Towards automatic recognition of irregular, short-open answers in *Fill-in-the-blank tests*

Método para el reconocimiento automático de respuestas abiertas, cortas e irregulares en cuestionarios de completar espacios

Sergio A. Rojas
Systems Engineer, Ph.D. in Computer Science, Assistant Professor at the Universidad Distrital Francisco José de Caldas. Bogotá, Colombia.
Contact: *srojas@udistrital.edu.co*

Received: August 21, 2012 **Classification: Research)**
Accepted: May 21, 2013 **Funding: Universidad Distrital Francisco José de Caldas**

***Keywords*:** automatic learning assessment, pattern recognition, text analysis.

***Palabras clave*:** evaluación automática de aprendizaje, reconocimiento de patrones, análisis de texto.

ABSTRACT

Assessment of student knowledge in Learning Management Systems such as Moodle is mostly conducted using close-ended questions (e.g. multiple-choice) whose answers are straightforward to grade without human intervention. FILL-IN-THE-BLANK tests are usually more challenging since they require test-takers to recall concepts and associations not available in the statement of the question itself (no choices or hints are given). Automatic assessment of the latter currently requires the test-taker to give a verbatim answer, that is, free of spelling or typographical mistakes. In this paper, we consider an adapted version of a classical text-matching algorithm that may prevent wrong grading in automatic assessment of FILL-IN-THE-BLANK questions whenever irregular (similar but not exact) answers occur due to such types of error. The technique was tested in two scenarios. In the first scenario, misspelled single-word answers to an Internet security questionnaire were correctly recognized within a two letter editing tolerance (achieving 99% accuracy). The second scenario involved short-open answers to computer programming quizzes (i.e. small blocks of code) requiring a structure that conforms to the syntactic rules of the programming language. Twenty-one real-world answers written up by students, taking a computer programming course, were assessed by the method. This assessment addressed the lack of precision in terms of programmer-style artifacts (such as unfamiliar variable or function nomenclature) and uses an admissible tolerance of up to 20% letter-level typos. These scores were satisfactory corroborated by a human expert. Additional findings and potential enhancements to the technique are also discussed.

RESUMEN

La evaluación de estudiantes en ambientes educativos virtuales como *Moodle*, se realiza mediante preguntas cerradas (por ejemplo, de selección múltiple) cuyas respuestas pueden ser calificadas automáticamente sin necesidad de intervención humana. Los cuestionarios de preguntas abiertas con espacios para rellenar permiten al estudiante escribir variantes que pueden diferir de la respuesta esperada debido a errores de ortografía o tipográficos, siendo inviable calificarlas automáticamente mediante comparación exacta. En este artículo se propone una técnica que utiliza una versión adaptada de un algoritmo de comparación de cadenas de caracteres, y que podría realizar la evaluación automática de este tipo de preguntas. Se presentan resultados en 2 escenarios. En el primero, se calificaron respuestas irregulares de una sola palabra en un cuestionario de seguridad en internet; las variantes fueron reconocidas correctamente con un 99% de exactitud y nivel de tolerancia de corrección de máximo dos letras. En el segundo, se evaluaron respuestas abiertas cortas a cuestionarios de programación de computadores, es decir, pequeños bloques de código de lenguaje de computador. Se calificaron 21 respuestas escritas por estudiantes de un curso real de programación, sin considerar particularidades como nomenclatura para variables o funciones, y utilizando una tolerancia de corrección de hasta 20% de la longitud de la respuesta libre. El método reconoció 12 respuestas como probablemente correctas con expresiones mal formadas o incompletas pero fácilmente subsanables. Esta calificación fue satisfactoriamente corroborada por un experto humano. También se discuten otros resultados, así como posibles mejoras de la técnica propuesta.

* * *

1. INTRODUCTION

Learning Management Systems (LMS) are widely-used nowadays to support students during their learning process in medium and high education, e.g. the *Moodle* platform (http://moodle.org, last visit: Aug 13, 2012). One the key components of LMS for course instructors and mentors are supporting tools for assessment of student knowledge. Such examination can be conducted either by means of close- or open-ended questions. Close-ended questions provide the test-taker with a problem formulation and a list of options where just one has to be selected (multiple choice, yes/no, true/false, matching terms) [1]. This type of questions is aimed at obtaining facts, usually by exercising the recollection skills of the respondent; these are the lower levels in the knowledge taxonomy [2]. Automatic grading of the test is straightforward to perform, and most LMS allow executing this task efficiently: the test-maker provides the correct set of options beforehand and the system records and verifies the answers chosen by the test-taker.

One of the disadvantages of close-ended questions is that in order to be effective, they require a careful design of the statement/options configuration so as to truthfully evaluate the level of comprehension that is expected from the respondent. Otherwise this type of test might be assessing the ability of the student to discard and select a default answer rather than their analytical and associative thinking skills. Another common criticism of this category of questions is that it is not clear if the student would have been able to come up with the answer had this answer not have been given in advance within the list of options.

In the opposite corner tests based on open-ended questions are found. These are questions where the examiner provides only a statement that the test-taker must respond freely. They are useful to obtain a more elaborated answer reflecting the views, feelings and opinions of the respondent, that is, the higher levels of the knowledge taxonomy [2]. The inquirer should worry in this case to propose incisive questions that challenge the creativity of the test-taker. Automatic grading in this type of question is more difficult, since the answers cannot be anticipated in advance, and hence a grammatical, syntactic and semantic analysis must be applied over the entered response. Usually their assessment requires some level of human intervention. This is nevertheless a hot topic of research, and a lot of effort has been made from the Natural Language Processing and Information Retrieval community (see [3] and references within).

In this paper we consider *fill-in-the-blank* questions, a sort of middle point between those two extremes of the testing spectrum. Here the examiner defines a question statement where several tokens or elements are missing and no choices are given (e.g. "*Artificial* ___________ *is the branch of* ____________ *science that studies how intelligence may be created in* ____________"). The test-taker must fill-in the gaps exercising his/her analytical, associative and recalling skills in order to solve correctly the questions or to complete the sentence to make sense. One may label this type of question semi-open since there are multiple combinations of words that can be used freely by the respondent. An automatic scoring system will take the filled-in-blank text and compare it to the correct solution given in advance by the examiner, i.e. a list of feasible words for each blank space. Notice that the expected text in each gap may comprise not only one, but a few number of words which is why they can also be regarded as *short-free* answers.

The relative freedom given to the respondent in this category of questions makes room for the occurrence of orthographical or typographical mistakes making them hard to recognize with simple text-matching procedures. Irregular answers arising due to substitution, insertion or elimination of characters would miss the correct credit because it would be unfeasible to match them against an exhaustive list of all possible erratic variants of the expected answer.

On the other hand, orthographic or typographic conformity is not usually the main goal of student assessment. To illustrate this point, automatic text-matching assessment of the last gap of the above question will give correct grade to answers reading exactly the word "computers", "machines", "robots" or "automata", but will miss credit for erratic answers such as "compters", "mashines", "robotz", "uatomata" or "automatons" which nonetheless might have been given correct or partial credit by a human examiner.

The challenge of automatic assessment of fill-in-the-blanks or short free-text responses has been approached with a number of techniques related to Latent Semantic Analysis [4] - [5] and Natural Language Preprocessing [6], including stemming (finding the linguistic root of a word), punctuation removal, dictionary spelling correction and statistical analysis of text corpuses. Our contribution in this paper is a much simpler yet effective method to compute a degree of mismatch of irregular answers to their exact expected versions. In this way an automatic examiner would be able to assign total or partial credit instead of a crisp accept/reject outcome for a given answer.

The method uses a variation of a classical string-to-string approximate matching algorithm that is able to recognize similar variants of an expected text due to edit operations. Preliminary results of this method have been previously reported [7]. In this paper we extend the study to give further details of its operation, elaborate on its potential applications in LMS platforms and courseware testing systems, and account on new empirical findings on automatic scoring of tests conducted over internet security and computer programming domains.

2. METHOD

The inspiration of the method is illustrated by the following arbitrary question from a hypothetical telecommunication networks test, concerning a software tool commonly used by network administrators:

"Wireshark is a well-known __________ tool suited for traffic analysis in LAN and WAN networks".

The correct answer would be "*sniffer*", a computer program that enables the administrator to run such analysis. A few variants of this word due to orthographical and typographical mistakes are listed below, some of which are particularly more likely to occur if the respondent is not a native English speaker:

Sniffer	esnniffer
sNiFfEr	S.N.I.F.F.E.R
Snifer	smiffer
Snnifer	zniffer
Asniffer	snlffer

It would have been unfeasible for a test-maker to foresee this list of variations in order to supply them to an exact-match algorithm that automatically grades correctly all-possible misspelled answers, among other things because the number of variants can increase exponentially in size. The goal of our method is to recognize as much as possible variants produced by admissible substitutions, insertions or eliminations of characters in a given expected correct answer (a string of characters representing a word or a term or a short sentence). The output of the method ought to be a score for the student answer indicating a perfect match or a degree of similarity expressed as the number of corrections required to transform an erratic variation into a correct answer.

The core of the method is the classic string-to-string matching algorithm due to Wagner and Fischer [8] widely-used originally in text processing, computational linguistics, and afterwards in computational biology for genome sequence alignment. The purpose of the algorithm is to compute the number of edits (corrections) necessary to transform the first string into the second. The edit operations allowed to convey such transformation are insertion, elimination and substitution. This algorithm (see figure 1) is explained below.

The two input strings **s** and **v**, are *n* and *m* characters in length respectively. The algorithm requires also the definition of an edit cost function, *editcost*(s_i, v_j), that defines the (dis)similarity between the characters s_i and v_j., that is, the cost of editing the first character to convert it into the second character. We will discuss the details of such function later on in the text. The algorithm maintains a distance matrix *D* such that any entry $D(i+1,j+1)$ holds the number of edits needed to transform the prefix substring $\mathbf{s}_{1:i}$ into the prefix substring $\mathbf{v}_{1:j}$. For example, the loop tagged as the first step in Figure 1, computes the edits required to blank-out the substrings of the input string **s**, starting from its first character (that is, it counts the number of eliminations). The loop on the second step similarly, computes the number of insertions required to obtain all the substrings of the input string **v** out of the empty string Ø.

The nested loops of the third step are the core of Wagner and Fischer's idea. They progressively fill the remainder entries of the *D* with a dynamic programming procedure that reuses previously calculated shorter substring distances,

```
Algorithm 1. stringdist(s, v)
Input: canonical string s=($s_1$,…, $s_n$), variant string v=($v_1$,…, $v_m$)
0. Initialize D: a zeroed matrix with ($n+1$) rows and ($m+1$) columns
1. Repeat for $i=1,…,n$
        $D(i+1,1)=D(i,1)+editcost(s_i, Ø)$
2. Repeat for $j=1,…,m$
        $D(1,j+1)=D(1,j)+editcost(Ø, v_j)$
3. Repeat for $i=1,…,n$
    Repeat for $j=1,…,m$
        $D(i+1,j+1)=min(D(i,j+1)+editcost(s_i, Ø), D(i+1,j)+editcost(Ø, v_j), D(i,j)+editcost(s_i, v_j))$
Output: edit distance in $D(n+1,m+1)$
```

Figure 1. String-to-String distance algorithm to compute the edit distance between character strings **s** and **v** using the functions editcost() which returns the cost of editing one character into another, and min() which chooses the minimum between three distances

Note: The null (empty) character is denoted Ø.

Source: Adapted from [8].

up to the full extent of the original input strings. Such entries are obtained as the minimum between three distances, namely the distance between $\mathbf{s}_{1:i-1}$ and $\mathbf{v}_{1:j}$ plus the cost of deletion of letter s_i from **s**, the distance between $\mathbf{s}_{1:i}$ and $\mathbf{v}_{1:j-1}$ plus the cost of insertion of symbol v_j into **s**, and lastly, the distance between $\mathbf{s}_{1:i-1}$ and $\mathbf{v}_{1:j-1}$ plus the cost of substitution of s_i by v_j in **s**. As a result, the value computed in $D(n+1,m+1)$ holds the edit distance (i.e. number of corrections) required to transform **s** into **v**. A zero or low value in this output would indicate that the two strings can be regarded as equivalent.

From the description of the algorithm, it is clear that the function *editcost*(s_i, v_j) is pivotal in the correct matching of the two strings. Thus we have carefully defined this function as it is shown in figure 2. The function was designed with two aims. On the one hand, the function ignores (assign zero cost to) replacement of a normal character with a comparable non-alphabetical symbol (for example, an ‘s’ can be substituted by an ‘z’, an ‘a’ with an ‘á’. an ‘n’ with an ‘ñ’ and so on); likewise the insertion of a punctuation character is ignored (symbols such as commas, semi-colons, question marks). That is to say, no corrections are accounted in these cases. On the other hand, substitutions with non-comparable characters, insertions of characters other than punctuation marks as well as deletions are treated as modifications incurring a cost of one correction.

```
Algorithm 2. editcost($s_i$, $v_j$)
0. Set d to zero
1. If $v_j$ is empty
        d=1        /* Cost of elimination */
2. Else if $s_i$ is empty and $v_j$ is not a punctuation symbol
        d=1        /* Cost of insertion */
3. Else if $s_i$ is empty and $v_j$ is a punctuation symbol
        d=0        /* Cost of insertion of separators */
4. Else if $v_j$ is interchangeable with $s_i$
        d=0        /* Cost of admissible substitution */
5. Else
        d=1        /* Cost of other substitutions */
Output: edit cost in d
```

Figure 2. The proposed edit cost function

Source: own work.

Now that the technique has been described, we shall proceed reporting on two experiments conducted to validate its feasibility.

3. EXPERIMENTS AND RESULTS

We carried out two experiments in order to exemplify the potential application of the proposed method. The first experiment was aimed at assessing a simple question from an internet security questionnaire. The second experiment was designed with a broader scope of assessing skills for writing small pieces of code in a computer programming language. All algorithms and experimental test-bed were implemented in Octave 3.2.4. Datasets are available on request.

3.1 Assessing irregular answers in a fill-in-the-gap questionnaire

This experiment embraced testing the feasibility of the method to grade hypothetical answers to the following single-question questionnaire related to email security and privacy:

"Most emails with the word ____________ in its subject line are spam targeted to potential users of the pharmaceutical drug commercialised by Pfizer".

We collected a synthetic list of 186 possible variants of the correct answer "*Viagra*" gathered from internet forums. The distribution of edit distances from the variants in the list to the correct answer is showed in figure 3. It can be seen in these results that the method recognised 80 % of the answers with a perfect match, that is, 150 variants were considered equivalent to the original term (no corrections were accounted). The remainder 36 irregular answers scoring a distance greater than zero contain heavy misspelling mistakes or bogus insertion of letters, some of them very

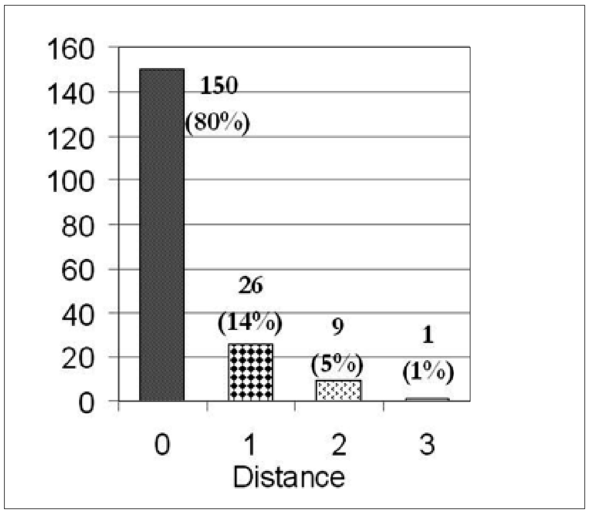

Figure 3. Edit cost distribution of irregular answers to the "Viagra" test

Source: own work.

unlikely to occur in real-life (e.g. "Viagorea", "Viargvra", "ViaTagra", "ViagDrHa", "ViaVErga", "V?l?a?g?r?aÿ"). An excerpt of the synthetic answer list with the scores computed for each variant is shown in table 1.

In order to extend the ability to recognise erratic but semantically well-intended answers, the automatic evaluation may assign full credit not only on a perfect-match basis, but also on compliance with a specified distance threshold. For example, in this experiment a distance threshold of 2 would have accepted as correct 99 % of these simulated answers. Such a threshold would establish a relaxed margin on irregularities or misspellings for correctness approval. In the previous example the irregularity margin of 2 corrections imply an editing rate of less than 33 % of total length of the correct answer. The possibility of assigning correct (or partial) credit to irregular answers with low corruption levels stands up as the main advantage of this technique, compared to a naïve exact-matching automatic scoring taking crisp accept/reject decisions on a rigorous matching basis only.

Table 1. An excerpt of 80 answers from the “Viagra” dataset (edit cost in brackets)

ViaVErga (3)	Vi$agra (1)	V&iagra (1)	vi-@gr@ (0)	viag*ra (0)
Viagorea (2)	Viag&ra (1)	via-gra (0)	via---gra (0)	viagr*a (0)
Viargvra (2)	ViagrYa (1)	V-I-A-G-R-A (0)	via_gra (0)	viagra* (0)
ViaTagra (2)	VIA7GRA (1)	Vi/agra (0)	vi(@)gr@ (0)	*v*iagra (0)
ViagDrHa (2)	Viag&ra (1)	V?1?@?G’?Ra (0)	via.gra (0)	*vi*agra (0)
ViaJ1gra (2)	Viag%ra (1)	Vi-ag.ra (0)	Viaggra (0)	*via*gra (0)
V?I?a?g?r?aÿ (2)	Viagzra (1)	vi**agra (0)	V1@grA (0)	*viag*ra (0)
Viarga (2)	Vigra (1)	Vii-agra (0)	Viag)ra (0)	*viagr*a (0)
VIxAGRAÿ (2)	Viag$ra (1)	V/i/a/g/r/a (0)	V?iagr?a (0)	*viagra* (0)
Vi gr (2)	VyAGRA (1)	Viagr(a (0)	v-ii-a=g-ra (0)	v*i*agra (0)
viagrgaÿ (2)	viagdra (1)	vi@g*r@ (0)	vi@gr|@| (0)	v*ia*gra (0)
ViaZUgra (2)	ViagWra (1)	vi@gr*@ (0)	vi@|g|r@ (0)	v*iag*ra (0)
ViaaPrga (2)	Vkiagra (1)	Viagr^a (0)	*viagra (0)	v*iagr*a (0)
Viaoygra (2)	Vi.ag.raÿ (1)	V-i.a-g*r-a (0)	v*iagra (0)	v*iagra* (0)
V?I?A?G?R?A (1)	Viagvra (1)	via---gra (0)	vi*agra (0)	vi*a*gra (0)
Viag@ra (1)	Viagara (1)	V|i|a|g|r|a (0)	via*gra (0)	vi*ag*ra (0)

Source: own work.

3.2 Assessing writing of blocks of code in a computer programming language

In this second experiment we intended to broaden the application of the method to automatic grading of larger semi open-ended questions within language domains were strict syntax defines a rigid structure for the possible answers. A clear example of this kind of domain is computer programming languages. Syntax in these languages is extremely stringent as its purpose is to communicate commands to a computer in a precise way. If a sentence fails to comply with the syntax it would be rejected by the compiler that transcribes into machine code the piece of code written in programming language.

Within this context a short-free answer (i.e. a small block of code) ought to exhibit an explicit structure that is anticipated to conform to both the syntactic rules of the language and the algorithmic patterns given by the course instructor. From this point of view, the proposed automatic assessment method can be used to compare the fragments of code authored by the student and the expected/template solution devised by the instructor. The idea in this regard is to obtain a similarity score of these two strings indicating a correction rate for the code written by the student. This score can be used as a hint to accept/reject the answer or as an indication of the degree of correctness of the block of code. Although this would be by no means an absolute measure of correctness, it might suggest which answers deserve or not further visual inspection in order to assign a definite marking.

More to the point one may argue that the main goal of assessing computer programming knowledge is not perfect compliance to the language syntax but rather to corroborate the achievement of technical and analytical skills such as abstraction, implementation of algorithmic structures and allocation of responsibilities, among others. On the other hand, being able to recall keywords and adhering strictly to the syntactic constructs of the language is a secondary goal that can always be validated with the error report generated by the compiler on the fragments of code. However, in a programming course with a large number of students, compiling each of the many fragments of code submitted in an LMS quiz may prove cumbersome and sometimes misleading since for example, a lack of a semicolon in a sentence (the character ';') would generate a failed compilation report, even if the rest of the code is correct. The motivation of using the method described earlier instead of a compiler to score a block of code is to account for the possibility of assigning partial (or full) credit to the student even if incomplete or malformed but still easily correctable expressions were written in his/her code. The latter will be closer to the verdict that a human evaluator would attain by common sense.

The experiment was conducted as follows. A quiz requiring the solution of two questions in the *Java* language (*Java* is a registered trademark of Oracle Corporation) was devised and uploaded online in a test management system (googledocs in the following address: http://tinyurl.com/JavaQuiz2). The fragments of code given as answers were collected from twenty-one students of an Object Oriented Programming (OOP) course offered at the School of Engineering of Universidad Distrital in the first term of 2012. The automatic assessment of each answer was carried out by running Algorithm 1 and 2 over two input strings: a canonical block of code representing a template solution provided by the instructor and each of the answers collected from the students.

We remark that no changes were made in the algorithms comprising the automatic assessment method; we keep them as tested in the previous experiment. Thus, because of the mechanics discussed earlier, characters such as whitespaces and line feed are ignored as separators. Furthermore, in relation to punctuation characters, the method ignores correctable mistakes such as replacing a comma for a semicolon (',' ↔ ';') or duplication of a semicolon (';;;'). In the same way, the method does not validate the lack of closing brackets or substitution of brackets by parenthesis or vice-versa. Of course, adjustments to the algorithms, in particular the *editcost()* function, with respect to the lists of admissible substitutions and punctuation marks can be made if a more rigorous compile-style evaluation is required.

Arguably a solution to a programming problem can be written correctly in multiple ways due to the usage of different but equivalent commands, algorithmic structures or operators. As a result, fragments of code of longer or shorter length can be conceived as correct answers. For this reason, in order to account for variability in the answers given by the students, the instructor was asked to provide up to six possible template solutions for each question (see table 2). The automatic scoring was carried out by matching each answer with every template solution using the proposed method and then recording the smallest similarity score (the minimum edit cost obtained across all template solutions).

Assuming that each student may come up with different solutions varying in length (number of characters) we proceeded to normalize the raw scores to obtain a length-independent score as the ratio of the edit distance with respect to the total length of the answer. This normalized score can also be regarded as a correction rate, or in other words, the proportion of changes needed to transform the student answer to its best-matching template solution. Thus the smaller this score, the closer the block of code reflects one of the instructor expected solutions. Such proportion may suggest an examiner a hint of which answers can be automatically accepted, and which should be rejected, or deserve further human inspection. If the examiner sets a threshold on the correction rate, say a percentage τ, hence answers scoring $\tilde{d} \leq \tau$ would be accepted as correct, with $\tilde{d}$ being the

Table 2. Questions and template solutions for the OOP quiz experiment

Quiz questions	*Q1. Declare a class of objects to represent a 3D point with integer coordinates (named x, y, z). Add a parametric constructor (with parameters x, y, z).*	*Q2. Write a method (called "compare()") with a string input parameter (named "text") that verifies if it matches the word "Java", and outputs true or false as a result.*

Template answer (Version 1)	public class Punto3D{
	private int x;
	private int y;
	private int z;
	public Punto3D(int x,int y,int z)
	{ this.x=x; this.y=y; this.z=z; } }
		public boolean comparar(String texto){
		if(texto.equals("Java"))
		return true;
		else
		return false; }

Template answer (Version 2)	public class Punto3D{
	int x; int y; int z;
	public Punto3D(int x,int y,int z)
	{ this.x=x; this.y=y; this.z=z; } }
		public boolean comparar(String texto){
		return texto.equals("Java"); }

Template answer (Version 3)	class Punto3D{
	private int x;
	private int y;
	private int z;
	public Punto3D(int xp,int yp,int zp)
	{ x=xp; y=yp; z=zp; } }
		public boolean comparar(String texto){
		boolean respuesta;
		if(texto.equals("Java"))
		respuesta=true;
		else
		respuesta=false;
		return respuesta; }

Template answer (Version 4)	class Punto3D{
	int x; int y; int z;
	public Punto3D(int xp,int yp,int zp)
	{ x=xp; y=yp; z=zp; } }
		public boolean comparar(String texto){
		boolean respuesta=false;
		if(texto.equals("Java"))
		respuesta=true;
		return respuesta; }

Template answer (Version 5)	class Punto3D{
	private int x,y,z;
	public Punto3D(int xp,int yp,int zp)
	{ x=xp; y=yp; z=zp; } }
		N/A

Template answer (Version 6)	class Punto3D{
	private int x,y,z;
	public Punto3D(int x, int y, int z)
	{ this.x=x; this.y=y; this.z=z; } }
		N/A

Source: own work.

said normalized score. We will refer to $\tilde{d}$ simply as the "score" in the following.

We turn now to the results of the experiment that are summarised in table 3. Here answers submitted by students S1…S21 to questions 1 and 2 (Q1 and Q2 from table 2) are organized as one row per student. Scores computed as described above are shown in front of each answer. Firstly let us examine the results obtained for Q1 and confront them to the judgment made by a human expert. Let us assume that a correctness threshold $\tau = 20\%$ was defined. Therefore 8 out of 21 answers would have been marked as "correct" by the method (S1Q1, S2Q1, S5Q1, S8Q1, S15Q1, S16Q1, S17Q1 and S19Q1). Visual inspection by the human expert confirmed the correctness of these answers. Now we examine the remainder answers which are thus candidates to be rejected. Two answers on the verge of this threshold (S13Q1, S18Q1) were actually judged as correct by the expert, diverging from the template solutions only in the names used for class variables and parameter (‘**a**’ instead of ‘**x**’, ‘**b**’ for ‘**y**’, etc.). In contrast, answers with scores ranging from 30 % to 40 % exhibit fundamental flaws. Take for example answer S4Q1 with incoherent commands in the constructor block; S7Q1 similarly declares an inappropriate command in the constructor; S20Q1 defines a “default” constructor not the “parametric” constructor that was asked. Lastly, answers scoring higher than ≥45 % (S10Q1, S11Q1, S12Q1 and S14Q1) did not comply at all with the algorithmic structure that was inquired in the question. As for the expert’s opinion the last two sets of answers have to be clearly rejected as incorrect.

Table 3. Results of the automatic assessment of the OOP quiz. Scores are proportions of dissimilarity to best-matching template solution. The rightmost column displays scores of answers to Q2 with a rectified nomenclature (not shown)

Student ID	Answers to question 1 (Q1)	Score	Answers to question 2 (Q2)	Score	Score (mod.)

S1	public class Punto3D {
	private int x;private inty;private int z;
	public punto3D(int x, int y, int z) {this.x=x;this.y=y;this.z=z; } }
		1 %	public boolean cadena(String texto){
			if(texto == ‘Java’)
			{return true; }
			else {return false; } }
				14 %	8 %

S2	public class 3D{
	int x=0, y=0, z=0;
	public class 3D(int x, int y, int z)
	{ this.x=x; this.y=y; this.z=z; } }
		17 %	public boolean verificacion (String palabra){
			boolean encontro=false;
			if(palabra==“Java”){encontro=true;}
			return encontro; }
				45 %	7 %

S3	public punto(){
	pirvate x:pirvate y;private z;
	public punto3D(int nuevaX, int nuevaY, int nuevaZ)
	{nuevoX=x;nuevoY=y;nuevoZ=z;}
		41 %	public boolean verificar(String palabra){
			if(palabra.equals(“JAVA”))
			{return true;}
				39 %	21 %

S4	public class Punto {
	private double x;private double y;private double z;
	public void darPunto(){ x = new “”””; y= new “”””; z= new”””””;} }
		39 %	public boolean verificar(String palabra){
			if (palabra==“Java”){ return true;}
			else{return false;} }
				23 %	8 %

S5	public class Punto(){
	int x;int y;int z;
	public Punto(int x1,int y1, int z1)
	{x=x1;y=y1;z=z1;} }
		19 %	public boolean texto(String frase){
			try {if frase == “Java” return try;}
			elsereturn false;}
				29 %	15 %

S6	public class 3d{
	int x, y z;
	public 3d (int a. int b, int c)
	{ x=a; y=b; z=c;}
		42 %	public boolean palabraJava(String p){
			if (p.equals (‘Java’) ? true: false)return p }
				38 %	15 %

S7	Public class Punto(){
	int x; int y int z;
	public Punto(){}
	JoptionPane.showMessageDialog(“”el punto esta ubicado en la cordenadas: “”
	+ “”(“” + x+ “”,”” + y + “”,”” + z + “”)””);}
		33 %	public boolean java(String palabra){
			if (palabra.equals(“Java”){ return true;}
			else return false: }
				18 %	1 %

Student ID	Answers to question 1 (Q1)	Score	Answers to question 2 (Q2)	Score	Score (mod.)

S8	public class Punto3D{
	private int x;private inr y;private int z;
	public Punto3D(int elX, int elY, int elZ)
	{x=elX;y=elY;z=elZ;} }
		19 %	public Boolean darjava(String texto){
			boolean a=false;
			if(texto.equalsignorecase(“java”))
			{ a= true; return a} }
				33 %	27 %

S9	public class 3D{
	private int x;private int y;private int z;
	public 3D(int puntox,int puntoy,int pun-toz)
	{x=puntox;y=puntoy;z=puntoz;}
		31 %	public boolean cadena(String cadena){
			if.equals(Java)return true; }
				40 %	30 %

S10	private int x,y,z;
	public void 3D() {x= x;y=y;z=z;}
	public
		68 %	public boolean cadena(String palabra){
			if (palabra.equals(“Java”))
			{ return true; } }
				37 %	19 %

S11	public void objeto (int x, int y, int z)
	{this.x=x;this.y =y;this. z=z;}
		44 %	public boolean comparacion (String fra-se){
			boolean p = false;
			String palabra = “java”;
			if (frase.equals (palabra){return true;}
			return p; {
				38 %	29 %

S12	public int 3d (int a, int b, int c)
	{int x= a,int y= b;int z = c;
		54 %	import javax.swing.JOptionPane;
			public class mundo {
			public static void main(String[] args) {
			boolean Comprobacion = false;
			String x;
			x=JOptionPane.
			showInputDialog(“Texto”);
			if(“Java”.equals(x)){
			Comprobacion = true;
			System.out.println(“Es Java”);} } }
				53 %	53 %

S13	public class Punto3D{
	private int a,b,c;
	public Punto3D(int x,int y,int z)
	{a=x;b=y;c=z;} }
		22 %	public boolean comprobar(String pala-bra){
			if(palabra=“Java”)return true;
			elsereturn false; }
				21 %	12 %

S14	public void 3d(int x, int y, int z)
	{this.x=x;this.y=y;this.z=z;}
		47 %	public boolean revisar (String cadena){
			String c1=“Java”;
			boolean resultado=false;
			if(c1.equals(cadena)){resultado=true;}
			return resultado; }
				42 %	28 %

S15	public class Punto3D{
	private int x; private int y;private int z;
	public Punto3D(int a, int b, int c)
	{ x=a; y=b; z=c; } }
		16 %	public boolean esJava(String palabra){
			return palabra.equals(“Java”) ?
			true:false; }
				29 %	12 %

S16	public class Punto3D{
	private int X; private int Y;private int Z;
	public Punto3D(int x, int y, int z){
	this.X = x; this.X = y; this.X = z;} }
		3 %	public boolean verificarPalabra(String palabra){
			return (palabra == “Java”) ? true : fal-se; }
				35 %	13 %

S17	public class Punto{
	private int X;private int Y;private int Z;
	public Punto(int x, int y, int z)
	{ X = x; Y = y; Z = z;} }
		15 %	public boolean verificar(String palabra){
			if(palabra.equals(“Java”)) return true;
			else return false; }
				18 %	1 %

Student ID	Answers to question 1 (Q1)	Score	Answers to question 2 (Q2)	Score	Score (mod.)
S18	public class punto3D(){ private int x,y,z; public punto3D(int a, int b, int c) {x=a;y=b;z=c;} }	21 %	public boolean verificacion(String n){ boolean a = flase; if (n.equals(“Java”)) a=true; else a=false; return a; }	26 %	24 %
S19	public class Puntos{ private int x;private int y;private int z; public Puntos(int cx,int cy,int cz) {x=cx;y=cy;z= cz;} }	20 %	public boolean verificar(String palabra){ boolean respuesta=false; if(palabra.equals(“java”)) {respuesta=true;} return respuesta; }	42 %	1 %
S20	public class Punto{ private int x;private int y;private int z; public Punto(){x =0;y=0;z=0;} }	36 %	public boolean verificarPalabra(String palabra){ if(palabra.equals(“Java”))return true;}	41 %	21 %
S21	public class Punto3D { private int X;private int Y;private int Z; public Punto3D(){X=0;Y=0;Z=0;} public Punto3D(int x, int y, int z){this.X = x;this.Y = y;this.Z = z;} public static void main(String[] args) {Punto3D p1 = new Punto3D();Punto3D p2 = new Punto3D(2, 3, 4);} }	34 %	public boolean stringIsJava(String cadena){ boolean resultado = false; if (cadena.equals(“Java”)) {resultado=true;} return resultado; }	38 %	1 %

Source: own work.

It is interesting to note that the expert credited as correct answer S9Q1 that would have been rejected by the method since it obtained a score of 31 %. The reason the method scored badly in this question is related to nomenclature: the student used different longer or shorter notation for class and variable names compared to those used in the template solutions (‘**3d**’ instead of ‘**Punto3D**’, ‘**puntox**’ instead of ‘**x**’, etc.). Thus the method added a number of edit operations (deletions when a shorter name was defined, and insertions for a longer name). Those alterations have actually no relation with the correct structure of the answer, but still influenced the correction rate. The same can be said about answer S3Q1 (41 %) which scored slightly higher because it additionally incurred in inversion errors (‘**pirvate**’ instead of ‘**private**’).

Let us move forward now to examination of results for Q2. Using the same threshold τ=20 % only three questions would have been accepted as correct (S1Q2, S7Q2, S17Q2). Rising the threshold to τ=30 % the accepted list is increased to 8 questions (S1Q2, S4Q2, S5Q2, S7Q2, S13Q2, S15Q2, S17Q2 and S18Q2). From these, only questions S7Q2, S15Q2, S17Q2 and S18Q2 were reckoned as correct by the expert; the remainder failed primarily because the comparison command ‘**text==“Java”**’ (which is invalid in the syntax of the language) should have been written as ‘**text.equals(“Java”)**’ .

It is worth to observe that the scores of Q2 are predominately higher than those of Q1 (that is, they scored worse). We found that compared to the answers provided to Q1, in Q2 the wrong use of nomenclature is a matter of concern: the test-maker asked to name the block ‘**compare()**’ but students used multiple variations including ‘**cadena()**’, ‘**verificacion()**’, ‘**verificar()**’, ‘**texto()**’, ‘**palabraJava()**’, ‘**java()**’, ‘**comparacion()**’, ‘**comprobar()**’, ‘**revisar()**’, ‘**verificarPalabra()**’, ‘**stringIsJava()**’. Similarly the function parameter was named ‘**palabra**’, ‘**frase**’, ‘**p**’, ‘**cadena**’, ‘**n**’ whereas the name requested by the text-maker was ‘**text**’. Moreover given that the expected solution to Q2 is shorter compared to Q1, these alterations in nomenclature actually represent a big number of insertion/deletion operations accounting to a large overall edit distance, and therefore, a large correction rate.

With the intention of validating the latter intuition, a third experiment was conducted where the nomenclature of the raw answers to Q2 was altered to mirror those of the expected template solutions and then were re-scored with the proposed method. Results are shown in the rightmost column of table 3. A quick view reveals that the scores are drastically lower (that is, better). Using again the threshold τ=20 % now 13 answers would have been accepted as correct (S1Q2, S2Q2, S4Q2, S5Q2, S6Q2, S7Q2, S10Q2, S13Q2, S15Q2, S16Q2, S17Q2, S19Q2, S21Q2), that is 62 % of the given answers. Among these however, several answers appear using the wrong comparison command mentioned before, which effectively renders them incorrect in the opinion of the expert. This suggests that a more stringent threshold can be used now given that the nomenclature has been rectified. Using τ=1 % for example, the only four answers using the proper ‘**.equals()**’ command and acceptable algorithmic structure will be accepted as correct. Notice also that some answers using the proper ‘**.equals()**’ command are nonetheless written using a bad algorithmic structure (e.g. S9Q2, S10Q2) and consequently they obtained bad scores compared to the new threshold. We additionally observe that answers including redundant temporal variables that obviously were not contemplated in the expert’s solutions scored badly as well (S11Q2, S14Q2) in spite of being correctly constructed. In summary we remark that should students have used the requested nomenclature and avoided unnecessary redundancies then the method would have been able to perform a more detailed assessment even to the level of a precise comparison command.

4. DISCUSSION AND CONCLUSION

The method described in this paper is intended to conduct automatic assessment of fill-in-the-blank questions tolerating irregular versions of a canonical answer due to orthographical or typographical mistakes. This feature is desirable within an open-ended short-free evaluation framework where a complete record of such variants is unfeasible to be anticipated in advance. The similarity score computed with the method allows for recognising an irregular answer which otherwise a naïve exact-matching comparison will miss. Besides, the similarity score could also being used to give partial credit which again is difficult to do with the naïve matching technique. Furthermore, when the irregular answer is recognised the expected (rectified) answer can be given *pari passu* as formative feedback to reinforce the learning process of the student. Open to consideration is the possible use of the method for real-time correction where hints to remedy possible mistakes are given to the student as he progresses writing up his answers to an online test.

In its existing form the scores computed with the method can be used to reach a crisp decision on accepting or rejecting an answer. But alternatively it may also be used to define judgment categories based on the similarity score, for instance “definitely accept”, “requiring human inspection” and “definitively reject”. In this way the labor of the human examiner would be narrowed down to those answers in the second category only. Otherwise answers can be given an automatic continuous grade in proportion to the score assigned with the method.

Two empirical studies were performed to illustrate the potential of the method. The first one was aimed at showing the ability of the method to recognise misspelled variants of a fill-in-the-blank question. On a list of 186 synthetic irregular answers a perfect-match rate of 80 % was achieved. When a correctness rate threshold of 30 % was set, the matching rate increased to 99 %. It is interesting to note that the additional recognized variants were regarded by a human expert as bizarre and unlikely to occur in reality, which may explain why they were initially rejected by the method. We believe that these results corroborate the promise of the method in performing a reliable assessment in this scenario. What is more, it could be valuable for questions requiring answers in a second language where the student is not native speaker and therefore particularly more susceptible to make spelling mistakes.

In the second empirical scenario the method was tested to assess a questionnaire where a longer but still short-free answer was expected. The experiment was constraint to the specific domain of computer programming where answers are fragments of code that must comply with stringent algorithmic patterns and syntax rules. The answers were collected from real-world code written up by students of a computer programming course. The method obtained promising results in recognizing answers that were judged on the whole as correct by a human expert, even if containing small syntax errors. We found however as a drawback that freedom in using nomenclature for entity names in the program code, may have a misleading impact in rejecting answers as incorrect. Thus either highlighted directions about permissible entity nomenclature accompanying the statement of the question, or other techniques such as regular expression recognition of nomenclature terms, are needed to obtain a more reliable assessment in this scenario.

The regular expression approach is an appealing avenue of future research and it may take advantage from the stringent structure characterizing fragments of code. In a computer language like *Java* it would be not difficult to define regular expression rule such as "**public class * {**" that would identify the name of a class, or the rule "**private int *;**" to extract the name of an attribute. The custom nomenclature can be drawn out in this way and be replaced with the canonical nomenclature defined by the test-maker and such operation will not alter the meaning or structure of the original fragment of code. Following this regular expression preprocessing the modified answer can be evaluated with the proposed method to give a more truthful assessment.

On the other hand it is remarkable that the same generic configuration of the method was used in both experiments despite being from two very different domains. Certainly tuning the method to the specific OOP domain of the second experiment will benefit the recognition rate so as to become more accurate in the identification of precise algorithmic structures or commands. For instance, it would be interesting to investigate the effect of having different configurations of costs (not only binary, but continuous values) in the edit distance function, as well as the effect of defining more precisely admissible punctuation marks and separators required by the syntax of the language when invoking this function.

As a concluding comment we recall in passing that the purpose of the method for checking the answer of an OOP question was to evaluate the semantics and not the syntax of the answer. Nonetheless a composite evaluation using an average of the results of our method and those of a compiler report over the fragment of code might give a more realistic scoring. Even more powerful would be to combine those scores and others obtained with Latent Semantic Analysis and other Natural Language Processing techniques (see e.g. [4] - [6]). The ultimate goal is to bring LMS and courseware management platforms closer to the realm of automatic scoring of semi-open-ended questions and release tutors and instructors from the tedious task of such manual assessment, allowing them to focus in learning-support activities of more merit.

REFERENCIAS

[1] H. Schuman, S. Presser, "The Open and Closed Question", *American Sociological Review*, vol. 44, no. 5, pp. 692-712, 1979.

[2] S. Jordan, T. Mitchell, "E-assessment for learning? The potential of short free-text questions with tailored feedback", *British Journal of Educational Technology*, vol. 40, no. 2, pp. 371-385, 2009.

[3] L. Cutrone, M. Chang, "Automarking: Automatic Assessment of Open Questions", in *Proceedings of the 10th IEEE International Conference on Advanced Learning Technologies*, pp. 143-147, Sousse, Tunisia, July, 2010.

[4] R. Klein, A. Kyrilov, M. Tokman, "Automated assessment of short free-text responses in computer science using latent semantic analysis", in *Proceedings Of The 16th Annual Joint Conference On Innovation And Technology In Computer Science Education* (ITiCSE '11), ACM, New York, USA, pp. 158-162, 2011.

[5] D. Perez, A. Gliozzo, et ál., "Automatic assessment of students' free-text answers underpinned by the combination of a bleu-inspired algorithm and latent semantic analysis", in *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference, FLAIRS,* 2005.

[6] N. Kaur, K. Jyoti, "Automated assessment of short one-line free-text responses in computer science", *International Journal of Computer Science and Informatics*, vol. 2, no. 1, pp.105-109, 2012.

[7] V. Cardona, and S. Rojas, "Recognising irregular answers in automatic assessment of fill-in-the-blank tests", in *Proceedings of the IEEE Workshop on Engineering Applications* (WEA 2012), Bogotá, Colombia, 2012.

[8] R. Wagner, and M. Fischer, "The string-to-string correction problem", *Journal of the ACM*, 21, pp. 168-173, 1974.