Measuring Item Similarity in Introductory Programming

Radek Pelánek
Masaryk University
Brno, Czech Republic
pelanek@fi.muni.cz

Tomáš Effenberger
Masaryk University
Brno, Czech Republic
tomas.effenberger@mail.muni.cz

Matěj Vaněk
Masaryk University
Brno, Czech Republic
vanekmatej@mail.muni.cz

Vojtěch Sassmann
Masaryk University
Brno, Czech Republic
445320@mail.muni.cz

Dominik Gmiterko
Masaryk University
Brno, Czech Republic
ienze@mail.muni.cz

ABSTRACT

A personalized learning system needs a large pool of items for learners to solve. When working with a large pool of items, it is useful to measure the similarity of items. We outline a general approach to measuring the similarity of items and discuss specific measures for items used in introductory programming. Evaluation of quality of similarity measures is difficult. To this end, we propose an evaluation approach utilizing three levels of abstraction. We illustrate our approach to measuring similarity and provide evaluation using items from three diverse programming environments.

INTRODUCTION

A key part of learning is active solving of educational items (problems, questions, assignments). For high-quality education, we need large pools of items to solve. To use a large item pool efficiently, we need to be able to navigate it. For this, it is very useful to be able to measure the similarity of individual items. Similarity measures have many applications, particularly in adaptive learning systems. They may be used for automatic recommendations of activities (similarly to the use of similarity in product recommender systems [1]), for suggesting similar worked out examples that serve as hints [2], for improving learner and domain models, or for providing insight and feedback to teachers, content creators, and tool developers.

In this work we focus on the study of similarity of programming problems in the context of introductory programming, specifically for programming exercises in Python and programming problems with a robot on a grid, using a simplified graphical programming language as used for example in popular *Hour of code* activities. There are other domains with complex items, where measuring similarity of items may be useful, e.g., mathematics, physics, or chemistry. Introductory programming has the advantage that the item statements and solutions are more easily processed and learners' solutions are also readily available.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

L@S'18, London

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN .
DOI:

Our main contribution is the proposal of a systematic approach to studying similarity measures, which explicitly highlights many choices that we have to make to specify a similarity measure. This general approach to measuring and using similarity of educational items is outlined in Figure 1: based on the available data we compute similarity, which can be utilized in many ways. Several sources of data can be used for measuring similarity: 1. *an item statement*: specification of the item that a learner should solve, e.g., as a natural language description of the task or an input-output specification, 2. *item solutions*: in the case of programming a solution to an item is a program written in a given programming language; we can use a sample solution provided by the item author or solutions submitted by learners, 3. *data about learners' performance*: for example item solving times, number of attempts needed, or hints taken.

In analyzing and applying similarity it is useful to explicitly distinguish two matrices, which naturally occur in computations: a *feature matrix*, in which rows correspond to items and columns to features of items (e.g., keywords occurring in an item statement or an item solution), and *item similarity matrix*, which is a square matrix S, where S_{ij} denotes similarity of items i and j. Figure 1 shows typical steps in the computation and application of similarity. For each step there are many possible choices for their specific realization. For example, the *Arrow I* (computing a similarity matrix from a feature matrix) can be done using Euclidean distance, Pearson correlation coefficient, cosine similarity, and many other measures. Similarly, there are many specific ways how to transform an item solution into a feature matrix (*Arrow B*) and many algorithms for performing clustering (*Arrow H*). Moreover, individual steps are independent and can be combined.

A similarity measure for a particular application can thus be specified in many ways. To pick a specific measure, we need to compare and evaluate measures. We discuss methods for performing such evaluation, which utilize several levels of abstraction, and report on experience with several case stud-

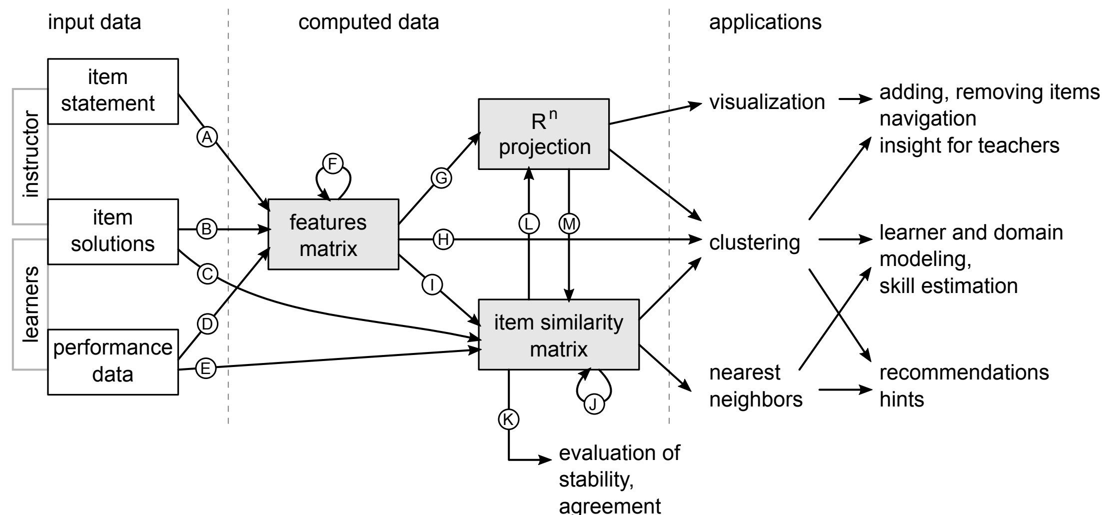

Figure 1. The general approach to computing and applying item similarity. The arrows that are discussed in more detail in the paper are denoted by letters and referenced in the text as *Arrow X*.

ies with different introductory programming environments (Python programming and two robot programming environments).

COMPUTING ITEM SIMILARITY

The standard item in introductory programming is to write a program implementing a specified functionality in a general-purpose programming language. Examples of such items are "Write a function that outputs divisors of a given number.", "Write a function that replaces every second letter in a string by X.", or "Write a function that outputs frequencies of letters in a given text.". An item statement in this context is given by a natural language description of the task, usually with some examples of input-output behavior. An item typically contains also a sample solution, which can be used for automatic evaluation of learners solutions.

A less standard, but very popular approach to teaching basic programming concepts is to use a simplified programming environment to program robots on a grid. Problems of this type are often used in the well-known *Hour of code* activity, with the participation of millions of learners. An item in this context is given by a specific world, which is typically a grid with a robot and some obstacles or enemies, and some restrictions placed on a program, e.g., there is often a limit on the number of commands, so that the problem has to be solved using loops instead of long list of simple commands. Solutions are written in a restricted programming language, which contains elementary commands for a robot (move forward, turn, shoot) and basic control flow commands. The program is often specified using graphical interface, a common choice today is the Blockly interface.

Processing the Input Data

As the first step, we need to get from the raw input data to the matrix form. To compute similarity based on an item statement, it is natural to go through the feature matrix (*Arrow A*). The choice of features depends on the specific type of programming environment. For the standard programming items where the item is specified using a natural language, the features have to be obtained from this text. A basic technique is to use the bag-of-words model, i.e., representing the text by a vector with the number of occurrences of each word (with standard processing, e.g., lemmatization and omission of stop words). Other features may be derived from the input-output specification, e.g., features describing types of variables (e.g., integer, string, list). In the case of robot programming, an item is typically specified by a robot world description and a set of available commands; basic features describing the world, e.g., size of the world, the presence of obstacles.

Another approach to measuring the similarity of items is to utilize solutions, i.e., programming codes that solve an item. The basic approach is to utilize a single solution – either the sample solution provided by the item author or the most common learner solution. Here we have two natural approaches to computing similarity: via feature matrix or by direct computation of similarities. With the *features approach* (*Arrow B*) we analyze the source code (typically using traversal of the abstract syntax tree) to compute features describing the occurrence of programming concepts and keywords like while, if, return, print, or use of operators. The basic approach is again the bag-of-words model, only applied to programming keywords instead of words in a natural language. In this representation we lose information about the structure of the program – we only retain the presence of concepts and the frequency of their occurrence.

The second approach is to directly compute item similarity matrix (*Arrow C*) by computing *edit distance* between the selected solutions for the two items. The edit distance can be computed in several ways, for example *tree edit distance* for the abstract syntax tree, potentially with some specific modifications for programs (in the specific programming language), or the basic *Levenshtein edit distance* for the canonized code, which is applicable particularly for the robot programming exercises, where programs can be relatively easily canonized.

Performance Data

Finally, we may use data on the performance of learners while solving items, e.g., the correctness of their solutions, the number of attempts, problem solving time, or hints taken. These data can be transformed into features (*Arrow D*) like average performance, the variance of performance, or the ratio of learners who successfully finish an item. Such features in most applications will not carry sufficiently diverse information to compute useful similarity between items, but these features may be useful as an addition to feature matrix based on an item statement or solution.

We can, however, compute similarity directly from the performance data (*Arrow E*): similarity of items i and j is based on the correlation of performance of learners on items i and j with respect to a specific performance measure (the correlation is computed over learners who solved both items i and j). This approach has been previously thoroughly evaluated in the case of binary (correctness) performance data [4]. In the case of programming problems, it is natural to use primarily problem solving times (rather than correctness).

Data Transformations

Once we compute the item features or basic item similarities, we can process them using a number of transformations. In contrast to the above-described processing of input data, which necessarily involves details specific for a particular type of items, the data transformation steps are rather general – they can be used for arbitrary feature matrices and are covered by general machine learning techniques. The feature matrix may be normalized or transformed (*Arrow F*) using techniques like *TF-IDF* (term frequency–inverse document frequency) transformation. From feature matrix or item similarity matrix we can compute a projection to R^n (*Arrow G* and *Arrow L*). Such a projection is typically used for applications, particularly for visualization of items. It can, however, also be a useful processing step in the computation of item similarities, e.g., for decorrelating features.

The computation of the similarity matrix based on the feature matrix (*Arrow I*) or its projection into R^n (*Arrow M*) is a common operation in machine learning, with many choices available, e.g., cosine similarity, Pearson correlation coefficient, and Euclidean distance (transformed into similarity measure by subtraction). These measures are used widely in recommender systems, with the experience that the choice of a suitable measure depends on a particular data set [1].

EVALUATION AND DISCUSSION

As the previous section shows, there is a wide number of techniques that can be used for computation of similarity. Moreover, individual steps can be combined in many ways. What is a good approach to computing similarity? Which decisions matter? These are hard questions since we cannot easily evaluate the quality of measures. Without going into details of a specific application, we cannot objectively compare measures. However, it is not reasonable to do the evaluation only with respect to the final application. Consider for example the use of similarity measures in the recommendation of items in an learning system. Evaluating the quality of recommendations is a complex task even if we compare just a single version of a recommendation algorithm to a control group. It is not feasible to evaluate variants of a recommendation algorithm for many similarity measures.

We thus need to analyze similarity measures even without considering specifics of a particular application. Such evaluation cannot give us verdicts about which measures are good or bad. But we can evaluate which decisions in the similarity computations are really important – which computation pipelines lead to different results. In this way, we can narrow the number of measures that need to be explored for a particular application from hundreds to few cases.

To this end, we propose to analyze similarity measures at several levels of abstraction:

1. Similarity of items for a specific similarity measure. This allows us to get the basic understanding of what kind of output we can obtain.
2. Agreement of different measures (across all items). This allows us to get understanding how much do the choices made in the computation matter.
3. Analysis of different agreement measures. To measure agreement between similarity measures we must also choose some method of quantification (*Arrow K*).

Case Studies

We have performed evaluation of similarity measures for several introductory programming environments: two Python programming settings (28 items used in the web system `tutor.fi.muni.cz` and 72 items used in a a large university programming class), and two two robot programming environments: *Robotanist* (74 items, also available at the web `tutor.fi.muni.cz`), which uses very simple programming commands (arrows for movement, colors for conditions), and *RoboMission* (`en.robomise.cz`, 75 items), which uses a richer world (e.g., meteorites, worm holes, colors, diamonds) and is programmed in the Blockly interface using a richer set of commands (movement, shooting, several types of conditions, repeat and while loops). Here we present examples of the analysis for the RoboMission environment and summarize the main results with the focus on the second level of analysis. Specific results that support the claims below are available in the full version of the paper [3].

To analyze agreement between two similarity measures, we first compute the item similarity matrix for of each of them and then compute the agreement as a correlation of values in these two similarity matrices. For a set of similarity measures, this gives us a matrix of agreement values, as illustrated in Figure 2. Similarity measures based on item statement vs. solution are only weakly related, as they focus on different aspects of items and their similarity. Measures that use the same source of data only in different ways are typically highly correlated. Figure 2 shows agreement between measures that use sample solutions. A low agreement is only between the binarized bag-of-words approach, which completely ignores the structure of the solution, and Levenshtein or Tree Edit Distance approaches, which are on the opposite spectrum of

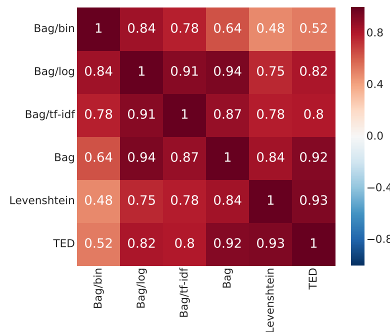

Figure 2. Agreement between similarity measures that use sample solutions (for problems in RoboMission), ordered from the most structure-ignoring approach on the top/left (bag-of-words, binarization transformation, Euclidean distance) to the most structure-based approach on the bottom/right (Tree Edit Distance).

the focus on the structure. Using a bag-of-words with log-counts (possibly with some feature weights normalization, such as IDF) is a reasonable compromise. The effect of the choice of a function for computing similarity from feature matrix (*Arrow I*) seems to be small. For measures that combine multiple sources of data, feature normalization is important for adjusting different scales and avoiding one source of data being far more important than the other.

Instead of using the sample solution provided by the instructor, we can use learners solutions, e.g., the top most common learners' solutions or an average of several top solutions. We evaluated this approach for Python programming. The measures based on sample solution and top learner solution have high overall correlation, but for individual items they may differ quite significantly – in some cases the sample solution and top learner solution may be very different as even simple programming problems like computing factorial can be written in widely different ways: using for loop, while loop, or recursion. When data on learner solutions are available, using the average of the 3 most common solutions seems to be a robust approach.

For the Robotanist and Python programming, we also explored item similarity based on learners' performance (problem solving times). The correlation between measures based on solution and measures based on performance is between 0.2 and 0.4. This is a weak agreement, but it turns out that with the size of our data (in the order of hundreds of solutions per item) the measures based on performance are not yet stable. Thus even this weak agreement may indicate a relationship between these different approaches to measuring similarity. This relationship needs to be explored for larger data sets.

So far we used simple correlation to measure agreement, i.e., to quantify agreement of two similarity matrices we flatten them into vectors and then compute Pearson's correlation coefficient over these vectors. But there are other choices to measuring agreement. Particularly in applications, where the intended use of similarity measures is to pick the closest neighbors (e.g., a recommendation of similar exercises), it may be more relevant to measure the agreement of rankings or agreement on top N positions. To evaluate the impact of this choice it is useful to perform analysis on a higher level of abstraction – comparing results obtained using different measures of agreement. We have compared the basic correlation agreement with agreement on the top N positions. Although these are quite different approaches to measuring agreement of measures, they lead to very similar results (see full version [3] for details), i.e., it is not a fundamental decision which one we choose to use for evaluation.

Basic Recommendations for Similarity Computation

The suitable choice of a similarity measure depends, of course, on a particular setting (programming environment, characteristics of available input data) and the particular application of the measure. However, it is useful to have a basic default choice which can serve as a baseline, which can be further improved.

Based on our explorations, we propose to use the following steps to compute such a default similarity measure: 1. Use item solutions as input data. 2. Compute feature matrix based on the data using the basic bag-of-words approach – computing number of occurrences of natural programming keywords. 3. Normalize the feature matrix, specifically using some variant of the TF-IDF transformation. 4. Compute item similarity based on the feature matrix using the Euclidean distance of vectors in the normalized feature matrix.

The aspect that can most importantly change the results of the computation is the first step – the choice of input data. We believe that item solutions are a good default because for introductory programming problems they are basically always available and the basic bag-of-word analysis can be easily performed in different settings. For performance data to be useful, it is necessary to collect large data on learner behavior, which limits their applicability. The form of item statement data can depend on a particular application (e.g., it differs significantly between our robot programming problems and Python programming problems), which makes its use more application specific.

REFERENCES

1. Christian Desrosiers and George Karypis. 2011. A comprehensive survey of neighborhood-based recommendation methods. In *Recommender systems handbook*. Springer, 107–144.

2. Roya Hosseini and Peter Brusilovsky. 2017. A study of concept-based similarity approaches for recommending program examples. *New Review of Hypermedia and Multimedia* (2017), 1–28.

3. Radek Pelánek, Tomáš Effenberger, Matěj Vaněk, Vojtěch Sassmann, and Dominik Gmiterko. 2018. Measuring Item Similarity in Introductory Programming: Python and Robot Programming Case Studies. (2018). `https://www.fi.muni.cz/adaptivelearning/sim-full.pdf` Full version of the paper.

4. Jiří Řihák and Radek Pelánek. 2017. Measuring Similarity of Educational Items Using Data on Learners' Performance. In *Educational Data Mining*. 16–23.