 and 7 and failed on question 9.

[1]Before asking this question, participants were shown the other program that they had not used during the experiment.

[2]For example: 'VSC2' means the second participant using VSC etc.

Visual Studio Code

	VSC1	VSC2	VSC3	VSC4	VSC5
Q1	0:40	0:27	0:05	0:10	1:05
Q2	0:50	0:51	1:16	0:43	1:34
Q3	3:24	6:39	15:12	6:27	3:15
Q4	1:56	1:38	0:33	3:32	3:15
Q5	2:47	2:15	1:55	2:33	3:02
Q6	1:46	1:00	4:12	1:31	1:20
Q7	3:07	2:05	0:43	1:44	0:00
Q8	1:45	0:50	0:45	0:39	0:34
Q9	1:20	1:47	0:00	0:41	1:22
Q10	4:42	5:34	0:00	0:42	0:57

Table 2. Results of the experiment for the participants who used Visual Studio Code.

The participants who used VSC got an average score of 5.4 (2.79 standard deviation) and spent an average of 20.5 minutes answering all of the questions. Similar to the people who used PythonVis, all participants got question 6 right while also failing question 9 (which means that all participants failed question 9). Note that there are 3 cells in which the time recorded is '0:00'. In the case of VSC5 (who failed to follow the instructions) some questions were answered out of order, leading to question 7 being skipped entirely. Finally, one participant ran out of time after answering question 8 and thus gave no answer to question 9 or 10.

Interview Themes

In this section, themes found throughout the interviews will be presented and exemplified through quotes from the participants. The findings are presented in a descending order of frequency.

Interactivity in VR

Most participants highlighted the importance of being able to interact with the code -as in, clicking the code to get more information- for the sake of program comprehension, and how this interactivity was missing from PythonVis. This was indicated by eight participants in total, and many of the suggested features were given as a response to questions 3 and 4 during the interview. The suggestions can be categorised into 3 major categories: A visual representation of function calls, rearranging the virtual world to avoid head-turning, and finally, interacting with the code panels.

Four participants mentioned being able to click on a function call and getting some sort of pointer to the function being called in the form of a line renderer or an arrow.

"(...) being able to press the function and then see that aha, it goes over there, rather than it [the tracker] jumps around and you have to look around, instead you just get that connection." -VR1

"(...) have like an arrow or something that leads you to that function. I think it is hard to, like, look for stuff [in VR] (...)" -VR4

Three participants suggested that whichever file was currently being executed should be placed in front of the user in the virtual world, as opposed to the user turning their head around to find that information themselves. Interestingly, one participant (VR2) did experience slight neck strain after the experiment for this reason.

"Yeah, I kind of feel like having it all around you is maybe a bit too much to be in this position [turns head back and forth] for a long time. But maybe, like, having the option to, like, scroll through it to have it always be in front of you. Yeah, that would maybe be quicker." -VR2

Finally, three participants suggested that users should be able to interact with the code panels directly, either by picking them up, walking around them, moving them around etc.

"I think that would be, like, a really good quality for VR, like, being able to, like, move everything, like, and then you're like, oh you could just put all these things to the side while I focus on this (...)" -VR4

"(...) if you had a bit more- if you had the option to take these [the functions] and put them wherever you want and spread them out, then this [VR] would be really nice." -VSC4

Overview in VR

Seven participants found that VR provided them with a better overview of the program as a whole.

"Yeah, I like how you have, like, so much space to play around with because I often am so frustrated with my computer screen because it's so packed and then I have to- like, I can't have like all of them open at the same time(...)" -VR4

"Because when you're in VR, you have like a 360 space so you can have, like, files all around you but when you're in a computer and have like many files, it's... I don't know if it's if- when you are in VR you have a better overview of everything (...)" -VR3

Two participants held the opposite view and felt that it would be easier to explore a code base on a computer due to the lower effort required to look at different parts of the code.

"Here [in VR] you need to move around your head while in- on a monitor you only have to move your eyes a little (...)" -VSC2

"That is the hardest part, like, having to physically look around while on a computer you would have the two functions next to each other almost." -VR5

Not Using Debugging Features

During the experiment, six participants were observed using the debugging features of their respective programs very scarcely. This observation was most apparent in the PythonVis group, with two participants using the stepping functionality,

and four participants not using it at all. When examining the experience form, all the offending participants rated their experience of *"using a debugger"* at either a two or a three (out of five). VR1 found the feature annoying to use in PythonVis and thus stopped using it (see section *"Losing Track of Current Line"* below). VR2 and VR5 opted to just read the code instead of using the stepping features; VR2 later mentioned that they forgot the feature existed. VR3, VSC4 and VSC5 were using the features but felt they got stuck in a loop, only returning to the same breakpoints and thus, stopped using the features.

Syntax Highlighting

Five participants noticed the lack of syntax highlighting in PythonVis, stating that its absence made the code harder to read and certain things harder to find.

"(...) if I had, like- all methods being called in one colour or something, then I would have maybe been a bit quicker." -VR2

"(...) it makes it very easy for me to see what I can skip, depending on what I'm looking for." -VSC4

One participant didn't notice the lack of color in PythonVis until it was pointed out to them, after which they said:

Maybe that's also why it was so hard to get through the code, cause they're [sic] so monotone. -VR4

Search Function

Five participants wished that they had access to a search feature to find specific details in the code, such as functions or function calls.

"It took me a lot of time to find specific things like 'prune' and stuff like that. It would go a lot faster if I could just look them up" -VSC1

"(...) if I could search for 'variance' I could see, like, if it would be in any of the calls, for example." -VR2

Readability in VR

Four participants mentioned that the text in VR was hard to read.

"(...) I liked watching it bit by bit, but there was something with the resolution, or if it was the contrast and such that... made me confused." -VR1

Three participants found it hard to read the line that the yellow indicator was on.

"(...) it was a bit hard with the contrast with the yellow and the white text." -VSC3

Many participants used the laser pointer on the right controller to assist in reading. When asked whether this was just a nice feature or a necessary crutch to use, one participant said:

"It was very pleasant being able to use it. It would have been possible without it but it would have been harder." -VR5

Running the Code

Three participants wanted to run the code and study the output in order to understand it better.

"I would have liked to run the code. To see [the program's output], and it would have made more sense." -VSC1

Comments

Three participants wanted the code to be commented to better understand the code base.

"(...) this buildtree one, it's too- With my Python knowledge it's a function that is too complex for such little time. If i had some comments maybe it would've been possible." -VSC2

"I think for the other function of the code, can take some notes [sic] describe what they do. Maybe it's more easy to just skip that and just read the note" -VSC5

Keyboard in VR

Three participants suggested that a virtual keyboard and mouse could be implemented instead of the Vive controllers.

"I think, for my part it would've been almost better with a normal keyboard (...) because you're more used to writing code with a keyboard." -VR1

Losing Track of Current Line

Three participants lost track of what part of the program was currently being executed when using the stepping features of the debugger; two participants experienced this in PythonVis and one experienced it in VSC.

"(...) when you used this 'skip' or 'step', you pressed it and then you had to find where it had gone, so I stopped using them." -VR1

(...) I barely understood that [VSC] skipped to a different file, but it feels like there in the VR thing you get it. Like, it was all there." -VSC3

Stepping Features

One participant wanted to be able to restart the program as well as take a step back in the code.

"(...) 'step back' and maybe, like uh, would be nice that I could like, restart the program (...) when I run it I was not able to, like, stop it and- and start again" -VR3

Furthermore, one participant wanted the 'step out' feature, which does exist in VSC but was not permitted to be used during the experiment.

"I would like a 'step out' because I don't need to go down into this entropy function every time (...)" -VSC4

DISCUSSION

In this section, the themes found in the results are discussed in relation to previous work within the field and the data gathered in the experiments. The method employed in this study is thoroughly discussed and finally, opportunities for future research is presented.

Better Overview, Worse Details

The main takeaway from this study is the fact that VR, as a medium, seems to give users a good, broad overview of a code base, but fails in comparison to a traditional setup when it comes to reading the code line by line. This was made evident during the interviews as seven participants found the visualisation approach satisfactory and four participants noted that some text was hard to read. These findings coincide with those of Romano et al. and Dominic et al. The former's study used established spatial metaphors to display an overview of a code base and yielded mostly positive feedback, saying that the VR tool was more useful and playful to use. The latter had the code on a flat virtual monitor in VR, and received more negative feedback from participants, stating that it felt like it added an unnecessary layer, a new foreign environment to access the same program.

Thus, it is understandable that PythonVis would yield the presented results as it exists in the proverbial intersection between these two studies. The visualisation employed receiving positive feedback for providing a better overview by utilising the wider screen space that the medium offers, and the raw code displayed in PythonVis receiving negative feedback, as it suffers from the same limitations as those presented by Dominic et al.; an added unnecessary layer which obstructs readability.

The better overview can be further observed in the differences between the two groups' responses to questions 6 and 8 which were primarily focused on locating functions throughout the files. Participants using PythonVis took on average 29 and 37 seconds to answer these questions respectively, compared to the 118 and 55 seconds of the participants using VSC. Many participants were positive to the visual representation of the code base, stating that it was very clear which code belonged to what function and, in turn, what file that function belonged to. One participant noted that he, at a glance, could find out a lot of information: Which file is the most important, which file is the biggest, what is the general structure of each file etc. This could potentially be due to the visualisation method employed by PythonVis, which takes advantage of the increased screen space provided by VR.

Questions three and seven had participants reading functions line by line and, as expected, participants using VSC answered these questions in 419 and 92 seconds, and the participants using PythonVis answered them in 389 and 155 seconds respectively. However, do note that these averages include two extreme outliers: VSC3 spending over 15 minutes answering the questions (the average is 296 without VSC3) and VSC5 skipping the question giving a time of 0 seconds (the average is 115 without VSC5). When excluding these outliers, VSC performs better in the questions related to reading the code line by line.

The importance of syntax highlighting could be observed in the differences in response time to question five, 183.8 seconds for PythonVis and 150.4 seconds for VSC. As the answer to the question is "nothing", participants using VSC may have had an advantage as the syntax highlighting present in VSC shows all return statements in a separate color, making the absence easier to identify at a glance. When not identifiable at a glance, PythonVis users were then forced to scan the function line by line, which again, can prove difficult. This importance can further be observed in the responses to question 3. When attempting to find the answer, three participants using PythonVis mistook the class *decisionnode* to be a function which did not occur for any of the VSC users, possibly due to different colors being given to functions and classes.

Finally, question seven (*What does the drawtree function do?*), regards a function that uses an imported package, which is commonly displayed at the top of a Python file. Upon noticing the line which used the package, VR5 simply glanced up, noticed the package and continued with the question. VSC4 had to scroll all the way to the top of the file, locate the import line, and proceed to scroll all the way back down. An anecdotal example, but this sort of behaviour could be observed with all participants. VSC users scrolled up and down a lot, switching between files and spent a lot of time searching for certain functions or lines of code, which seemed to be an easier task in VR.

Interactivity

Eight participants talked about the importance of being able to interact with the code in order to learn about it. Some of the features they suggested adding already exist in VSC, such as the call stack, search function or syntax highlighting, but some suggestions were more related to how one could use the medium of VR to convey more information through interactivity. Firstly, the lines indicating function calls -as suggested by four participants- could use the 3D space provided by the virtual world to show how functions are connected. Attempting to show this information in VSC may prove difficult as the information is restricted to the 2D monitor, and drawing these connecting lines will likely obstruct valuable information [18].

Three participants discussed how they would like to be able to interact with the code panels themselves by picking them up or walking around them. While they did not give a reason as to *why* they wanted this, the addition of this could be an example of how you can employ VR's ability to use motion and manipulation for perception and retention. Being able to physically interact with the virtual objects could improve the navigation and comprehension of the code, as in the case of Code Bubbles, as presented by Elliot et al. [4]. Perhaps PythonVis could allow users to walk up to the code panels, rescale them, move them around, highlight them etc. While this would certainly employ the aforementioned affordances, it would also increase the physical load put on participants, perhaps to the point where the application would become too

inconvenient to use over a longer session.

While motion and movement were not extensively utilised by PythonVis, the impact on the spatial memory could possibly be observed during the interview with VR1. When talking about the code they had just seen (while not wearing the VR headset), VR1 was gesturing and pointing to the places at which certain functions and files had been floating inside the VR world. If the participants using PythonVis were approached today, perhaps they could still identify where each file was placed and perhaps even some general information about each file. This example of VR's impact on spatial memory could be further investigated in future studies.

Python & Debugging

The original aim of this study was to fill an observed knowledge gap, by investigating how VR may be used when debugging code, and when coding in Python. When it comes to Python, it is quite a simple programming language with syntax that is not very strict [2]. As VR seems to make reading more difficult due to its limited resolution, Python is quite suitable for the medium as the simple syntax makes the code more readable (no semicolons, indentation instead of brackets etc.). Perhaps, this could be taken even further by implementing a programming language like Scratch in VR, with the Scratch blocks being 3D blocks in the virtual world as a way to teach programming to children, by utilising the aforementioned affordances of VR and multimodal learning [10] [4]. Given the positive results shown by Romano et al. as well as Oberhauser and Lecon, along with the positive feedback given to PythonVis' visualisation and ability to provide an overview, VR could potentially be used to provide an overview over a whole code base before moving on to a traditional monitor for further inspection. However, as most people are familiar with text based code, pursuing a virtual IDE, with better screen resolution and syntax highlighting, program comprehension for text-based python may still be improved by the use of VR.

As for debugging in VR, a number of participants were not using the debugging features extensively and opted to simply read the code. This may be due to participants not being used to using debuggers. For example: one participant had worked with Python on a daily basis, preferred to use other ways of troubleshooting their code. Furthermore, this phenomena may be observable in the Experience Form, as only one participant rated their experience with debuggers as higher than a 3.
In the case of PythonVis, users may also have stopped using the feature because when the tracker teleported in the virtual world, users became disoriented. One participant also mentioned feeling some neck strain from searching for the tracker so much. This could have been avoided by employing more of the modalities VR has to offer. For example by making the tracker animated or adding audio, could have made the tracker easier to locate. It may also have been circumvented by instead of moving the tracker to the files, the files moved in front of the user, as suggested by three participants.

Method Discussion

While this study yielded interesting results, there were a number of ways in which it could have been improved to strengthen its validity.

Design Oversights

Firstly, when creating PythonVis, the code from one function in VSC was used in two different functions in PythonVis. This was an oversight by the developer and should have been avoided for the validity of the study. Fortunately, this function was not essential to the questions asked during the experiment and the error was only noticed by one participant (VR5).

Another oversight was related to the final question of the experiment: *"What function is never used"*. There were actually two functions in the code base which were never called at runtime, meaning if participants had given either of these functions as an answer to question ten, they got the question right. This did affect the results as out of the six correct responses to this question, four gave one of the functions as an answer and the remaining two gave the other one.

Applications Were Not Identical

As presented in section 3, precautions were taken to ensure that PythonVis and VSC were as similar as possible, in order to ensure that the only differing factor was the medium itself. Despite these efforts, there are a number of ways in which the two applications differ. The participants using VSC had a call stack available on the screen and while only two participants used it (VSC3, VSC4) it should have either been added to PythonVis or removed from VSC for the purposes of this study. However, the most important inconsistency is the lack of syntax highlighting in PythonVis as it's inclusion would help participants finding variables, function calls or return statements (or lack thereof, as in the case of question 5).

Due to the limitations put on VSC in an attempt to keep the applications equal, one could argue that the experiment is no longer realistic, as a user of VSC would not have these restrictions in a real scenario. For a study of this nature to be more authentic, participants using VSC should have the freedom to use any feature as long as those features are also available in the alternate medium, which unfortunately was not the case for PythonVis.

Technical Limitations

The headset used for this study was a HTC Vive, released in 2016, with a resolution of 1080x1200 pixels per eye. Since then, a number of VR headsets have been developed with better processing power and resolution. While PythonVis did not suffer from performance issues like those of Oberhauser and Lecon, a better headset with better resolution (for example the Vive Pro 2 with 2448x2448 pixels per eye) might have helped participants with reading the code line by line; a seemingly difficult task with program comprehension in VR [15].

Question Ordering

While the order of the questions were considered when designing the experiment, the decision to place the third question in that position may have affected the results of the study. The

question required participants to explain a function in major detail, which may have been daunting to participants so early in the experiment. Furthermore, when simpler questions followed (such as question five and six), participants sometimes misinterpreted the question, assuming the task was the same as for question three, and thus, started explaining the relevant functions in major detail; a practice not required to answer the question. Because of this, participants spent more time than necessary on subsequent questions which may have skewed the results of the study.

The Future

If further development were to be done on PythonVis (or if this study were to be repeated) there are a number of things that should be addressed. Firstly, the panels with the code should be interactable in the same way as the question panel, to engage more of the affordances of VR using movement and motion. Additionally, more of the features available in VSC should be added to PythonVis, namely the call stack, syntax highlighting, search function, and most importantly, the ability to click on a line of code and get the same information as you would in VSC (similar words being highlighted).

As evident from the needed inclusion of the tutorial before the study, and the many suggestions of the virtual mouse and keyboard, a sort of mixed reality might be needed to ease programmers into the virtual world. Simply dropping them into a foreign world, with a foreign tool and a foreign code base may simply be too alienating. In his blog post *Working From Orbit*, Tomlinson details his virtual programming space, which he spends 8-10 hours every day immersed in [24]. The implementation that Tomlinson presents may be the future of the virtual office as it addresses most of the hazards, and makes use of the some of the affordances VR has to offer. While Tomlinson's tool (and PythonVis) does not employ the affordances of manipulation and motion, this may not be suitable for a tool which is intended for a full workday's use, as the continuous movement would be straining.

Ultimately, all people are different and while a virtual programming environment is suitable for Tomlinson, this may not be the case for everyone. In this study, while the VRSQ indicate that PythonVis had close to no effect on participants, there was one participant (VR4) who showed an increase in five out of the nine possible areas of discomfort, and this was after only a 25 minute session. If virtual programming were to become more ubiquitous in the future, these outliers must be taken into account as VR may not be the optimal medium for everyone.

Future Research

Future research could aim to correct some of the issues mentioned above. A study could be made to investigate readability of code in VR and try to answer the question whether the syntax highlighting or the limited resolution is the biggest contributing factor to code comprehension in VR. Furthermore, future research could investigate implementations of any number of the features that were sought after by the users of PythonVis; how does one visualise function calls, make a virtual keyboard or call stack, track the current line in a less disorienting way, etc.

Ethics

As this study occurred during the COVID-19 pandemic, all equipment used for the experiment was wiped down with disinfectant before each participant's session. Hand sanitiser was provided so that participants could disinfect their hands upon arrival to the lab.

CONCLUSION

Program comprehension is highly relevant in today's digitalised world. To explore debugging -a cornerstone of program comprehension- in VR, PythonVis was created and evaluated through an experiment and interviews. The findings indicate that while VR, as a medium, can be used to give programmers a good overview of a code base, it may not be suitable for reading large amounts of text. However, these findings may be due to a lack of features in PythonVis, or due to the limited resolution of the VR headset used in the study. This study lays the foundation and presents multiple ways in which future research could further explore the possibilities of program comprehension and debugging in a VR context.

Acknowledgements

I would like to express my gratitude towards my supervisor Martin Hedlund for his continuous support throughout the development of PythonVis and during the writing of this thesis. I would also like to thank Peter Hedeta for helping me with the pilot study and all of the other participants who gave such good insights during the experiments.

REFERENCES

[1] Vladimir Averbukh, Natalya Averbukh, Pavel Vasev, Ilya Gvozdarev, Georgy Levchuk, Leonid Melkozerov, and Igor Mikhaylov. 2019. Metaphors for software visualization systems based on virtual reality. In *International Conference on Augmented Reality, Virtual Reality and Computer Graphics*. Springer, 60–70.

[2] Andrey Bogdanchikov, Meirambek Zhaparov, and Rassim Suliyev. 2013. Python to learn programming. In *Journal of Physics: Conference Series*, Vol. 423. IOP Publishing, 012027.

[3] James Dominic, Brock Tubre, Jada Houser, Charles Ritter, Deborah Kunkel, and Paige Rodeghero. 2020. Program Comprehension in Virtual Reality. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 391–395. DOI:http://dx.doi.org/10.1145/3387904.3389287

[4] Anthony Elliott, Brian Peiris, and Chris Parnin. 2015. Virtual reality in software engineering: Affordances, applications, and challenges. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 547–550.

[5] Fatih ERKEN and Haluk BİRSEN. Cognitive Differences Between Online and Virtual Reality News in

the Context of Recall and Comprehension. *Yeni Medya* 2021, 10 (????), 1–24.

[6] Daesang Kim and Dong-Joong Kim. 2012. Effect of screen size on multimedia vocabulary learning. *British Journal of Educational Technology* 43, 1 (2012), 62–70.

[7] Hyun K Kim, Jaehyun Park, Yeongcheol Choi, and Mungyeong Choe. 2018. Virtual reality sickness questionnaire (VRSQ): Motion sickness measurement index in a virtual reality environment. *Applied ergonomics* 69 (2018), 66–73.

[8] Tanja Kojić, Danish Ali, Robert Greinacher, Sebastian Möller, and Jan-Niklas Voigt-Antons. 2020. User experience of reading in virtual reality—finding values for text distance, size and contrast. In *2020 Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*. IEEE, 1–6.

[9] Gerry Leisman, Ahmed A Moustafa, and Tal Shafir. 2016. Thinking, walking, talking: integratory motor and cognitive brain function. *Frontiers in public health* 4 (2016), 94.

[10] Angeline Stoll Lillard. 2016. *Montessori: The science behind the genius*. Oxford University Press.

[11] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 1–37.

[12] Dominic W. Massaro. 2012. *Multimodal Learning*. Springer US, Boston, MA, 2375–2378. DOI: http://dx.doi.org/10.1007/978-1-4419-1428-6_273

[13] Microsoft. 2021. Visual studio code - code editing. redefined. (Nov 2021). https://code.visualstudio.com/

[14] David Moreno-Lumbreras, Gregorio Robles, Daniel Izquierdo-Cortázar, and Jesus M Gonzalez-Barahona. 2021. To VR or not to VR: Is virtual reality suitable to understand software development metrics? *arXiv preprint arXiv:2109.13768* (2021).

[15] Roy Oberhauser and Carsten Lecon. 2017a. Gamified Virtual Reality for Program Code Structure Comprehension. *International Journal of Virtual Reality* 17, 2 (2017).

[16] Roy Oberhauser and Carsten Lecon. 2017b. Virtual reality flythrough of program code structures. In *Proceedings of the Virtual Reality International Conference-Laval Virtual 2017*. 1–4.

[17] Giuseppe Riva, Valentina Mancuso, Silvia Cavedoni, and Chiara Stramba-Badiale. 2020. Virtual reality in neurorehabilitation: a review of its effects on multiple cognitive domains. *Expert Review of Medical Devices* 17, 10 (2020), 1035–1061.

[18] George G Robertson, Jock D Mackinlay, and Stuart K Card. 1991. Cone trees: animated 3D visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 189–194.

[19] Simone Romano, Nicola Capece, Ugo Erra, Giuseppe Scanniello, and Michele Lanza. 2019. On the use of virtual reality in software visualization: The case of the city metaphor. *Information and Software Technology* 114 (2019), 92–106.

[20] Toby Segaran. 2007. *Programming collective intelligence: building smart web 2.0 applications*. " O'Reilly Media, Inc.".

[21] Janet Siegmund. 2016. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. IEEE, 13–20.

[22] Unity Technologies. 2021. (Nov 2021). https://unity.com/

[23] Rebecca Tiarks. 2011. What maintenance programmers really do: An observational study. In *Workshop on software reengineering*. Citeseer, 36–37.

[24] Paul Tomlinson. 2021. Working From Orbit VR Productivity In (or Above) a WFA World. (Sep 2021). https://blog.immersed.team/working-from-orbit-39bf95a6d385

Appendix A: Experience Form Results

This shows the responses to the Experience Form.

	VR1	VR2	VR3	VR4	VR5	VSC1	VSC2	VSC3	VSC4	VSC5
Python	4	3	2	2	4	2	3	5	4	1
Debugger	2	3	3	3	2	1	2	5	3	2
VSC	4	4	4	3	4	1	2	4	4	4
VR	2	2	2	1	3	3	2	2	3	2
HTC Vive	1	2	2	1	2	2	2	1	3	2

Appendix B: VRSQ Results

This shows the responses to the Virtual Reality Sickness Questionnaire. The upper value in each cell is the rating given before the experiment, the lower is the rating given after the experiment.

	VR1	VR2	VR3	VR4	VR5
General Discomfort	1	1	1	2	1
	1	2	1	3	1
Fatigue	1	1	2	3	2
	1	1	3	1	1
Headache	1	1	1	2	1
	1	1	1	2	1
Eye Strain	1	2	2	2	2
	3	2	2	3	2
Difficulty Focusing	2	1	2	2	2
	1	1	2	2	2
Fullness of Head	1	1	2	2	1
	1	2	2	3	1
Blurred Vision	2	1	1	1	1
	1	1	1	2	1
Dizziness With Closed Eyes	1	1	1	1	1
	1	1	1	3	1
Vertigo	1	1	1	1	1
	1	1	1	1	1

Appendix C: The Code Base

tutorial.py

```
def testFunction():
    print("line 1")
    print("line 2")
    print("line 3")

print("line 1")
print("line 2")
print("line 3")
print("line 4")
print("line 5")
print("line 6")
print("line 7")
print("line 8")
print("line 9")
print("line 10")
print("line 11")
print("line 12")
testFunction()
print("line 14")
print("line 15")
print("line 16")
print("line 17")
print("line 18")
print("line 19")
print("line 20")
```

main.py

```
import tree as t
import data as g

def buildtree(rows,scoref=t.entropy):
  if len(rows)==0: return decisionnode()
  current_score=scoref(rows)
  best_gain=0.0
  best_criteria=None
  best_sets=None
  column_count=len(rows[0])-1
  for col in range(0,column_count):
    column_values={}
    for row in rows:
       column_values[row[col]]=1
    for value in column_values.keys():
      (set1,set2)=t.divideset(rows,col,value)
      p=float(len(set1))/len(rows)
      gain=current_score-p*scoref(set1)-(1-p)*scoref(set2)
      if gain>best_gain and len(set1)>0 and len(set2)>0:
        best_gain=gain
        best_criteria=(col,value)
        best_sets=(set1,set2)
  if best_gain>0:
    trueBranch=buildtree(best_sets[0])
    falseBranch=buildtree(best_sets[1])
    return t.decisionnode(col=best_criteria[0],value=best_criteria[1],
                          tb=trueBranch,fb=falseBranch)
  else:
    return t.decisionnode(results=t.uniquecounts(rows))

data = t.my_data
print("The Gini impurity:")
print(t.giniimpurity(data))
print("The entropy:")
print(t.entropy(data))
set1,set2 = t.divideset(data,2,'yes')
print("Set 1:")
print(t.giniimpurity(set1))
print(t.entropy(set1))
print("Set 2:")
print(t.giniimpurity(set2))
print(t.entropy(set2))
print("Building and printing tree...")
```

```
tree = buildtree(data)
print(t.printtree(tree))
print("Drawing tree...")
t.drawtree(tree, jpeg="treeview.jpg")
print("Create observation...")
obs = g.generateobservation(True)
print(obs)
print("Make prediction")
prediciton = g.classify(obs,tree)
print(prediciton)
mdobs = g.generateobservation(False)
mdprediction = g.mdclassify(mdobs,tree)
print("Prune with 1.0 mingain")
t.prune(tree,1.0)
print(t.printtree(tree))
```

tree.py

```
import data
from PIL import Image,ImageDraw
from math import log
my_data = data.generaterandomdata()

class decisionnode:
  def __init__(self,col=-1,value=None,results=None,tb=None,fb=None):
    self.col=col
    self.value=value
    self.results=results
    self.tb=tb
    self.fb=fb

def divideset(rows,column,value):
   split_function=None
   if isinstance(value,int) or isinstance(value,float):
      split_function=lambda row:row[column]>=value
   else:
      split_function=lambda row:row[column]==value
   set1=[row for row in rows if split_function(row)]
   set2=[row for row in rows if not split_function(row)]
   return (set1,set2)

def uniquecounts(rows):
   results={}
   for row in rows:
      r=row[len(row)-1]
      if r not in results: results[r]=0
      results[r]+=1
   return results

def giniimpurity(rows):
  total=len(rows)
  counts=uniquecounts(rows)
  imp=0
  for k1 in counts:
    p1=float(counts[k1])/total
    for k2 in counts:
      if k1==k2: continue
      p2=float(counts[k2])/total
      imp+=p1*p2
  return imp
```

```
def entropy(rows):
   log2=lambda x:log(x)/log(2)
   results=uniquecounts(rows)
   ent=0.0
   for r in results.keys():
      p=float(results[r])/len(rows)
      ent=ent-p*log2(p)
   return ent

def printtree(tree,indent=''):
   if tree.results!=None:
      print(str(tree.results))
   else:
      print(str(tree.col)+':'+str(tree.value)+'? ')
      print(indent+'T->',)
      printtree(tree.tb,indent+'  ')
      print(indent+'F->',)
      printtree(tree.fb,indent+'  ')

def getwidth(tree):
  if tree.tb==None and tree.fb==None: return 1
  return getwidth(tree.tb)+getwidth(tree.fb)

def getdepth(tree):
  if tree.tb==None and tree.fb==None: return 0
  return max(getdepth(tree.tb),getdepth(tree.fb))+1

def drawtree(tree,jpeg='tree.jpg'):
    w=getwidth(tree)*100
    h=getdepth(tree)*100+120
    img=Image.new('RGB',(w,h),(255,255,255))
    draw=ImageDraw.Draw(img)
    drawnode(draw,tree,w/2,20)
    img.save(jpeg,'JPEG')

def drawnode(draw,tree,x,y):
  if tree.results==None:
    w1=getwidth(tree.fb)*100
    w2=getwidth(tree.tb)*100
    left=x-(w1+w2)/2
    right=x+(w1+w2)/2
    draw.text((x-20,y-10),str(tree.col)+':'+str(tree.value),(0,0,0))
    draw.line((x,y,left+w1/2,y+100),fill=(255,0,0))
    draw.line((x,y,right-w2/2,y+100),fill=(255,0,0))
    drawnode(draw,tree.fb,left+w1/2,y+100)
    drawnode(draw,tree.tb,right-w2/2,y+100)
```

```
  else:
    txt=' \n'.join(['%s:%d'%v for v in tree.results.items()])
    draw.text((x-20,y),txt,(0,0,0))

def prune(tree,mingain):
  if tree.tb.results==None:
    prune(tree.tb,mingain)
  if tree.fb.results==None:
    prune(tree.fb,mingain)
  if tree.tb.results!=None and tree.fb.results!=None:
    tb,fb=[],[]
    for v,c in tree.tb.results.items():
      tb+=[[v]]*c
    for v,c in tree.fb.results.items():
      fb+=[[v]]*c
    delta=entropy(tb+fb)-(entropy(tb)+entropy(fb)/2)
    if delta<mingain:
      tree.tb,tree.fb=None,None
      tree.results=uniquecounts(tb+fb)

def variance(rows):
  if len(rows)==0: return 0
  data=[float(row[len(row)-1]) for row in rows]
  mean=sum(data)/len(data)
  variance=sum([(d-mean)**2 for d in data])/len(data)
  return variance
```

data.py

```
import random as r

sites = ["slashdot","google","digg","(direct)","kiwitobes"]
countries = ["USA", "France", "UK", "New Zealand"]
opt = ["no", "yes"]

def generatestaticdata():
    data=[['slashdot','USA','yes',18,'None'],
        ['google','France','yes',23,'Premium'],
        ['digg','USA','yes',24,'Basic'],
        ['kiwitobes','France','yes',23,'Basic'],
        ['google','UK','no',21,'Premium'],
        ['(direct)','New Zealand','no',12,'None'],
        ['(direct)','UK','no',21,'Basic'],
        ['google','USA','no',24,'Premium'],
        ['slashdot','France','yes',19,'None'],
        ['digg','USA','no',18,'None'],
        ['google','UK','no',18,'None'],
        ['kiwitobes','UK','no',19,'None'],
        ['digg','New Zealand','yes',12,'Basic'],
        ['slashdot','UK','no',21,'None'],
        ['google','UK','yes',18,'Basic'],
        ['kiwitobes','France','yes',19,'Basic']]
    return data

def generaterandomdata():
    data = []
    for i in range(15):
        data.append([r.choice(sites),r.choice(countries),r.choice(opt),r.rand
    return data

def generateobservation(completeObs):
    observation = [r.choice(sites),r.choice(countries),r.choice(opt),r.randra
    if completeObs is False:
        for i in range(len(observation)):
            x = r.choice([True,False])
            if x:
                observation[i] = None
    return observation

def classify(observation,tree):
  if tree.results!=None:
    return tree.results
```

```
  else:
    v=observation[tree.col]
    branch=None
    if isinstance(v,int) or isinstance(v,float):
      if v>=tree.value: branch=tree.tb
      else: branch=tree.fb
    else:
      if v==tree.value: branch=tree.tb
      else: branch=tree.fb
    return classify(observation,branch)

def mdclassify(observation,tree):
  if tree.results!=None:
    return tree.results
  else:
    v=observation[tree.col]
    if v==None:
      tr,fr=mdclassify(observation,tree.tb),mdclassify(observation,tree.fb)
      tcount=sum(tr.values())
      fcount=sum(fr.values())
      tw=float(tcount)/(tcount+fcount)
      fw=float(fcount)/(tcount+fcount)
      result={}
      for k,v in tr.items(): result[k]=v*tw
      for k,v in fr.items(): result[k]=v*fw
      return result
    else:
      if isinstance(v,int) or isinstance(v,float):
        if v>=tree.value: branch=tree.tb
        else: branch=tree.fb
      else:
        if v==tree.value: branch=tree.tb
        else: branch=tree.fb
      return mdclassify(observation,branch)
```