Machines that learn how to code open-ended survey data

Andrea Esuli and **Fabrizio Sebastiani**
Consiglio Nazionale delle Ricerche, Italy

We describe an industrial-strength software system for automatically coding open-ended survey responses. The system is based on a *learning* metaphor, whereby automated verbatim coders are *automatically* generated by a general-purpose process that learns, from a user-provided sample of manually coded verbatims, the characteristics that new, uncoded verbatims should have in order to be attributed the codes in the codeframe. In this paper we discuss the basic workings of this software and present the results of experiments we have run on several datasets of real respondent data, in which we have compared the accuracy of the software against the accuracy of human coders.

Introduction

Open-ended questions are an important way to obtain informative data in surveys, and this is so for a variety of applications, including market research, customer relationship management, enterprise relationship management, and opinion research in the social and political sciences (Schuman & Presser 1979; Reja *et al*. 2003). Closed questions generate data that are certainly more manageable, but suffer from several shortcomings, since they straitjacket the respondent into conveying her thoughts and opinions into categories that the questionnaire designer has developed a priori. As a result, a lot of information that the respondent might potentially provide is lost. On the contrary, open-ended questions allow freedom of thought, and the responses that are returned may provide perspectives and slants that had not been anticipated by the questionnaire designer, thus providing far richer information on the opinions of the respondent. Furthermore, asking an open question tells the respondent

Received (in revised form): 10 March 2009

© 2010 The Market Research Society
DOI: 10.2501/S147078531020165X

that her opinions are seriously taken into account and her needs cared about; the same cannot be said of closed questions, since these may instead convey the impression that the interviewers are interested only in orthodox responses and orthodox respondents.

Unfortunately, the price one has to pay for including open-ended questions in a questionnaire is a much greater difficulty in using the data obtained from the respondents. Once the designer has developed a codeframe (aka 'codebook') for the question under consideration, a human coder needs to read the returned answers one by one in order to assign them the appropriate codes; this may require a lot of human effort, depending on the size of the respondent pool, and does not lend itself to the fast turnaround of results.

In the recent past, computer-assisted methods of coding open-ended verbatim responses (henceforth 'verbatims') have been proposed as a solution. Unfortunately, they still fall short of truly automating the coding process, since they all require a lot of human involvement in the coding process. Some of these systems, such as Confirmit[1] (O'Hara & Macer 2005), Voxco's Command Center[2] (Macer 2007b), SPSS' mrInterview,[3] and Snap[4] are (as far as open-ended questions are concerned) essentially powerful, user-friendly systems that assist and enhance the productivity of the user in manually coding the verbatims; the only difference with coding as it was performed before computers were born, is that all the objects of interest are in digital form, so that paper and pencil are not involved. Some other systems, such as Language Logic's Ascribe[5] (Macer 2002), SphinxSurvey[6] (Macer 1999), streamBASE GmbH's Coding-Modul[7] (Macer 2007a), SPSS' Text Analysis for Surveys,[8] Provalis Research's Wordstat[9] (Macer 2008), and the no longer available Verbastat (Macer 2000), further assist the user by means of (sometimes sophisticated) word searching, text matching or 'text mining' capabilities; still, the decision whether a given code should or should not be attributed to a verbatim ultimately rests with the user. Other systems, such as iSquare's i2 SmartSearch,[10] rely on the user to engineer rules for automated verbatim coding; while these rules are indeed capable

[1] www.confirmit.com/ access date 29 June 2010
[2] www.voxco.com/ access date 29 June 2010
[3] www.spss.com/mrinterview/ access date 29 June 2010
[4] www.snapsurveys.com/ access date 29 June 2010
[5] www.languagelogic.info/ access date 29 June 2010
[6] www.sphinxsurvey.com/en/home/home_sphinx.php access date 29 June 2010
[7] www.streambase.de/ access date 29 June 2010
[8] www.spss-sa.com/spss_text_analysis_for_surveys.html access date 29 June 2010
[9] www.provalisresearch.com/wordstat/Wordstat.html access date 29 June 2010
[10] www.isquare.de/i2SmartSearch.htm access date 29 June 2010

of automating the coding process, the human effort involved in writing the rules is still high, as is the level of expertise required.

In this paper we propose instead a radically different approach to developing automated verbatim coding systems. The approach is based on a *learning* metaphor, whereby automated verbatim coders are *automatically* generated by a general-purpose process that learns, from a user-provided sample of manually coded verbatims, the characteristics that new, uncoded verbatims should have in order to be attributed the codes in the codeframe. This approach adds a further level of automation to the methods described above, since no human involvement is required aside from that of providing a sample of manually coded verbatims. The net effect is that any human coder – not necessarily a computer-savvy one – may set up and operate such a system in full autonomy.

In the remainder of the paper we will exemplify this approach by describing an industrial-strength software system (dubbed VCS: Verbatim Coding System) that we have developed along these lines. This software can code data at a rate of tens of thousands of open-ended verbatims per hour, and can address responses formulated in any of five major European languages (English, Spanish, French, German and Italian).

The rest of this paper is organized as follows. The next section describes the basic philosophy underlying the machine learning approach to verbatim coding and the overall mode of operation of the VCS system. After that, we present the results of experiments in which the accuracy and the efficiency of VCS are tested on several datasets of real respondent data. Then we add further insight to VCS by giving, in question-and-answer format, a number of clarifications of its workings. Next, we look at some further features available in VCS, and this is followed by the conclusion.

VCS: an automated verbatim coding system

VCS is an adaptive system for automatically coding verbatim responses under *any* user-specified codeframe; given such a codeframe, VCS *automatically* generates an automatic coder for this codeframe.

Actually, the basic unit along which VCS works is not the codeframe but the *code*: given a codeframe consisting of several codes, for each such code VCS automatically generates a *binary coder*, i.e. a system capable of deciding whether a given verbatim should or should not be attributed the code. The consequence of this mode of operation is that all binary coders are applied to the verbatim independently of each other, and that zero, one or several codes can be attributed to the same verbatim (however,

see Q&A 5 in the section titled 'Frequently asked questions', below, for exceptions).

VCS is based on a *learning metaphor*, according to which the system learns from manually coded data the characteristics that a new verbatim should have in order to be attributed the code. The manually coded verbatims that are fed to the system for the purpose of generating the binary coders are called the *training verbatims*. The training verbatims need to include *positive examples* of the code (i.e. verbatims to which a human coder has attributed the code) and *negative examples* of the code (i.e. verbatims to which a human coder has decided not to attribute the code). By examining both, the system generates a 'mental model' of what it takes for a verbatim to be attributed the code; once these mental models (namely, the binary coders) are generated, they can be applied to coding as yet uncoded verbatims (from now on the as yet uncoded verbatims that are automatically coded by the binary coders will be called *test verbatims*). It is typically the case that, in a real application, the training verbatims will be much fewer than the test verbatims, so that the human coder, after coding a small portion of a survey and training the system with it, will have the binary coders code automatically the remaining large part of the survey.

In practice, it is not the case that training proceeds on a code-by-code basis. In VCS a user wanting to provide training examples to the system typically reads a verbatim and attributes to it the codes she deems fit; the intended meaning is that for all the codes in the codeframe that she does not attribute to the verbatim, the verbatim is to be considered a negative example.

It is important to recognize that VCS does not attempt to learn how to code in an 'objectively correct' fashion, since coding is an inherently subjective task, in which different coders often disagree with each other on a certain coding decision, even after attempts at reconciling their views. What VCS learns to do is to *replicate the subjective behaviour of the human coder who has coded the training examples*. If two or more coders have been involved in manually coding the training examples, each coding a separate batch of verbatims, then VCS will mediate between them, and its coding behaviour will be influenced by the subjectivities of both. This means that it is of key importance to provide training examples that are reliably coded, if possible by an expert coder (however, see the sub-section entitled 'Training data cleaning', below, for some computer assistance in this phase).

Advantages of the learning-based approach are that, unlike with several other computerised solutions for coding open-ended data:

- no domain-dependent dictionaries are involved; VCS can be called a 'plug-and-play' system, where the only input necessary for the system to work is manually coded data for training
- there is no need to pay experts for writing coding rules in some arcane (Boolean, fuzzy, probabilistic, etc.) language; what the system does is basically generate those rules automatically, without human intervention, from the training examples
- it is easy to update the system to handle a revised codeframe, a brand new codeframe or a brand new survey; if a user, after training the system, needs to add a new code to the codeframe, she need only add training examples for the new code (this may simply mean checking which of the previously coded examples have the new code too) and have the system generate the binary coder for the new code; the binary coders for the other codes are unaffected; if a user, after training the system, needs to set it up to work on an entirely new question, or an entirely new survey, she does not need to 're-program' the system, she needs only to provide the appropriate training examples for the new question or survey (on this, see Q&A 6 in the section titled 'Frequently asked questions', below).

Note that VCS does not even look (i) at the text of the question that has elicited the answer, or (ii) at the textual descriptions of the codes in the codeframe (the codes can thus be numerical codes with no apparent meaning). So, how does VCS learn from manually coded verbatims? VCS is endowed with a sophisticated linguistic analysis module that, given a verbatim, extracts *linguistic patterns* from it; in essence, this module transforms the verbatim into an internal representation that consists of the set of patterns extracted from the verbatim. Patterns may vary widely in nature. Some patterns are of a lexical nature (i.e. they are simple words), some are of a syntactic nature (e.g. noun phrases, verb phrases, clauses, entire sentences), some have a semantic nature (in which semantic categories – some of them sentiment-related – have been substituted for actual words), and some are of yet different types. More importantly, once the patterns have been extracted from the training verbatim, VCS learns how each pattern is correlated to a given code. For instance, if a given pattern occurs in most training verbatims labelled with code *C*, and occurs in only a few of the training verbatims that are *not* labelled with *C*, then it is deemed to be highly correlated with code *C*. Such a pattern is useful, since when it is discovered in an as yet uncoded verbatim, it will bring evidence that the verbatim should be attributed code *C*. All patterns

encountered in the uncoded verbatim will thus bring a little piece of evidence, for or against attributing code *C* to the verbatim; a complex rule for the combination of evidence will then take the final decision.

A visual description of the process upon which VCS is based is given in Figure 1. The area at top left represents the *training* phase: a human coder manually codes a (typically small) sample of uncoded verbatims and feeds them to a 'trainer', who then generates the binary coders. The area at the bottom represents the automatic *coding* phase: the binary coders generated in the training phase are fed to a general-purpose coding engine that, once fed with a (typically large) set of yet uncoded verbatims, automatically codes them. Now these verbatims are ready for use in reporting (bottom left) or for taking individual decisions based on the codes automatically attributed to the individual responses, such as calling up a customer whose response has been given the code 'Very unhappy; may defect to the competition' (bottom right).

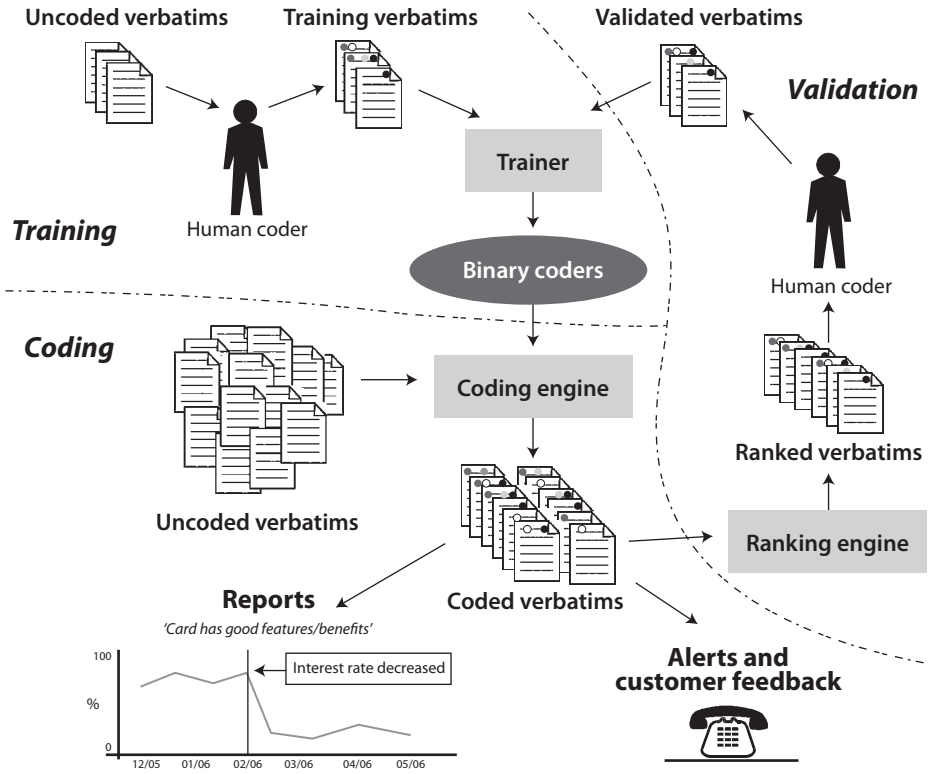

Figure 1 A visual description of the process upon which VCS is based

The area at top right represents a phase we have not discussed yet: the *validation* phase. After automatic coding has been performed, the user may wish to take a look at some of the automatically coded verbatims and correct any potential mistakes she spots. If she does so, the manually corrected verbatims may be used as further training examples, so that the system can be re-trained with an augmented training set. It turns out, unsurprisingly, that the re-trained binary coders tend to be more accurate than the previously generated ones, especially when coding verbatims that are somehow 'similar' to the ones that have been manually corrected. More than one re-training iteration can be performed, depending on available 'humanpower' and on whether or not the desired level of accuracy has been reached.

Usually, in a given iteration of the validation process the user will inspect and correct only a small portion of the automatically coded verbatims. VCS, upon returning the automatically coded verbatims, sorts them in terms of the *confidence* with which the binary coders have coded them, so that the verbatims that VCS has coded with the smallest confidence will be placed at the top of the list. The net effect is that the user is encouraged to, first and foremost, validate (i) the verbatims that have a higher chance of being miscoded, which allows her to remove mistakes from the result set; and (ii) the verbatims that, being problematic to VCS, convey the largest amount of information to VCS when provided as training examples.

Testing VCS on real-world sets of verbatim data

In this section we present the results of testing VCS on a number of benchmark datasets of real respondent data. From now on, when we refer to a *dataset* we mean a set of *manually coded* verbatims returned by respondents to a given question, plus the codeframe that the human coders have used for coding them.

Different qualities of a system may be tested in an experiment. In our experiments we test three aspects, namely:

- *accuracy*, which measures how frequently the coding decisions of VCS agree with the coding decisions of the human coder who originally coded the data
- *training efficiency*, which measures how fast is VCS in training the binary coders for a given codeframe when using a given training set
- *coding efficiency*, which measures how fast the automatically generated binary coders are in coding as yet uncoded verbatims.

VCS accuracy tests are the subject of the section entitled 'Testing for accuracy', below, while VCS training and coding efficiency tests are the subject of the section that follows it, entitled 'Testing for efficiency'.

Usually, an experiment involving learning machines is run according to the *train-and-test* methodology, which consists of splitting the dataset into two non-overlapping parts:

1. the *training set*, which is used for training the binary coders
2. the *test set*, which is used for testing the binary coders; this is done by feeding the binary coders with the verbatims in the test set, each stripped from the codes that have been attributed to it by the human coder, asking the binary coders to code them, and comparing the coding decisions of the binary coders with the coding decisions the human coder had taken on the same verbatims.

In a well-crafted experiment it is important that there are no verbatims in common between the training set and the test set since, quite obviously, it would be unrealistically easy for a binary coder to code the verbatims it has been trained on.

We have carried out all our experiments according to a popular, more refined variant of the train-and-test technique, called *10-fold cross-validation*. This consists of splitting the dataset into ten equally sized sets of verbatims, and running ten train-and-test experiments, each of which consists of using one set (always a different one) as the test set and the union of the other nine as the training set. The performance of the binary coders is then computed as the average performance that the binary coders have displayed across the ten different experiments. This 10-fold cross-validation provides more reliable results than a single train-and-test experiment, since all the verbatims in the dataset are, sooner or later, being tested upon.

Table 1 reports the results of our experiments on 13 different datasets (the meaning of the various column headers will be clarified soon). The first ten datasets (LL-A to LL-L) consist of verbatims from market research surveys and were provided by Language Logic LLC. The LL-B, LL-D and LL-F to LL-L datasets are from a large consumer packaged good study, with both open-ended and brand-list questions. The LL-A, LL-C and LL-E datasets are instead from one wave of a continuous ('tracking') survey that Language Logic LLC codes 12 times a year, which consists of 'semi-open' brand questions (i.e. questions – such as 'What is your favourite soft drink?' – that, although in principle eliciting a textual response, usually generate

Table 1 Results of experiments on ten market research datasets (LL-A to LL-L), two customer satisfaction datasets (Egg-A and Egg-B), and one social science dataset (ANES L/D)

DS	#V	#C	AVC	AVL	F_1^{μ}	PD_A	PD_M	Tt	Ct
LL-A	201	18	21.00	1.35	0.92	0.8%	4.0%	0.9	0.1
LL-B	501	34	26.65	1.65	0.90	0.6%	4.8%	10.0	0.4
LL-C	201	20	10.05	1.61	0.89	0.7%	7.4%	0.9	0.1
LL-D	501	27	45.30	3.32	0.85	0.8%	5.6%	24.7	1.0
LL-E	201	39	8.41	2.57	0.84	0.4%	2.5%	4.7	0.2
LL-F	501	57	37.58	6.99	0.82	0.7%	4.8%	61.0	2.5
LL-G	501	104	21.30	6.25	0.80	0.5%	5.2%	123.3	5.0
LL-H	501	86	30.08	7.87	0.79	0.7%	5.7%	136.4	5.5
LL-I	501	69	33.16	7.70	0.78	0.8%	5.2%	102.3	4.1
LL-L	501	65	29.40	5.58	0.75	1.0%	9.6%	84.9	3.4
Egg-A	1000	16	123.37	27.37	0.62	1.5%	3.1%	173.0	7.0
Egg-B	924	21	67.19	26.47	0.55	1.6%	3.6%	175.6	7.1
ANES L/D	2665	1	1396.00	30.83	0.86	4.6%	4.6%	47.0	1.9

Notes: The columns represent the name of the dataset (DS), the number of verbatims in the dataset (*#V*), the number of codes in the codeframe (*#C*), the average number of positive training verbatims per code (*AVC*), the average number of (non-unique) words in a verbatim (i.e., the 'average verbatim length' – *AVL*), the accuracy at the individual level (F_1^{μ}), the accuracy at the aggregate level (PD_M and PD_A), training time (*Tt*) and coding time (*Ct*).

many responses consisting of only the name of a product or brand, with this name coming from a small set of such names). The next two datasets (EGG-A and EGG-B) consist of verbatims from customer satisfaction surveys and were provided by Egg plc;[11] for both datasets, which were collected in the context of two different surveys, respondents were answering the question 'Have we done anything recently that has especially disappointed you?' The last dataset (ANES L/D) consists of verbatims from a political survey run in 1992 and was obtained from the American National Election Studies (ANES) committee. Two sets of verbatims were used: the first was returned in answer to the question 'Is there anything in particular about Mr Clinton that might make you want to vote for him? If so, what is that?' while the second was returned in answer to the question 'Is there anything in particular about Mr Clinton that might make you want to vote against him? What is that?' Our coding task consisted of guessing whether the verbatim belongs to the former or to the latter set.

[11] http://www.egg.com/ access date 29 June 2010

Testing for accuracy

Accuracy may be measured at two different levels:

1. at the *individual level* – a hypothetical perfect coding system is the one that, given a code *C*, assigns *C* to all and only the verbatims to which the human coder would assign *C*
2. at the *aggregate level* – here, a hypothetical perfect coding system is the one that, given a code *C*, assigns *C* to *x*% of the verbatims exactly when the human coder would assign *C* to *x*% of the verbatims.

The former notion of accuracy is especially interesting for measuring the suitability of the system to customer satisfaction applications, where a user is interested in correctly classifying each individual respondent, so as to be able to cater for her specific needs (e.g. ringing her up if she is particularly unsatisfied). The latter notion of accuracy, on the contrary, is especially interesting for opinion surveys and market research, where a user is typically interested in the *percentage* of respondents that fit under a given class, and may be less interested in knowing what each individual respondent thinks.

Note that accuracy at the individual level implies accuracy at the aggregate level, but not vice versa! A system perfectly accurate at the aggregate level may be inaccurate at the individual level; this may happen when the coding errors a system has made are split into two equally sized subsets of false positives and false negatives, respectively. Therefore, measures of accuracy at the aggregate level are somehow more lenient than measures of accuracy at the individual level (but they are certainly no less important).

Testing for accuracy at the individual level

Mathematical measures of individual accuracy Accuracy testing requires a mathematical measure of accuracy to be defined and agreed upon. The one we adopt for accuracy at the individual level (or individual accuracy, for short), called F_1, consists of the 'harmonic mean' of two other measures, precision and recall.

- For a code *C*, *precision* (denoted π) measures the ability of the system to avoid 'overcoding', i.e. attributing *C* when it should not be attributed. In other words, precision measures the ability of the system to avoid 'false positives' (aka 'errors of commission') for code *C*.
- For a code *C*, *recall* (denoted ρ) measures the ability of the system to avoid 'undercoding', i.e. failing to attribute *C* when it should instead

be attributed. In other words, recall measures the ability of the system to avoid 'false negatives' (aka 'errors of omission') for code *C*.

Table 2 The four-cell contingency table for code *C* resulting from an experiment

		Coder says	
Code *C*		**YES**	**NO**
System says	**YES**	*TP*	*FP*
	NO	*FN*	*TN*

In a given experiment, precision and recall for a code *C* are computed from a four-cell *contingency table* (see Table 2), whose cells contain the numbers of true positives, false positives, false negatives and true negatives, respectively, resulting from the experiment. Precision, recall and F_1 are defined as:

$$\pi = \frac{TP}{TP + FP} \qquad \rho = \frac{TP}{TP + FN}$$

$$F_1 = \frac{2 \cdot \pi \cdot \rho}{\pi + \rho} = \frac{2 \cdot TP}{(2 \cdot TP) + FP + FN}$$

For instance, assume that our test set contains 100 verbatims and that our codeframe consists of two codes: C_1 and C_2. Table 3 illustrates two possible contingency tables for C_1 and C_2, and the relative computations of precision, recall and F_1.

Precision, recall and F_1 can also be computed relative to an entire codeframe (in this case they are noted π^μ, ρ^μ and F_1^μ) by using a 'combined' contingency table. Table 4 illustrates such a contingency table as resulting

Table 3 Example contingency tables for two hypothetical codes C_1 and C_2 and the corresponding computations for precision, recall and F_1

		Coder says	
Code C_1		**YES**	**NO**
System says	**YES**	15	7
	NO	7	70

$$\pi = \frac{15}{15+7} = \frac{15}{22} = 0.682$$

$$\rho = \frac{15}{15+8} = \frac{15}{23} = 0.652$$

$$F_1 = \frac{2 \cdot 0.682 \cdot 0.652}{0.682 + 0.652} = \mathbf{0.667}$$

		Coder says	
Code C_2		**YES**	**NO**
System says	**YES**	22	13
	NO	5	60

$$\pi_j = \frac{22}{22+13} = \frac{22}{35} = 0.629$$

$$\rho_j = \frac{22}{22+5} = \frac{22}{27} = 0.815$$

$$F_1 = \frac{2 \cdot 0.629 \cdot 0.815}{0.629 + 0.815} = \mathbf{0.710}$$

Table 4 Combined contingency table for codes C_1 and C_2 as resulting from the data of Table 3 and the corresponding computations for precision, recall and F_1

Codes C_1 and C_2		Coder says: **YES**	Coder says: **NO**
System says	**YES**	15 + 22	7 + 13
	NO	8 + 5	70 + 60

$$\pi^{\mu} = \frac{(15+22)}{(15+22)+(7+13)} = \frac{37}{57} = 0.649$$

$$\rho^{\mu} = \frac{(15+22)}{(15+22)+(8+5)} = \frac{37}{50} = 0.740$$

$$F_1^{\mu} = \frac{2 \cdot 0.649 \cdot 0.740}{0.649 + 0.740} = \mathbf{0.692}$$

from the data of Table 3 (the hypothetical codeframe here consists of codes C_1 and C_2 only).

There are several reasons why F_1 is a good measure of individual accuracy. First of all, F_1 always equals 0 for the 'pervert binary coder' (the one for which $TP = TN = 0$, i.e. no correct coding decision) and F_1 always equals 1 for the 'perfect system' (the one for which $FN = FP = 0$, i.e. no wrong coding decision). Second, it partially rewards partial success: i.e. if the true codes of a verbatim are C_1, C_2, C_3, C_4, attributing it C_1, C_2, C_3 is rewarded more than attributing it C_1 only. Third, it is not 'easy to game', since it has very low values for 'trivial' coding systems – e.g. the 'trivial rejector' (the binary coder that never assigns the code) has $F_1 = 0$ while the 'trivial acceptor' (the binary coder that always assigns the code) has $F_1 = 2 \cdot TP/(2 \cdot TP) + FP$, which is usually low). Fourth, it is symmetric – i.e. its values do not change if one switches the roles of the human coder and the automatic coder; this means that F_1 can also be used as a measure of agreement between any two coders (human or machine) since it does not require us to specify who among the two is the 'gold standard' against which the other needs to be checked. For all these reasons, F_1 is a de facto standard in the field of text classification (Sebastiani 2002, 2006).

How accurate is VCS? In order to analyse the results of Table 1, let us first concentrate on individual accuracy and F_1. The F_1 values obtained across the 15 experiments vary from $F_1 = 0.92$ (best result, on LL-A) to $F_1 = 0.55$ (worst result, on Egg-B), with an average of $F_1 = 0.77$.

How good are these results? Let us remember that F_1 is a measure of how closely VCS can mimic the coding behaviour of the human coder (let's call her K_1) who has originally coded the test data. Since the alternative to computer coding is manual coding, the question 'How good are the results of VCS?' is probably best posed as 'How accurate are the results of VCS with respect to the results I could obtain from a human coder?' In order to make this comparison, what we can do is check how closely this latter

coder (let's call her K_2) can mimic the coding behaviour of K_1, and compare $F_1(VCS, K_1)$ with $F_1(K_2, K_1)$. In other words, an answer to this question can be given only following a careful *intercoder agreement* study, in which two human coders, K_1 and K_2, are asked to independently code a test set of verbatims after one coder has 'learned how to code' from a training set of verbatims coded by the other coder. Unfortunately, this question is left unanswered for most of the 13 datasets listed in Table 1, since for coding most of them only one human coder was involved, and no intercoder agreement studies were performed. The only exceptions are the Egg-A and Egg-B datasets: in Macer, Pearson & Sebastiani (2007), the accuracy of VCS was tested in the context of a thorough intercoder agreement study, which showed that VCS obtained F_1 values, with respect to either K_1 or K_2, only marginally inferior to $F_1(K_2,K_1)$.[12]

Why does the accuracy of VCS vary so widely? A second question we might ask is 'Why does the accuracy that VCS obtains vary so widely across different datasets? Can I somehow predict where in this range will VCS perform on my data?' The answer to this is: 'Somehow, yes.' We have experimentally observed (and machine learning theory confirms) that the F_1 of VCS tends (i) to increase with the average number of (positive) training verbatims per code (AVC) provided to the system, and (ii) to decrease with the average length of the training verbatims (AVL). In short, a dataset in which there are many (resp., few) training verbatims per code and whose verbatims tend to be short (resp., long) tends to bring about high (resp., low) F_1.

Note that parameter (i) is within the control of the user, while parameter (ii) is not. That is, if there are too few training examples per code, the user can manually code new verbatims, thus increasing AVC at will; however, if verbatims are long, the user certainly cannot shorten them. So, is VCS doomed to inaccuracy when the dataset consists of long verbatims (i.e. AVL is high), as is the case in customer satisfaction datasets? No, it is simply the case that, if AVL has a high value, AVC needs to have a high value too in order to 'compensate' for the high AVL value. A simple explanation for this phenomenon may be offered through an example drawn from college mathematics. Long verbatims usually mean complex, articulated sentences, with many different linguistic expressions (e.g. words, phrases) and phenomena (e.g. syntactic patterns, idioms) showing up in the dataset.

[12] These F_1 results are not reported here since, in Macer *et al*. (2007), experiments were run with a standard training-and-test methodology, with a 70/30 split of the data between training and test, and are thus not comparable with the F_1 results of the present paper, obtained with 10-fold cross-validation and with a 90/10 split.

If the number of phenomena to 'explain' is large, there needs to be a large amount of information in order to explain them. This information is nothing else than the training examples; this explains why when the data consists of long verbatims, many training examples are needed to make sense of them, i.e. to obtain good accuracy. This is akin to what happens in systems of linear equations: if there are many variables, many equations are needed to find a solution (if there are many variables and few equations, the system is underconstrained). In our case, the linguistic expressions and phenomena are the variables, and the training verbatims are the equations constraining these variables: long verbatims mean many linguistic phenomena, i.e. many variables, and accuracy thus requiremany training verbatims, i.e. many equations.

Testing for accuracy at the aggregate level

We now move to discussing accuracy at the aggregate level (or *aggregate accuracy*, for short). To measure it we use a mathematical function that we call *percentile discrepancy* (*PD*, for short), defined as the absolute value of the difference between the true percentage and the predicted percentage of verbatims with code *C*. In other words, suppose that 42% of the verbatims in the test set have code *C*, and suppose that VCS applies instead code *C* to 40.5%, or to 43.5%, of the verbatims in the test set; in both cases we say that VCS has obtained 1.5% *PD* on code *C*. A hypothetical ideal binary coder always obtains $PD = 0$.

For all tests in Table 1 we report both the average and the maximum *PD* value obtained across all codes (noted PD_A and PD_M, respectively). A look at these results shows that VCS is indeed very, very accurate at the aggregate level – for instance, PD_A results tell us that in 10 studies out of 13 VCS errs, on average across all codes in the codeframe, by 1.0% or less; this is a tolerable margin of error in many (if not most) market research applications, and it is even more tolerable if this is the only price to pay for coding very large amounts of verbatim data with no manual effort.

In order to visualise this, Figure 2 displays an example histogram from one of the datasets of Table 1 (dataset LL-E); for each code in the codeframe, two thin bars are displayed side by side, the rightmost representing the true percentage of verbatims that have the code, and the other the percentage of verbatims that VCS predicts have the code; the figure gives a compelling display of the aggregate accuracy of VCS.

Why is VCS so good at the aggregate level? The explanation is slightly technical, and comes down to the fact that, in order to obtain results accurate at the individual level, the internal algorithms of VCS attempt to

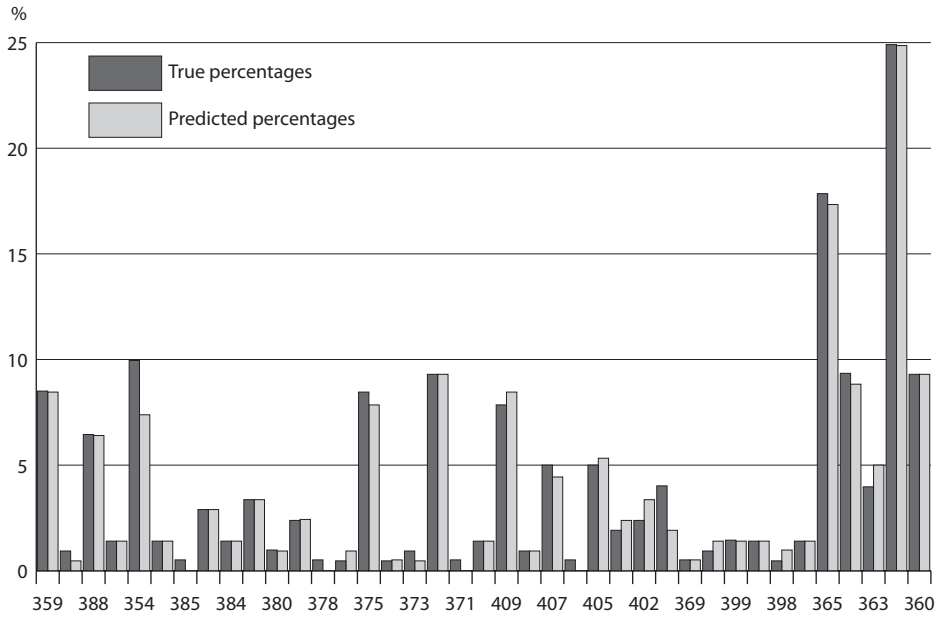

Figure 2 True percentages against predicted percentages for the LL-E dataset. The worst value of *PD* (2.5%) is obtained for code 354, where 10% is the correct percentage and 7.5% is the percentage predicted by VCS

generate binary coders that maximise F_1. Maximising F_1 means trying to balance false positives and false negatives, since F_1 has the property that, among a set of candidate binary coders for code C that make the same number of mistakes on the same dataset, the one that has the highest F_1 is the one for which the mistakes are equally split into false positives (*FP*) and false negatives (*FN*). And when $FP = FN$, then $PD = 0\%$ …

Of course, similarly to individual accuracy, how good are the *PD* values obtained by VCS would be best assessed with respect to an intercoder agreement study. Interestingly enough, on EGG-A and EGG-B (the only datasets in Table 1 for which intercoder agreement studies were performed, as reported in Macer *et al*. (2007), the *PD* values that human coders K_1 and K_2 obtained with respect to each other, were similar or sometimes even higher (i.e. worse) than the results that VCS obtained with respect to either coder! The reason is that it is often the case than one coder is *consistently* more liberal than the other in assigning the code (i.e. her mistakes are mostly false positives) while the other is *consistently* more conservative (i.e. her mistakes are mostly false negatives);[13] both these behaviours bring

[13] This is an example of so-called *correlated coder error*; see e.g. Sturgis (2004).

about high *PD*, while VCS, by attempting to maximise F_1, attempts to strike a balance between liberal and conservative tendencies, thus reducing *PD*.

Testing for efficiency

We now move to discussing efficiency (i.e. computer time requirements) issues. Training and coding running times (in seconds) for all of our 15 experiments are reported in Table 1; each result refers to the time required to run one of the ten experiments involved in the 10-fold cross-validation, thus involving training from 90% of the verbatims and coding the other 10%.

In order to give an idea of the efficiency of VCS on a more representative example, our tests on the EGG-A and EGG-B datasets translate into the fact that, for a 20-code codeframe, (i) the binary coders can be trained from 1000 training examples in approximately two minutes altogether; and (ii) under the same codeframe, 100,000 verbatims can be coded automatically in approximately eight minutes. In our tests on the LL-A to LL-L datasets both training and coding were, on average, approximately 7.6 times faster than on EGG-A and EGG-B (due to lower AVL). In general, the algorithms we employ within VCS are such that training time (resp., coding time) increases proportionally with the number of training verbatims (resp., with the number of verbatims to code), with the number of codes in the codeframe, and with the average length of the training verbatims.

Overall, the results above indicate that VCS is completely up to being employed even on studies of gigantic size (e.g. as when coding huge backlogs of data for retrospective analysis), on which training and coding would be performed literally in a matter of a few hours altogether.

Frequently asked questions

We now try to clarify a number of points about the workings of VCS by adopting a question-and-answer format, thus dealing with the questions that are more frequently asked about VCS.

1 Does VCS attribute identical codes to identical verbatims?
Yes, as may be expected, two or more identical copies of the same verbatim that are encountered at coding time are attributed exactly the same codes. Unless, of course, the binary coders have been retrained after validation, or have been retrained with a different (e.g. augmented) set of training

verbatims, in which case a verbatim that had been coded one way before retraining could be coded another way after retraining.

2 What about multiple (consistently coded) training copies of the same verbatim? Will VCS treat them as if a single copy had been included in the training set?
No, VCS is sensitive to the presence of multiple consistently coded training copies of the same verbatim; these have the effect of 'reinforcing the point', e.g. telling the system that this verbatim is an *important* positive example of the codes that are attached to it. This case indeed occurs often, e.g. in 'semi-open questions' from market research questionnaires, in which answers, albeit textual in nature, are typically very short, often consisting of one or two words. It certainly seems natural that verbatims frequently encountered in training should be treated in a 'reinforced' way.

3 If two inconsistently coded copies of the same verbatim are provided for training, does the system fall apart?
No, VCS was designed to handle these types of inconsistency, since it is frequently the case that inconsistently coded copies of the same verbatim are encountered at training time (especially if the training set has been generated by two or more human coders, each working independently of the others and each coding a separate batch of examples). VCS implements a 'fail soft' policy, such that, if two copies of verbatim *v* are encountered at training time, one a positive and the other a negative example of code *C*, the binary coder for code *C* will simply behave as if it had been provided with neither of these two training examples – i.e. contradictory pieces of training information neutralise each other. The binary coders for codes other than *C* are obviously unaffected.

4 If a given verbatim is provided as a training example, and an identical copy of it is encountered again, uncoded, in the coding phase, does this copy receive the same codes that the training copy has?
Not necessarily. The 'training' copy of the verbatim can have a set of codes, and the 'uncoded' copy may be automatically attributed a different set of codes, since the coding behaviour of the system is determined, holistically, by all the training it has received, and not by a single training example. Were it not like this, of course, the system could not 'fail soft' as discussed in the answer to question 3.

5 Does VCS always attribute at least one code to each verbatim?
No. In the default setting VCS may attribute several codes, one code or no code at all to a given verbatim, since each code in the codeframe is treated in isolation of the others. If no code at all is attributed to the verbatim, this is akin to giving the verbatim the special code 'Others'.

However, the user may change this default setting on a question-by-question basis, i.e. for a given question the user may specify that one and only one of the following constraints should be enforced: (i) at least *n* codes must be attached to a verbatim; (ii) at most *n* codes must be attached to a verbatim; (iii) exactly *n* codes must be attached to a verbatim. In all these, the value of *n* can be specified by the user.

6 I have trained the system to work for a certain question, using a certain codeframe. Now I need to set it up to work on a different question, but using the same codeframe as before. Can I use for this new question the binary coders I had trained on the same codeframe for the previous question?
Yes, or sort of. The best course of action is to use the previously generated binary coders and then to engage in an accurate validation effort. The reason is that there are both commonalities and differences in meaning between the same code as used in two different contexts. It is our experience that the commonalities are stronger than the differences, so this suggests that we should leverage on the previously generated binary coders; however, careful validation will smooth out the differences, and will attune the binary coders to the meaning that the codes take up in the new context. In sum, if the manual effort one can afford is, say, manually coding 500 (or even fewer) verbatims, one can certainly start from scratch, but it is probably better if one instead runs on the uncoded verbatims the binary coders previously generated for the same codeframe and then validates 500 of the automatically classified ones.

7 Does VCS cater for sentiment-related codes/distinctions, such as 'Positive' and 'Negative'?
Yes, VCS was designed with survey research in mind – i.e. with the awareness that surveys not always attempt to capture purely topic-related distinctions but often attempt to capture notions that have to do with the emotions, sentiments and reactions of respondents. See 'Concluding remarks', below, for more details.

8 How much time and money am I going to save on a project by using VCS?
The answer depends very much on how many training examples are provided, how many verbatims are then coded automatically, how high is the per-verbatim cost of human coding, whether fast turnaround of results is important to you, how accurate the resulting system is required to be, and many other variables. A detailed study of practical benefits can be found in operation can be found in Macer *et al*. (2007, p. 15).

9 Will accuracy keep increasing if I keep adding more and more training data, no matter how much I have already added?
No, it is typically the case that accuracy will, at a certain point, plateau. In general, the increase in accuracy due to the increase in the number of training examples tends to be brisker at the earlier stages, and slower at the later stages: adding ten training documents to a training set consisting of 100 documents usually brings about higher benefit than adding ten training documents to a training set consisting of 1000 documents.

10 If so, do I then run the risk of making accuracy decrease *as a result of adding* too many *training examples?*
No, it is practically never the case that there is such a risk. While it is always possible, although rare, that accuracy decreases as a result of adding more training examples (e.g. this might certainly happen in case these newly added examples have been miscoded by the human coder), these are usually local phenomena of small magnitude; in these cases, adding further training examples should bring the situation back to normal.

11 If I want to increase accuracy for a given code, I know I should add more training examples. Are positive and negative examples of this code equally valuable?
No. While it is true that both positive and negative examples are necessary, positive examples are more informative than negative examples. Just imagine a child being taught by her father what a tiger is. It is certainly more informative for the father to show the child a picture of a tiger and say to her 'This is a tiger!' than to show her a picture of a penguin and say to her 'This is a not a tiger!' So, one should strive to add *positive* examples of the code. Note that, since adding a training verbatim means telling the system, for *each* code in the codeframe, whether the verbatim has the code or not, and since each verbatim typically has only one or few out of the many codes in the codeframe, negative examples abound anyway.

12 Can I tune the internals of VCS, so as to attempt to increase its accuracy?
No, VCS does not contain manually inspectable and manually tuneable coding rules. The only way a user can improve the coding behaviour of VCS is by providing more training data (i.e. adding more training data or validating more automatically coded data). This is not a limitation of VCS, it is its beauty, since no skills are required to operate this software other than standard coding skills. The internals of VCS are already optimised based on sophisticated mathematics, and allowing the user to turn knobs would inevitably mean damage to the accuracy of coding.

Other features of VCS

Robustness to ill-formed input

One VCS that is of particular interest in customer satisfaction and market research applications is robustness to orthographically, syntactically or other type of ill-formed input. Certainly, we cannot expect verbatims to be conveyed in flawless English (or other language), since in most cases verbatim text is produced carelessly, and by casual users who are anything but professional writers. So, a system that requires verbatim text to conform to the standard rules of English grammar and to be free of typographical errors would be doomed to failure in this application context. VCS is instead devised to be robust to the presence of ill-formed input; to illustrate this, the results of Table 1 were obtained on datasets of authentic verbatim text, and are indeed fraught with ill-formed text of many types. Indeed, VCS *learns* to deal with ill-formed verbatims from ill-formed training verbatims; in other words, once the binary coders are trained from a training set that itself contains ill-formed input, they will tend to outperform, in coding as yet uncoded ill-formed verbatims, binary coders that have instead been trained on well-formed input, since, upon coding, they will encounter ill-formed linguistic patterns they have encountered in the training stage (e.g. common abbreviations, common syntactical mistakes, common typos, slang, idioms).

Dynamic estimation of current and future accuracy levels

A second VCS feature we have not yet discussed has to do with letting the user know what accuracy she can expect from VCS on a given set of verbatims, given the amount of training she has performed already, and

letting her know whether it is likely that there is room for improvement by undertaking further training or validation. Indeed, once the user submits a set of training examples, VCS generates the binary coders and returns the (F_1, PD_M, PD_A) values computed by 10-fold cross-validation on the training set; these figures thus represent an estimate (actually a pessimistic estimate, given that only 90% of the training set has been used for training in each of the ten experiments) of the accuracy that the generated binary coders will display on as yet uncoded verbatims. Moreover, VCS computes (F_1, PD_M, PD_A) by 10-fold cross-validation on 80% of the training set, and returns to the user the percentile difference between this triplet of results and the previously mentioned triplet; this difference thus represents the difference in performance that providing the other 20% of training brought about, and can thus be interpreted as the current 'accuracy trend', i.e. an indication of whether performing further training or validation is still going to bring about an improvement in accuracy, and to what degree.

Mock verbatims

Even when the training set is reasonably sizeable it often happens that, while for some codes there are lots of training examples, other codes are heavily undersubscribed, to the point that, for some of them, there may be only a handful of or even no training examples at all. For these latter codes, generating an accurate binary coder is hardly possible, if at all. In these cases, hand coding other verbatims with the intent of providing more training data for the underpopulated codes may be of little help, since examples of these codes tend to occur rarely.

In this case, VCS allows the user to define 'mock' training verbatims for these codes, i.e. examples of these codes that were not genuinely provided by a respondent but are made up by the user. In other words, VCS allows the user to provide examples of 'what a verbatim that would be assigned this code might look like'. Such a mock verbatim may take two forms: (1) a verbatim that is completely made up by the user, or (2) a verbatim that the user has cropped away from some other genuine, longer verbatim. VCS treats mock verbatims differently from authentic ones, since (i) it flags them as non-authentic, so that they do not get confused with the authentic ones for reporting and other similar purposes, and (ii) it gives them extra weight in the training phase, since it deems that their very nature makes them 'paradigmatic' examples of the codes that are attached to them.

In order to minimise the user's work in defining mock verbatims, given a training set VCS presents to the user a list of the codes in the codeframe

ranked in increasing number of associated training examples, so that the user may indeed concentrate on defining mock verbatims for those codes that are more in need of them.

Training data cleaning

In many situations it is actually the case that there might be miscodings in the verbatims that are being provided to the system for training purposes; this might be a result of the fact that junior, inexperienced coders have done the coding, that the coding had been done under time pressure, or some other reason. Of course, 'bad data in = bad data out', i.e. you cannot expect the system to code accurately if it has been trained with inaccurate data. As a result, a user might wish to have a computerised tool that helps her in 'cleaning' the training data, i.e. in identifying the miscoded training verbatims so as to be able to correct them. Of course, such a tool should make it so that she does not need to revise the *entire* set of training examples.

Our 'training data cleaning' tool returns to the user, as a side-effect of training, a sorted list of the training verbatims, sorted in order of decreasing likelihood that the verbatim was indeed miscoded. This allows the user to revise the training set starting from the top of the list, where the 'bad' examples are more likely to be located, working down the list until she sees fit. The tool works on a code-by-code basis, i.e. for any given code the tool returns a list of examples sorted in decreasing likelihood that the verbatim was indeed miscoded *for this code* (i.e. it is either a false positive or a false negative for this code). This allows the user to perform *selective* cleaning – e.g. a cleaning operation for those codes whose binary coders have not yet reached the desired level of accuracy – and forget about codes on which VCS is already performing well.

How does the tool work? The basic philosophy underlying it is that a training verbatim has a high chance of having been miscoded when the codes manually attributed to it are highly *at odds* with the codes manually attributed to the other training verbatims. For instance, a training verbatim to which a given code has been attributed and that is highly similar to many other verbatims to which the same code has *not* been attributed, is suspect, and will thus be placed high in the sorted list for that code. In other words, VCS has a notion of the *internal consistency* of the training set, and the tool sorts the training verbatims in terms of how much they contribute to disrupting this internal consistency.

Support for hierarchical codeframes

VCS natively supports codeframes structured as trees of supercodes (e.g. 'Soft drink') and subcodes (e.g. 'Coke'). At training time, all training examples of the subcodes will also be, by definition, training examples of the corresponding supercode. At coding time, a given verbatim will be fed to the binary coder for the subcode only if the binary coder for the corresponding supercode has returned a positive decision; this will ultimately bring about an exponential increase in coding efficiency, since for each verbatim to code entire subtrees will be discarded from consideration, thus allowing VCS to operate speedily even on codeframes consisting of several thousands codes.

Concluding remarks

Before concluding, note that we have not given much detail on the internal workings of VCS. This is mostly due to the fact that a description of these workings, to any level of detail, would be likely to distract the reader from understanding the important facts about VCS, i.e. how a researcher should use it and what she should expect from it. The interested reader may in any case reconstruct, if not all details, the main tracts of how VCS works by looking at the authors' published research on issues of text analysis for meaning and opinion extraction (Baccianella *et al.* 2009), learning algorithms for text classification (Esuli *et al.* 2008), and opinion mining for sentiment analysis (Argamon *et al.* 2007; Esuli & Sebastiani 2007a, 2007b). In particular, the tool used in the validation phase (see the section above, entitled 'VCS: an automated verbatim coding system) draws inspiration from basic research reported in Esuli and Sebastiani (2009a); the training data cleaning tool (see 'Training data cleaning', above) is based on the very recent work by Esuli and Sebastiani (2009b); and the support for hierarchical codeframes is based on insights obtained in Esuli *et al.* (2008) and Fagni and Sebastiani (2010).

Overall, we think that the VCS system we have presented has the potential to revolutionise the activity of coding open-ended responses as we know it today, since (i) it allows a user to autonomously create automatic coding systems for *any* user-specified codeframe and for *any* type of survey conducted (as of now) in any of five major European languages, with no need for specialised dictionaries or domain-dependent resources; (ii) it permits improvement of the accuracy of the generated coding systems almost at will, by validating, through a convenient interface, carefully selected samples of the automatically coded verbatims.

Even more importantly, for doing any of the above it requires on the part of the user no more skills than ordinary coding skills.

The experimental results obtained by running VCS on a variety of real respondent datasets confirm that the approach embodied by it is a viable one, since all these experiments were characterised by good accuracy at the individual level, very good accuracy at the aggregate level, and excellent training and coding speed.

The main challenge for the future will consist in obtaining even higher levels of accuracy, and on even more difficult types of data, such as highly problematic textual data (such as data obtained from OCR, or data from surveys administered via cellphone text messaging) or audio data (as collected in CATI).

Acknowledgements

Many people have contributed to the VCS project in the course of the past seven years. Daniela Giorgetti is to be credited with first bringing the very existence of 'survey coding' to the attention of the second author, and for first creating with him 'version 0.1' of the system, for designing and implementing which Irina Prodanof provided partial funding and for testing which Tom Smith provided datasets. Tiziano Fagni, Ivano Luberti, Fabio Nanni and Antonio Paduano have been instrumental in turning all of the ideas discussed in this paper into an industrial-strength software package. Tim Macer has been a constant source of advice, feedback and ideas. Thanks also to Charles Baylis, Lee Birkett, Rudy Bublitz, John Jennick, Dean Kotcha, Tara O'Hara, Mark Pearson, Rino Razzi, Lara Rice, Carol Sheppard, Chrissy Stevens and Rich Thoman, for useful discussions, and for ultimately contributing to making all of this happen.

References

Argamon, S., Bloom, K., Esuli, A. & Sebastiani, F. (2007) Automatically determining attitude type and force for sentiment analysis. *Proceedings of the 3rd Language Technology Conference (LTC'07)*. Poznań, PL, pp. 369–373.

Baccianella, S., Esuli, A. & Sebastiani, F. (2009) Multi-facet rating of product reviews. *Proceedings of the 31st European Conference on Information Retrieval (ECIR'09)*. Toulouse, FR, pp. 461–472.

Esuli, A., Fagni, T. & Sebastiani, F. (2008) Boosting multi-label hierarchical text categorization. *Information Retrieval*, **11**, 4, pp. 287–313.

Esuli, A. & Sebastiani, F. (2007a) PageRanking WordNet synsets: an application to opinion mining. *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL'07)*. Prague, CZ, pp. 424–431.

Esuli, A. & Sebastiani, F. (2007b) Random-walk models of term semantics: an application to opinion-related properties. *Proceedings of the 3rd Language Technology Conference (LTC'07)*. Poznań, PL, pp. 221–225.
Esuli, A. & Sebastiani, F. (2009a) Active learning strategies for multi-label text classification. *Proceedings of the 31st European Conference on Information Retrieval (ECIR'09)*. Toulouse, FR, pp. 102–113.
Esuli, A. & Sebastiani, F. (2009b) Training data cleaning for text classification. *Proceedings of the 2nd International Conference on the Theory of Information Retrieval (ICTIR'09)*. Cambridge, UK, pp. 29–41.
Fagni, T. & Sebastiani, F. (2010) Selecting negative examples for hierarchical text classification: an experimental comparison. *Journal of the American Society for Information Science and Technologies*, **61**, 11, pp. 2256–2265.
Macer, T. (1999) Designing the survey of the future. *Research*, **395**, p. 42.
Macer, T. (2000) Making coding easier. *Research*, **407**, p. 44.
Macer, T. (2002) Ascribe from Language Logic. *Quirk's Marketing Research Review*, **16**, 7, pp. 84–85.
Macer, T. (2007a) Coding-Module reviewed. *Research*, **490**, pp. 42–43.
Macer, T. (2007b) Voxco command center. *Quirk's Marketing Research Review*, **21**, 1, pp. 30–33.
Macer, T. (2008) WordStat from Provalis Research. *Research*, **508**, pp. 40–41.
Macer, T., Pearson, M. & Sebastiani, F. (2007) Cracking the code: what customers say, in their own words. *Proceedings of the 50th Annual Conference of the Market Research Society (MRS'07)*. Brighton, UK.
O'Hara, T.J. & Macer, T. (2005) Confirmit 9.0 reviewed. *Research*, **465**, pp. 36–37.
Reja, U., Manfreda, K.L., Hlebec, V. & Vehovar, V. (2003) Open-ended vs close-ended questions in web questionnaires. In: A. Ferligoj & A. Mrvar (eds) *Developments in Applied Statistics*. Ljubljana, SL: Faculty of Social Sciences, University of Ljubljana, pp. 159–177.
Schuman, H. & Presser, S. (1979) The open and closed question. *American Sociological Review*, **44**, 5, pp. 692–712.
Sebastiani, F. (2002) Machine learning in automated text categorization. *ACM Computing Surveys*, **34**, 1, pp. 1–47.
Sebastiani, F. (2006) Classification of text, automatic. In: K. Brown (ed.) *The Encyclopedia of Language and Linguistics*, vol. 2 (2nd edn). Amsterdam, NL: Elsevier Science Publishers, pp. 457–463.
Sturgis, P. (2004) The effect of coding error on time use surveys estimates. *Journal of Official Statistics*, **20**, 3, pp. 467–480.

About the authors

Andrea Esuli has been at the Italian National Research Council (the largest government-owned research institution in Italy) since 2005, first as a research associate (until 2009) and then as a researcher (since 2010). He holds a PhD in Information Engineering from the University of Pisa. He has authored more than 30 scientific articles in international journals and conferences in the fields of (among others) information retrieval, computational linguistics, text classification, opinion mining and content-based image search. He is the recent recipient of the Cor Baayen

Award 2010 from the European Research Consortium in Informatics and Mathematics.

Fabrizio Sebastiani has been a staff senior researcher with the Italian National Research Council since 2002. He is the co-editor-in-chief of *Foundations and Trends in Information Retrieval* (Now Publishers), an associate editor for *ACM Transactions on Information Systems* (ACM Press), and a member of the editorial boards of *Information Retrieval* (Kluwer) and *Information Processing and Management* (Elsevier). He has authored more than 100 scientific articles in international journals and conferences in the fields of (among others) information retrieval, machine learning, computational linguistics, text classification and opinion mining.

Address correspondence to: Fabrizio Sebastiani, Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche, 56124 Pisa, Italy.

Email: fabrizio.sebastiani@isti.cnr.it