Automated Generation of Data Quality Checks

Master's Thesis Project Description

Madelin Schumacher
Supervisors: Dr. Caterina Urban, Alexandra Bugariu

September 2017

1 Introduction

Data analysis programs are developed and used not only by computer scientists but also by biologists, statisticians and specialists from other fields. Even though these programs are correct, unexpected input data can lead to a program failure or to an erroneous output which might be difficult to detect.

Due to a common lack of documentation or because the user is not experienced enough it might not be clear how the input data has to be formatted, what value ranges can be used, what type the input data should have, do values have to be unique and so on. For programs needed in medical environments the input data used for a certain program might not be available to the programmer because of confidentiality issues.

This can lead to errors when the program is run by a user because the programmer could not test their software with real input data. [1] shows that most data errors happen for the following reasons: data is copied from a corrupted input source (e.g. misunderstanding someone at the telephone or working with illegible hand writing), input data that captures measurements of a physical process is wrong because of a flaw in the design or execution, a faulty preprocessing of raw data results in wrong values, or errors occur when merging multiple sources of input data and inconsistencies need to be resolved (e.g. different units, data representation, measurement periods and so on).

Many data analysis programs have to run for many hours to produce a result. It is cumbersome having to restart a program that ran for hours if an error occurred because of faulty input data. Often it is not feasible for the user to check the data by hand because data analysis programs work with a vast amount of input data.

To avoid running into program failures and to reduce the effort needed to check the input data by hand we will create a static analyzer and an input checker. Static program analysis is used to inspect a program's source code and infer information about it, without running it. This information can be used to check if certain properties hold. The result of this analysis will be used to automatically synthesize a checker that runs on the input data to search for possible errors. The checker reports problematic input values so that the user can fix those values before running the actual data analysis program.

2 Motivating Example

Listing 1 shows a data analysis program written in Python. The idea of the program is to read a file named *columnsAges.csv* containing comma separated lines with an identification of the town, the number of people living in the town and the ages of the citizens. As output it calculates and prints the average age per town.

The program goes through each comma separated line and splits it. The first value of the line is the identification of the town, the second value is the number of people living in the town and the other entries contain the ages of the people living there. The sum of these ages is calculated and divided by the number of people. Finally the value is printed.

This program is correct and will produce a correct result if the input data given in the file *columnsAges.csv* is as expected. However, the program makes several assumptions about the input data. If the data is not as expected it will lead to program errors or even worse, the program will finish without error but give a faulty output which might not be detected by the user.

The program will produce a *ZeroDivisionError* in line 7 if the number of people given in the input is zero. This could happen if each line is dedicated to a certain town and for one town no data was available. The program will raise a *ValueError* in line 4 if the input contains something other than comma separated numbers. For example if the data was created by hand and accidentally the letter *O* was inserted instead of the number *0*. An *IndexError* will happen in line 8 if the identification number of the town is not in the range of the *towns* array.

More difficult to detect is the case when a line does not contain as many ages as given in the second column of each line. If the number of people living in the town is smaller than the number of entries that are following, line 6 will cut off the remaining numbers and not consider them at all. Therefore faulty input data might not be detected because the program will report no error. If the number of people given is larger then the number of entries following, Python will simply ignore this and just sum the entries available. The user will not know that input data is missing.

```
1 towns = ['Zurich', 'Berne', 'Basel', 'Geneva']
2 inputfile = open('columnsAges.csv')
3 for line in inputfile:
4   columns = [int(value) for value in line.strip().split(',')]
5   num_people = columns[1]
6   sum_ages = sum(columns[2:num_people+2])
7   average_age = sum_ages / num_people
8   print('The average age in', towns[columns[0]], 'is', average_age)
```

Listing 1: Data Analysis Program in Python

To avoid these errors a programmer has to add error handling as shown in Listing 2. With this approach the errors would have been detected by the program itself. There are several reasons why the first program without the error handling might be used. The person creating this code might not be experienced enough to add error handling to the code or might just not consider possible errors of input data and just assume that it is correct. The code with the error handling is also much longer and takes more time to write.

Even the code from Listing 2 does not catch all errors that can happen because of unexpected input data such as using semicolons instead of commas to separate the values. This will result in cryptic errors for the end users and they might not be able to find the error in their input data or assume a faulty program. Additionally, the code with error handling will stop execution as soon as one error is detected. The user would have to fix the issue in the input data and rerun the program. This process has to be repeated every time the program finds a problem. It might even run for a long time before an error is detected. Using a tool that checks the input data before running the program will report all errors at once and the user can then fix those values and check the corrected input data file again.

```
towns = ['Zurich', 'Berne', 'Basel', 'Geneva']
inputfile = open('columnsAges.csv')
try:
  for line in inputfile:
    columns = [int(value) for value in line.strip().split(',')]
    num_people = columns[1]
    if(num_people == 0):
      print('The number of people in a town cannot be 0.')
      break
    if(num_people != len(columns)-2):
      print('The number of people is not equal to the number of ages given.')
      break
    sum_ages = sum(columns[2:num_people+2])
    average_age = sum_ages / num_people
    if(columns[0] > len(towns)-1):
      print('The town identification is unknown.')
      break
    print('The average age in', towns[columns[0]], 'is', average_age)
except ValueError:
  print('The value is not a correct number.')
```

Listing 2: Data Analysis Program with Error Handling in Python

3 Related Work

To ensure a successful program execution most approaches check correctness of the program itself by using formal verification ([2], [3]), model checking [5], or other means [4].

In contrast, we assume that the program works correctly for expected input data. Our goal is to ensure successful program execution by checking the input data for correctness. Less research has been done in this area.

CheckCell [6], for example, proposes *data debugging*, an approach to ensure input data correctness. It is a tool to detect data errors in spreadsheets based on the assumption that if one data cell has a big impact on the output compared to the other cells it is either rather important or wrong. CheckCell uses program analysis to identify output cells and their corresponding inputs. Statistical analysis is used to find out if an input value has a disproportionate influence on the output. The tool gives an impact score to each of the inputs and the user can choose what percentage of input values they want to check. The tool then shows to the user the found values one after another so that they can decide whether the value is wrong.

The approach solely uses the inputs and output of a function without considering the function itself. Input values are compared to each other to find faulty ones. Our approach on the other hand analyzes how the input values are used by the program and what assumptions are made about the input values. Because CheckCell is implemented as an add-in for Microsoft Excel, runtime errors are detected by Microsoft Excel and not the program itself. Our analysis reports values that will produce runtime errors that would otherwise not be detected before running the program.

4 Core Goals

- *Program Examples.* In a first step we will find and study examples of data analysis programs to gather common assumptions that programmers make about the input to their program. One source for code examples is Google Code Jam [7], an online coding competition. The contests consist of problems that require to write a program that takes some input, works with that data, and outputs a solution. The final programs of the contestants can be downloaded. In addition, we will collect and study code examples from other domains that can be found online. By manually inspecting the code we will find assumptions the programmer makes about the input values that will lead to a program failure when violated. (+)

- *Static Analysis Design.* The assumptions found in the first step need to be formalized to use them in an automated tool. We will build upon the Abstract Interpretations framework of Cousot [8] and design a static analysis that captures assumptions a program makes about the input data by inferring the preconditions. We will focus on finding assumptions that need to be fulfilled in order for the program not to yield an explicit error. This will be done as an over-approximation. The analysis finds necessary preconditions so that the execution will result in a program failure if a precondition is violated. We will focus on how the input values are used individually without considering their relationship to each other. For instance, the analysis will capture information about values that are outside of an expected range or values that are expected by the program but are missing in the input. (+++)

- *Input Checker Design.* We will design an approach that uses the information of the static analysis to automatically generate checks that expose input data values that violate the assumptions of a program. The analysis and the checks are run before the data analysis program would start. To be able to find and report an erroneous value, the checker needs to know about the structure of the input data. The information of the structure is implicitly given when the data is read by the program. After the static analysis, the checker iterates through the input data in a way that does not have to concur with the way the data is read by the program. That is why the static analysis needs to be designed in a way that captures enough information for the checker to match the input values with the assumptions about them in the data analysis program. (+++)

- *Tool Implementation.* The previously gained knowledge will be used to implement a tool using Python that targets data analysis programs written in Python. Before running the actual data analysis program, the tool infers assumptions of the program and automatically checks the input data to the program to discover unexpected values. This tool focuses on users that have no background in computer science. It should reduce the effort for end users to find and fix faulty values in the input data. Therefore we will design a meaningful approach to tell the user about possible issues of the data. This includes not overwhelming the user with unnecessary assumptions that are violated but rather gather these assumptions and create a useful method to show the faulty values to the user. For example in Listing 1 on line 8 the program prints the name of a town by accessing the array at the index given by the town identification input. The program will raise an exception if the array is accessed outside of its bounds. It is better to directly tell the user that the input value has to be in the range $[-4, 3]$ instead of first telling the user that the value has to be ≥ -4 and only later that the value also has to be ≤ 3. Furthermore, if there is only one value that is wrong, then it makes sense only to point out that particular entry, but if many values violate the same assumption, a better approach is to tell the user directly what assumption is made for all those data values. (++)

- *Tool Evaluation.* The tool can be evaluated on data analysis programs that use input data to run. The input data to a program can be manipulated to introduce unexpected values. We report the number of erroneous values found by our tool compared to those that are missed and the number of values overall. Another important aspect to evaluate is the quality of messages given to the user to fix the inputs. It is difficult to show this without doing an extensive user study. Users with no experience in computer science want simple to understand messages whereas more experienced users might want to see mathematically precise messages. We will therefore show the usability of our tool by presenting the efforts made to make the tool appealing to different kind of users (i.e. users with different levels of experience in computer science). (+)

5 Extensions

- *Further Assumptions.* The static analysis can be extended to furthermore gather information about values with an unexpected data type (e.g. a string instead of an integer), values that should be unique (e.g. input data that reports values for every year but should not contain multiple entries for one year) or values that have a certain format that matches a regular expression pattern (e.g. values that match ([01]?[0−9]|2[0−3]):[0−5][0−9], meaning values that are in 24-hour format). Furthermore, instead of focusing on individual data values, the analysis can be extended to capture relationships between them (e.g. one value is the sum of the values that follow on the same line). (+)
- *Implicit assumptions.* To catch more implicit assumptions, the analyzer can be extended. An example would be to check if certain input values are never used but would have been important for an analysis. In Listing 1 some of the ages are ignored because the number of people stated for a town is wrong. The program does not yield an explicit error but the problem could still be caught by an extended static analysis. In order to catch these assumptions they have to be formalized and the tool has to be extended. (++)
- *Improving Performance.* The first design will primarily focus on correctness. To further enhance usage of the tool, performance should be improved. Data analysis programs often work with large amounts of input data so that the checker might be slow when checking all the input values. In addition, the data analysis program might be large so that the analysis needs significant time to run. Extending the design by using multi-threading when checking the data or using a less complex analysis (e.g. simpler abstract domain) could improve performance. Another way is to narrow the search space by offering the user an interface to specify what kind of assumptions should be inferred or what part of the data should be checked. (++)
- *Under-Approximation.* The outcome of the static analysis we design will be an over-approximation of the correct result. An other approach is doing an under-approximation instead. The analysis finds the sufficient preconditions so that every error will be reported. The drawback is that the analysis might report correct values that do not yield an error when the program is running. Those values do not have to be fixed and the user has to decide whether a reported value would really result in program failure. To reduce the amount of correct values that should not be reported as problematic we can use dynamic analysis or testing. With a dynamic approach the tool can check if the data analysis program will indeed run into an erroneous state using the values reported by the static analysis. (+++)

References

[1] Joseph M Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

[2] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[3] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[4] Azadeh Farzan, Matthias Heizmann, Jochen Hoenicke, Zachary Kincaid, and Andreas Podelski. Automated program verification. In *International Conference on Language and Automata Theory and Applications*, pages 25–46. Springer, 2015.

[5] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[6] Daniel W. Barowy, Dimitar Gochev, and Emery D. Berger. Checkcell: Data debugging for spreadsheets. *SIGPLAN Not.*, 49(10):507–523, October 2014.

[7] Google code jam. `https://code.google.com/codejam/about`. Accessed: 2017-09-03.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.