

Bauhaus-Universität Weimar
Faculty of Media
Computer Science and Media

Towards Proofreading Using Human-based Computation

Bachelor Thesis

Teresa L. Holfeld
Born in Jena, Germany, on June 24th, 1986

Student number 60172

1. Supervisor: Prof. Dr. Benno Maria Stein
Tutor: Martin Potthast

Date of submission: April 13th, 2011

Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work.

Weimar, April 13th, 2011

.....
Teresa L. Holfeld

Abstract

The process of writing a text often requires proofreading, which is the task of detecting and correcting writing errors. There are several methods of proofreading, for instance asking friends or co-workers, using tools for automatic error detection, or consulting a professional proofreader, all involving certain costs of time or money. In this thesis we present human-based computation as an approach for semi-automatic proofreading that combines the convenience and low costs of tools for automatic error detection, and the feeling for language of human proofreaders.

In our research, we determined two interface metaphors that apply to the design of proofreading user interfaces (UIs): *editing* and *annotation*. We developed three different proofreading UIs for the application on Amazon Mechanical Turk (MTurk) that apply these interface metaphors to texts of different lengths. These UIs are: *editing a sentence*, *editing a paragraph* and *annotating a paragraph*. We further developed a UI for reviewing the results of proofreading tasks submitted by workers on MTurk.

We conducted six experiments to investigate the impact of different properties of our proofreading UIs to the proofreading results. As input texts we used two so-called English learner corpora: *English as a Second Language 123 Mass Noun Examples* (ESL123) and the *Montclair Electronic Language Database* (MELD). These are collections of erroneous essays or sentences written by English learners. Where not given, we retrieved all writing error positions and all correct versions of the texts, and used them as gold standards for our experiments. We then evaluated the results of our experiments in comparing them to this gold standard. The best experiments achieved a word-wise recall of 0.91 in error detection, and a mean BLEU of 0.67 regarding error correction. In these experiments we paid \$2 for the proofreading of 1 000 words on average, with a maximum total cost of \$12.50 per experiment. Our experiments show a maximum performance of 28.5 working hours in total. The evaluation of our experiments reveals that crowdsourcing is a cheap, fast and efficient facility for proofreading, compared to the costs for professional proofreading and to the time exposure for retrieving comparable proofreading results with traditional methods.

Contents

1	Introduction	2
2	Background and Related Work	4
2.1	Writing Errors and Proofreading	4
2.2	Automatic Error Detection	6
2.3	Human-based Computation	9
2.4	Crowdsourcing Texts	13
3	User Interfaces for Proofreading on Mechanical Turk	16
3.1	Metaphors and Design Principles for Proofreading UIs	16
3.2	Our Proofreading UI Implementations	22
3.3	A Backend UI for Result Reviewing	26
4	Evaluation of our Proofreading UIs	31
4.1	Writing Error Corpora	31
4.2	Experimental Methodology	34
4.3	Performance Measures	36
4.4	Evaluation Results	42
5	Conclusion	49
	Bibliography	51

Chapter 1

Introduction

Proofreading is an important part of writing a text, since making mistakes is nearly unavoidable. It is the task of detecting and correcting all kinds of writing errors, such as errors in spelling and grammar, or poor style and semantic errors [Naber, 2003]. However, proofreading is not cheap: if the author wants to consult a professional proofreader, he or she has to pay for this service. Prices differ according to the text's properties and negotiation. Anyway, the Society for Editors and Proofreaders suggests a minimum hourly wage of £20.25.¹ While professional proofreading services are often consulted for important texts like papers or dissertations, the proofreading of smaller and less important texts is usually done as a favor between colleagues or friends. But this, too, has its price, since they have to invest their free time.

Since the beginning of personal computing, there has also been research on automatic proofreading. This comprises spelling, grammar, and style checking, implemented by varying approaches that use dictionaries, syntactic analysis (parsing), language models, or statistical machine translation. Though, at present, few practical solutions are mature enough to be deployed in large scale, and there is no common, cheap and reliable software that incorporates all the desired functionality in a user-friendly way. Thus, writers still lack an important tool for their writing process.

In this thesis we explore a new approach to proofreading that falls right in between (semi-)professional and automatic proofreading. It is based on human-based computation, that is, using the working power of a large group of human workers and embedding it in a computational environment. By outsourcing the proofreading task to the “crowd” (crowdsourcing), we hope to achieve the following: (1) money that would be spent on the proofreading service could be saved, since crowdsourcing is much cheaper than expert labour, and (2)

¹SfEP's suggestions of minimum freelance wages can be accessed at http://www.sfep.org.uk/pub/mship/minimum_rates.asp (last access: 2011-03-20).

the time and effort of friends and colleagues is saved, since they only need to proofread a close-to-final draft.

Our working hypothesis is that the feeling for language of average native English speakers is sufficient to find and correct writing mistakes until a text conforms to common English. In order to test this hypothesis and to evaluate the feasibility and the quality of our approach, we conduct experiments on collections of error-annotated writing samples of English learners, so-called English learner corpora. We develop a user interface for proofreading that is shown to the workers on the web-based crowdsourcing platform Amazon Mechanical Turk (MTurk), and evaluate the impact of various design parameters on the quality of the proofreading results.

Chapter 2

Background and Related Work

There is much research on writing error analysis in the research field of linguistics, and much on automatic proofreading in the research field of computer science. In economics, the subject of crowdsourcing has grown in popularity during the last years, and many papers in a variety of research fields deal with MTurk as a facility for human-based computation. In this chapter we give an introduction to these background issues and related work. We elaborate on some terminology and position ourselves in the research fields of related projects.

2.1 Writing Errors and Proofreading

In this section we give a short survey of the different types of writing errors. We further describe the task of proofreading a text and distinguish it from the related task of copy editing.

Categorizing writing errors is not simple. The very same error can be categorized in different ways when seen from different angles. For example, the confusion of “then” and “than” can be classified as a phonetic spelling error as well as a grammatical confusion of prepositions. *Writing error analysis* is part of *corpus linguistics*. This is a research field of linguistics where large collections of writing samples (so-called corpora) are analyzed for writing errors, for example, for research on language learning. In this research field, there have been many attempts to set up taxonomies of error types and tag systems to annotate errors in writing error corpora properly (see section 4.1). For instance, [Lightbound, 2005] distinguishes 9 categories with up to 7 subcategories and up to 6 sub-subcategories, whereas [Granger, 2003] distinguishes 9 categories with up to 9 subcategories.

This sophisticated differentiation may be essential for a deep linguistic un-

derstanding of writing errors, but it is needlessly complicated for our purpose. In proofreading, the main concern is where errors are, and which corrections might be appropriate. Nevertheless, a simple classification is interesting to determine on which errors an error detection approach works best. Here, it is sufficient to distinguish a few categories according to existing solutions of automatic error detection. In doing so, we found the categorization of [Naber, 2003] quite appropriate. Naber distinguishes four error categories, namely spelling errors, grammar errors, style errors and semantic errors. We briefly describe these error types in the following overview.

Spelling errors are errors in orthography. This comprises so-called “typo”, the confusion of similar sounding words, and wrong case-sensitivity. Spell checkers typically detect these errors by comparing the words to a dictionary of the corresponding language.

Grammar errors comprise errors in punctuation, conjugation, gender or case, and can be subject-verb-disagreements or similar. These kinds of errors can, to some extent, be automatically detected using syntax parsing, or rule-based systems (see section 2.2).

Style errors comprise wrong collocations, and uncommon or complicated wording. Approaches for automatic style checking make use of, for instance, the probability of words given by language models, or statistical machine translation.

Semantic errors are errors in meaning or propositional logic, or statements that simply do not make sense. Grammatical sentences can have semantic errors, too (e.g. “MySQL is a great editor for programming!”) [Naber, 2003].

In everyday language the terms *proofreading* and *copy editing* are often used interchangeably when talking about manual detection and correction of writing errors. In the field of professional publishing and printing, there is a difference between both terms, though. In the publishing process, a text is written by the author, then edited by the copy editor until it is a finished typescript. The copy editor hands this typescript to the printing office, where the typesetter does the layout and hands it to the proofreader. The proofreader then finally checks the text for remaining errors. When the proofreader has eliminated all errors, the typescript gets printed.

In this context, a copy editor is employed by the publisher. The copy editor’s task is to check the correctness of content and argumentation of a text. A copy editor checks a text for the validity of facts and for the consistency of

the writing style, as well as the content for being appropriate and fitting the policy of the publisher [Butcher et al., 2006].

By contrast, a proofreader is employed by the printing office. Proofreading is the task of detecting and correcting errors originally introduced by typesetting, which was eponymous for “typo” [Wilson, 1901].

With regard to error types, copy editors are supposed to find mainly style and semantic errors, while proofreaders are supposed to find errors in orthography, punctuation and spelling, and to eliminate all errors the copy editor has missed, and thus do not alter the meaning of the text [Horwood, 1997].

Usually, publishers deal with professional authors with experience in the usage of written language. In this context, it can be assumed that the error rate is relatively low, and the distinction between copy editing and proofreading is reasonable. In contrast, we want to provide a cheap tool for less experienced authors, too. Beyond professional publishing, this distinction is not always necessary. Therefore, we refer to “proofreading” more broadly as identifying and correcting all kinds of the aforementioned errors, which may also include parts of the task of copy editing.

2.2 Automatic Error Detection

In this section we give an overview of existing approaches for automatic error detection in natural language writing. We propose a classification of various approaches, describe the principal solutions, and name some examples.

In the research field of automatic writing error detection, there are several approaches that detect errors of the first three categories described in section 2.1, that is, spell checking, grammar checking and style checking. These approaches have been developed to find either one special error type, e.g. mass noun / count noun confusions, or to cover several error categories, e.g. spelling errors as well as grammar errors and style errors.

Some of the approaches are publicly available solutions, like the *Hunspell*¹ spell checker, which is the underlying technology for OpenOffice, and the Unix commands *Style* and *Diction*² for grammar and style checking. However, there are much more solutions in this field of research, and with increasing diversity of approaches and improving results, the terms *spell checker*, *grammar checker* or *style checker* get ambiguous, since most software combine different approaches. With regard to the underlying techniques of these approaches, the following classification can be made: (1) dictionary-based approaches, (2)

¹Hunspell is released at <http://hunspell.sourceforge.net> (last access: 2011-02-26).

²Style and Diction are released at <http://www.gnu.org/software/diction/> (last access: 2011-02-26).

syntax-based approaches, (3) statistics-based approaches, (4) rule-based approaches, and (5) human-based approaches.

Dictionary-based Approaches

This is the most common concept of spell checkers: every word in the text is checked against a list of words representing the vocabulary of the particular language. If the word occurs in this dictionary, it is spelled right, otherwise it is marked as an error. Examples for spell checkers are *Hunspell* and its predecessor *MySpell*.³ Both implementations are based on *Ispell*.⁴ The Unix command *Diction* is an example for dictionary-based style checkers. Here, a text is checked to contain particular words or phrases, that are considered to be bad style. If a word or phrase matches, *Diction* displays a comment or suggestion what to write instead. The English version of *Diction* contains 685 entries, the Dutch version 339, and the German version 76 entries.

Syntax-based Approaches

The majority of approaches for grammatical error detection use syntax-based solutions. Usually, the whole text is parsed, and a parse tree is built up for each sentence, representing the grammatical structure of it. That is, the syntax of each sentence is determined. A sentence is considered ungrammatical, if building the parse tree did not succeed.

An early example is the *Critique* system, a grammar and style checker that uses parsing [Richardson and Braden-Harder, 1988]. *Microsoft Word* uses parsing for grammar checking, too, as is stated in [Helfrich and Music, 2000].

Statistics-based Approaches

These approaches use several statistical methods to detect error candidates in a text. This can be (1) the use of language models, (2) statistical machine translation, and (3) readability measures, as we will describe below.

The first uses large-scale corpora of written English that contain either plain text of written language, or are already organized in the form of n -grams. Those n -grams can be word n -grams, that is, sequences of n words of a sentence, or POS n -grams, sequences of n part-of-speech (POS) tags of the corresponding words. If the corpus does not already consist of n -grams,

³MySpell is released at <http://packages.debian.org/src:myspell> (last access: 2011-02-26).

⁴Ispell is released at <http://fmg-www.cs.ucla.edu/fmg-members/geoff/ispell.html> (last access: 2011-02-26).

they are retrieved in a pre-processing step. Then, a language model is built up from the corpus. That is, the probability for each n -gram is calculated. If an n -gram occurs often, it has a high probability to occur in natural language, if it occurs seldom, it has a low probability. N -grams with a low probability, or n -grams that just do not exist, are considered error candidates. An example for POS tagging for English text is the *CLAWS* (the Constituent Likelihood Automatic Word-tagging System) tagger. The latest version, CLAWS4, is built from the British National Corpus [Garside, 1996]. An example of a grammar checker that uses POS tagging is *Granska*, a grammar checker for Swedish text [Domeij et al., 2000]. An example for an approach that uses word n -gram probabilities is [Islam and Inkpen, 2009], where a language model built from the Google Web 1T n -gram data set is used for context sensitive spell checking. The browser and blog software plugin *After the Deadline*⁵ is a practical example for a language checking service that uses language models built from the Simple English edition of Wikipedia and Project Gutenberg [Mudge, 2010].

The second method, statistical machine translation (SMT), uses phrasal SMT paradigms to map ungrammatical phrases to their correct counterparts. This approach, as described in [Brockett et al., 2006], does not only detect error candidates but also provides suggestions for possible corrections.

The last method, calculating readability measures, should also be mentioned as an approach that applies several statistical computations. An example is the GNU command *Style*. *Style* does style checking by giving a survey of several readability measures that help the author estimating the quality of the text's style. Such readability measures are, for example, the Kincaid Formula, the Automated Readability Index, the Coleman-Liau Formula or the Flesch Reading Ease Score [Cherry and Vesterman, 1981].

Rule-based Approaches

Here, sentences are checked against a set of manually developed rules of grammar and style that use regular expressions or POS information of the text. An example is *FLAG*, an approach that uses two steps with different kinds of rules. First, it checks the text against so-called trigger rules that identify possible error candidates, then it checks these candidates against evidence rules that verify whether a candidate is a false positive or an actual error [Bredenkamp et al., 2000].

⁵After the Deadline is released at <http://www.afterthedecline.com> (last access: 2011-02-26).

Human-based Approaches

During the last years, the crowdsourcing platform *Amazon Mechanical Turk* (MTurk) has grown in popularity in the scientific community for solving human-based computation tasks. There are also some approaches that make use of MTurk for error annotation and text improvement. [Tetreault et al., 2010] describe a method for the detection and annotation of preposition errors by workers on MTurk [Tetreault et al., 2010]. As another example, *Soylent* is a word processing interface that uses MTurk for text improvement. It does text shortening and proofreading, and suggests corrections [Bernstein et al., 2010]. The approach presented in this thesis falls into this category.

2.3 Human-based Computation

In this section we describe what human-based computation is, and elaborate on the terminology. We briefly describe existing projects on human-based computation, and introduce the crowdsourcing platform Amazon Mechanical Turk.

There are a lot of things that can easily be done by humans but cannot (yet) be done by computers. Such tasks may fall into the areas of image recognition, speech recognition, and natural language processing. An example is the recognition of letters or objects in images, as in spam protection services, or for image labeling [Gentry et al., 2005]. The act of using the processing power of humans to solve such problems and to embed them in a computational environment is called *human computation* [von Ahn, 2005]. Because it evokes fewer connotations of a science fiction-like bio-computational use of the human brain (cyborgs, the Matrix, etc.), we prefer the term *human-based computation*. This term is borrowed from the research field of human-based evolutionary computing [Hammond and Fogarty, 2005], which belongs to the field of interactive evolutionary computing (a detailed survey is to be found in [Takagi, 2001]).

Human-based computation is also closely linked to *crowdsourcing*, described in more detail in section 2.4. Both terms are often used interchangeably, but while crowdsourcing is rather used with respect to economic issues, human-based computation is more often used when talking about technical aspects. In this thesis, we use the term human-based computation for the process of embedding tasks solved by humans in a computational environment. We use the term crowdsourcing for the aspect of problem solving by accumulating the results of a large group of workers.

Human-based computation is successfully used, for instance, for image la-

belonging or for digitizing books. The first can be done with so called *games with a purpose*. An example is the ESP Game, a free game where two players who are not allowed to communicate with each other are shown the same image. One player has to find a word by guessing what the other player would use to describe this picture, too. When both proposed the same word, they win points for their score. The system takes this word as a label for the image, for example, for image search. To increase the difficulty, the most common labels are taken as taboo words that the next players are not allowed to use [von Ahn and Dabbish, 2004].

A project that digitizes books with the aid of human-based computation is *reCAPTCHA*. CAPTCHAs are images that show some distorted text, often seen, for example, at web registration forms. For the purpose of spam protection, users have to typewrite these texts in order to verify that they are humans and not computer programs, such as spam bots. Computer programs still have problems recognizing distorted text, but for humans this is an easy task to solve, so CAPCHAs are able to protect web services against the misuse by spam bots. A reCAPTCHA image consists of two words. One is known by the reCAPTCHA program and verifies whether the user is a human or not. The other is a word that was not recognized by optical character recognition after scanning for Google Books. Since users have to typewrite both words, the crowd involuntarily helps digitizing old books and documents [von Ahn et al., 2008].

An alternative for implementing computer programs that include human-based computation tasks themselves is the use of external crowdsourcing platforms. Existing platforms often have a particular purpose for which they offer crowdsourcing as their solution. Most of these platforms (see table 2.1) use crowdsourcing for creative tasks (e.g. designing t-shirts like Threadless), marketing innovation and consulting (e.g. InnoCentive, and Chaordix), knowledge generation and question answering (e.g. evly, Wikipedia, and Yahoo! Answers). But most platforms do not provide a suitable way of returning the crowdsourced solutions so that they can be processed in a computational environment. There are only two platforms that support human-based computation: Amazon Mechanical Turk, and CrowdFlower, the latter of which also uses MTurk.

Amazon Mechanical Turk⁶ is a platform that can be used by both workers and so-called requesters. Requesters offer so-called HITs (*Human Intelligence Tasks*) and a payment for their completion, whereas workers earn money by doing these tasks. A reward for a HIT can be \$0.01 at minimum, and is rarely more than \$1.00 [Paolacci et al., 2010]. Requesters can review the results of

⁶MTurk can be accessed at <http://mturk.com> (last access: 2011-02-26).

their HITs, and decide afterwards which ones they approve and which they reject. Only approved work gets paid. [Horton and Chilton, 2010] estimate the median hourly rate being \$1.38. The payment is processed directly via Amazon Payments. On MTurk, requesters design HITs by themselves. They can choose from a selection of templates which can be altered to suit the needs of a given task, or design the HIT from scratch. *Apache Velocity Variables*⁷ enable batches of HITs with varying content. The requester can also determine how many workers can work on one HIT. This atomic task – the work of one worker on one HIT – is called *assignment*. Requesters can also set up qualification requirements, such as small tests, to restrict the access to their HITs to workers who have proven their abilities first.

MTurk is implemented as a web service, meaning “a software system designed to support interoperable machine-to-machine interaction over a network”, as defined by the World Wide Web Consortium (W3C).⁸ This enables requesters to integrate human-based computation tasks on MTurk into a computer program. With MTurk’s API it is possible to control the crowdsourcing process with nearly every programming language. Further, it provides command line tools to handle HITs via shell. If a requester uses the API or the command line tools, a HIT can either be designed as an XML document that conforms to the QuestionForm schema⁹ of MTurk, or as an external webpage that is then embedded into the HIT frame on the MTurk platform. Alternatively, requesters can also design a HIT via the web user interface for requesters, which is basically a rich text editor. However, the requester is always able to determine how a task is displayed to the worker, and what the return values of a HIT are.

This flexible structure for designing HITs is crucial for human-based computation, since it enables the requester to design HITs for arbitrary tasks. Popular areas of application are e.g. tagging images with the name of the product it displays, or to investigating office hours and addresses of particular offices or restaurants. Of course, MTurk can be used as well to improve some written text. Error annotation and text shortening has already been researched on MTurk; the former by [Tetreault et al., 2010], and the latter by [Bernstein et al., 2010]. We will present these papers in section 2.4. The design and implementation of our proofreading HIT is described in chapter 3.

⁷The Apache Velocity project is to be found at <http://velocity.apache.org/> (last access: 2011-03-04).

⁸The W3C definition for “web service” is to be found at <http://www.w3.org/TR/ws-gloss/#webservice> (last access: 2011-02-26).

⁹The QuestionForm schema is specified at http://docs.amazonwebservices.com/AWSMechanicalTurkRequester/2007-03-12/ApiReference_QuestionFormDataStructureArticle.html (last access: 2011-02-26).

Name	Homepage	Purpose
Creative tasks, ideas, design		
Kluster	http://www.kluster.com	Ideas, creative tasks
Idea Bounty	http://www.ideaabounty.com	Ideas, creative tasks, competitions
12designer	http://www.12designer.com	Design, competitions
VOdA	http://www.vo-agentur.de/	Ideas, creative tasks
Threadless	http://www.threadless.com	T-shirt design
DesignByHumans	http://www.designbyhumans.com	T-shirt design
Business solutions, consulting, marketing innovation		
Brainrack	http://www.brainrack.com	Business solutions
InnoCentive	http://www2.innocentive.com	Business solutions
Chaordix	http://www.chaordix.com	Consulting
Witology	http://witology.com/en	Marketing innovation, recruiting
Knowledge generation, question answering		
evly	http://www.evly.com	Knowledge, opinion
Wikipedia	http://www.wikipedia.org	Knowledge
Yahoo! Answers	http://answers.yahoo.com	Knowledge, question answering
Plash	http://ants.iis.sinica.edu.tw/plash/	Location aware services, traffic
Special target groups		
Crowdbands	http://www.crowdbands.com	Music business, music rating
TopCoder	http://www.topcoder.com	Writing computer programs
Micro-tasks, human-based computation		
Amazon Mechanical Turk	http://mturk.com	Micro-tasks, human-based computation
CrowdFlower	http://crowdflower.com	Micro-tasks, human-based computation
ClickWorker	http://www.clickworker.com	Text writing, labeling

Table 2.1: Existing crowdsourcing platforms. This table shows a selection of well-known crowdsourcing services found on the web. Platforms are categorized by their purpose.

2.4 Crowdsourcing Texts

In this section we describe what crowdsourcing is, and what its main fields of application are. We introduce two projects that apply crowdsourcing to text improvement, and briefly describe the *Soylent* word processor, the *Find-Fix-Verify* design pattern and the *TurKit* toolkit.

The term *crowdsourcing* was coined in 2006 by Jeff Howe, a journalist who writes for Wired Magazine. He became known for his eponymous article “The Rise of Crowdsourcing” [Howe, 2006b]. The article is published on crowdsourcing.com, where he later published a definition of this term:

“Simply defined, crowdsourcing represents the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call.” [Howe, 2006a]

There has been much research to prove that crowdsourcing can be used for the fields of knowledge generation (see [Surowiecki, 2005]) and finding business solutions (see [Brabham, 2008]). In his book, Surowiecki describes how the accumulated intelligence of crowds is able to outperform the knowledge of experts. Brabham—more concretely—explores the potential of crowdsourcing for problem solving. Summarized, both see crowdsourcing as a new way to solve problems for commercial issues as well as for not-for-profit issues. In doing so, the crowd can be cheaper, quicker and better than expert labour.

Yet, crowdsourcing is not the best solution for every situation. Surowiecki shows that a crowd can be good at solving problems of cognition, coordination and cooperation. *Cognition* means problems of predicting events in the future, or assuming a fact, for instance predicting election results or the number of sales of a certain product. *Coordination* problems are problems that require a group of people to communicate with each other for achieving a goal that they have in common. Examples are a group of students looking for a party, or a company looking for costumers that need their particular product. *Cooperation* means “the challenge of getting self-interested, distrustful people to work together” [Surowiecki, 2005]. Examples are the tax system, or the agreement of seller and buyer on what a reasonable price is. Surowiecki also states that the right amount of communication between the members of the crowd and the right size of it are crucial for the success of a crowdsourcing process. He says that diversity and independence are important prerequisites of a successful crowd [Surowiecki, 2005].

There are many examples showing that crowdsourcing has already been successfully applied. A popular example is Wikipedia, which in fact is a form

of knowledge generation by the crowd [Kittur and Kraut, 2008]. Another is *Yahoo! Answers*, which shows that question answering by the crowd does work well [Little et al., 2009].

Besides, a variety of crowdsourcing platforms have been developed that enable the application of crowdsourcing for various purposes via the world wide web. In table 2.1 we present a sample of 19 notable crowdsourcing platforms. The purposes and target groups of these platforms show that crowdsourcing seems to work well in the fields of creativity and innovation, business and marketing, content generation, and question answering.

It is likely that crowdsourcing is also suitable for improving texts, particularly for proofreading. Natural language is not just based on strict grammatical rules and a definite vocabulary, but is the result of the attempt of humans to communicate with each other. So language can be seen as an agreement of a large group of people to express things in a particular way, and writing as a standardized visual abstraction of it [Fromkin et al., 2003]. Since results retrieved by crowdsourcing are always an accumulation of the work of many, a crowd of people should be able to correct erroneous language in a way so that most of them would agree that it is used correctly. By crowdsourcing texts and text improvement, we assume that the accumulated orthographic knowledge and feeling for language of the crowd leads to a correct result.

In the last years, there have been some projects that successfully harnessed crowdsourcing on text improvement, proposed e.g. by [Tetreault et al., 2010] and [Bernstein et al., 2010].

[Tetreault et al., 2010] show that, in annotating preposition errors in English as a second language (ESL) writing, MTurk is as good as a trained rater (a person who detects and annotates such errors), but with lower costs and in less time. The evaluation of their experiment shows that workers achieved a mean kappa¹⁰ of 0.606 compared to the trained raters. The experiment's purpose was to show that MTurk is suitable for linguistic error annotation, but it also implies that error detection for purposes of proofreading should work, too.

[Bernstein et al., 2010] introduce *Soylent*, a word processor that embeds crowdsourcing for text shortening and proofreading. The two mentioned tasks are done by different parts of the tool, called *Shortn* and *Crowdproof*. Shortn uses MTurk to let complicated sentences be shortened and simplified. The results of the experiment show that workers shortened the texts to a length of 78% - 90% of the original. Crowdproof uses MTurk for finding and correcting errors, and found 67% of the errors in the experiment. Unfortunately,

¹⁰Cohen's kappa coefficient is a statistical measure for inter-annotator agreement, for details see [Cohen, 1960] or section 4.3.

the proofreading experiment just comprised 5 paragraphs with 49 errors altogether, which raises to question how convincing this result actually is.

However, [Bernstein et al., 2010] introduce the *Find-Fix-Verify* programming pattern for human-based computation on MTurk. It basically describes a method of chaining iterative HITs so that a proofreading experiment is divided into three steps: The detection of errors (*find*), their correction (*fix*), and the verification whether or not they were corrected accurately (*verify*). This chaining of iterative HITs is operationalized using the *TurKit* toolkit [Little et al., 2009], which was developed for simplifying the interaction with MTurk, and to enable an iteration of HITs. “Iteration of HITs” means, taking the results of one HIT and posting it again on MTurk, either as input for the same type of HIT, or as input for a different HIT, e.g. the fix and verify steps of the aforementioned design pattern.

In comparison to these experiments, the purpose of our research is not so much to show that crowdsourcing on MTurk can be harnessed for proofreading as it is to quantify how well it actually works, and which parameters work best.

Chapter 3

User Interfaces for Proofreading on Mechanical Turk

The aim of this thesis is to explore several methods of proofreading via crowdsourcing, and to figure out which of them work best. By several methods we mean different ways of proofreading, which we can study by implementing different user interfaces (UIs). With these proofreading UIs we can determine which interface metaphors and which text lengths are suitable. Therefore, we conduct experiments with proofreading UIs that implement different interface metaphors and display texts of different lengths. In these experiments, we let workers correct the same erroneous texts with different UIs. Of these texts we know where the errors are, and can then measure how many of these errors have been detected and how precisely they have been corrected. In this way we can compare the proofreading UIs and state which of them works best. In this chapter we introduce these UIs and describe their properties.

We discuss the metaphors and design principles of a proofreading UI in section 3.1. We describe our implementation of different proofreading UIs for HITs on MTurk in section 3.2. In section 3.3 we describe our implementation of a backend UI for result reviewing for MTurk requesters.

3.1 Metaphors and Design Principles for Proofreading UIs

In this section we give an introduction to interface metaphors. We recall how proofreading is done originally, in pen and paper form, and deduce metaphors that correspond to proofreading. We have a look at how existing computer-aided proofreading tools apply these metaphors to give an example of how proofreading UIs may look like. We further give an introduction to interface

design principles, and describe those that are important for proofreading UIs.

Metaphors are used in everyday language for describing abstract concepts in a more pictorial way. For example, an argument is often described with the metaphor of *war*. Expressions like claims being *indefensible* or a *weak point* of an argument being *attacked* all use this metaphor [Lakoff and Johnson, 1980]. In general, metaphors are “natural models, allowing us to take our knowledge of familiar, concrete objects and experiences and use it to give structure to more abstract concepts” [Erickson, 1990]. Applied to interface design, metaphors are models of rather abstract, technical concepts that remind the user of familiar objects or real-world concepts. A well-known example is the *desktop* metaphor. The purpose of interface metaphors is “to provide users with a useful model of the system” [Erickson, 1990]. In the following we describe two metaphors used in proofreading UIs.

The basic idea of a proofreading task is as follows: a proofreader reads a text, detects the errors, corrects them, and optionally names the error type. We can imagine how this is done when we think of a teacher who corrects the essays of his or her students. He or she would underline wrong words with a red pen, and would write the type of error alongside. Or he or she would cross out the wrong words and write the correction above. Corrections may look like shown in figure 3.1.

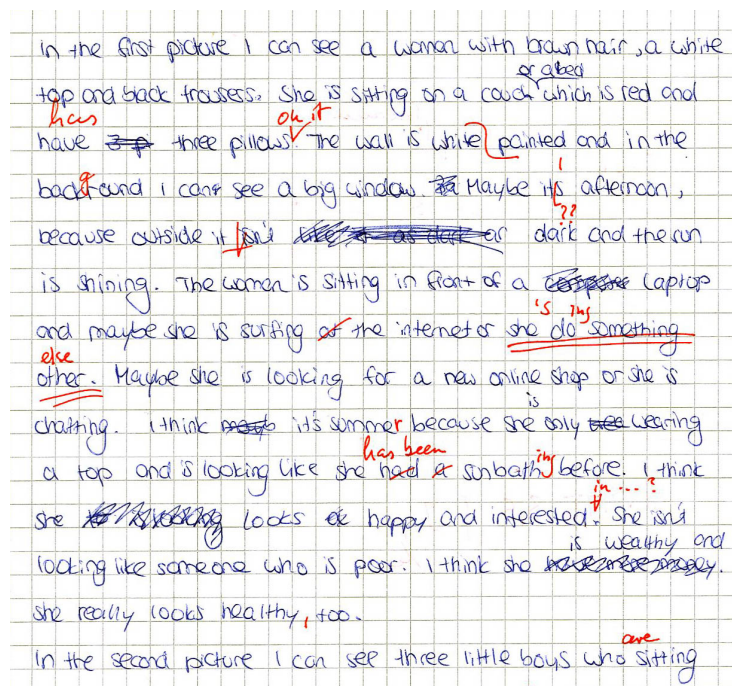


Figure 3.1: English essay, corrected by a teacher. Source: Margit Rentschler.

Here, we make a distinction between two methods, and two metaphors, respectively:

- Annotation
- Editing

Marking text (e.g. highlighting or underlining) and leaving a related comment (e.g. an error type or a short explanation) alongside can be seen as characteristics of the metaphor of *annotation*. Replacing words or passages (e.g. by crossing them out and writing a correction above), and rewriting of whole passages or sentences indicate the metaphor of *editing*.

There are some examples for existing software providing some kind of proofreading facility that apply these metaphors. For instance, Microsoft Word provides adding *comments*, and the *Track Changes* feature.¹ To add a comment, the proofreader selects a word or text passage, and then chooses the comment feature from the menu. The comment balloon appears alongside on the reviewing pane, as shown in figure 3.2.

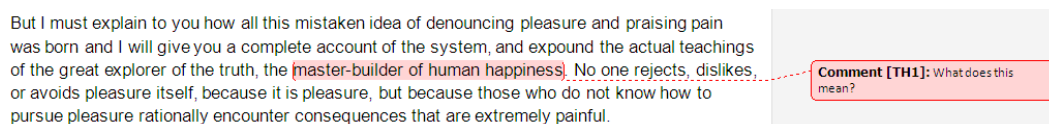


Figure 3.2: Comments in Microsoft Word.

To track changes, the proofreader selects the Track Changes feature from the menu. As long as it is activated, all changes made in the text are made visible. Deleted passages are crossed out and new passages are shown in a different color than the original text. On the left side, a vertical bar indicates that something in this line has been changed. An example of the Track Changes feature is shown in figure 3.3.

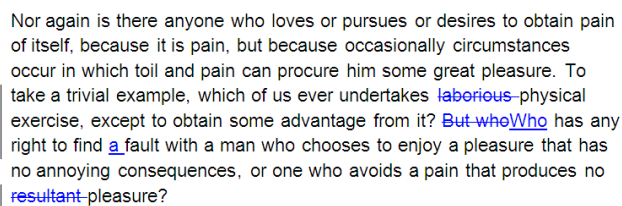


Figure 3.3: Track Changes in Microsoft Word.

¹A description of comments and the Track Changes feature can be found at <http://office.microsoft.com/en-us/word-help/CH010024383.aspx>.

The first feature, adding comments to selected text passages, uses the annotation metaphor. The second, Track Changes, uses the metaphor of editing.

PDF annotation software usually provides functionality that is similar to the comment feature of Microsoft Word. For instance, Adobe Acrobat, PDF-XChange, and PDFEdit allow the proofreader to select text in a PDF document and to comment this highlighting alongside. Additionally, unrelated comments can be placed anywhere in the document. Figure 3.4 shows a screenshot of a commented text passage in PDF-XChange Viewer.

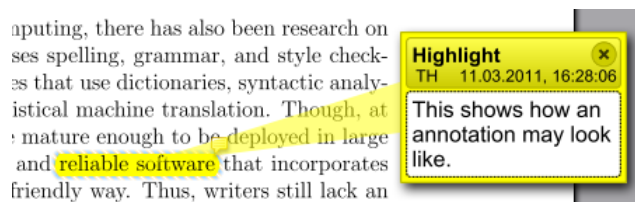


Figure 3.4: Annotated text in PDF-XChange Viewer.

As the term “PDF annotation software” indicates, these tools apply the annotation metaphor.

If we want to design proofreading HITs, we should implement these interface metaphors, too, so the worker can build on his or her experience of proofreading on paper, or with one of the aforementioned tools. We describe our implementation of both metaphors in section 3.2. In chapter 4 we evaluate the interfaces and show how well each metaphor works.

The design of HITs on MTurk is basically web design, since MTurk is a web-based crowdsourcing platform. If a worker selects a HIT, the meta information of the HIT is presented to him or her, and an iFrame displays the task that has been designed by the requester. The content of this task usually consists of a HIT description, text, images, or other media the worker has to work with, and an HTML form that the worker has to fill out in order to accomplish the task.

So, contrary to the design of websites that may comprise multiple pages, one of the challenges of HIT design is to provide all the required interaction on a single page with limited space. In addition, the goal of HIT design is to support crowdsourcing as well as possible. In crowdsourcing, the group of workers has to be diverse and big enough to achieve best results. Plus, the costs for human-based computation should be minimized. Therefore, the HIT should be assigned to many workers rather than to few, and be as small, simple, and cheap as possible. Moreover, designing a HIT is made difficult by the fact that a worker is self-interested and naturally wants to work as little as possible and to spend as little time as possible. For facing these challenges of

HIT design—(1) limited space, (2) best crowdsourcing performance, and (3) self-interest of workers—it should be designed as a minimalist user interface.

John Carroll proposed four basic principles of minimalism design in his book *Minimalism Beyond the Nurnberg Funnel* [Carroll, 1998], which are presented and applied to WebUI design in the article *An Application of the Principles of Minimalism to the Design of Human-Computer Interfaces* by JoAnn Hackos [Hackos, 1999]. These four minimalism design principles are:

1. “Choose an action-oriented approach”
2. “Anchor the tool in the task domain”
3. “Support error recognition and recovery”
4. “Support reading to do, study, and locate”

These principles are the outcomes of studies and evaluation of user interfaces of computer applications and websites. In the following, we introduce the four principles as described in [Hackos, 1999], and show how these principles can be applied to UI design of HITs.

1st Principle: Choose an Action-oriented Approach

A HIT is a user interface for a human-based computation task. According to the first principle, this means that it should support workers to do something. The basic idea of such an action-oriented approach is to “make critical actions immediately apparent when the users enter the interface” [Hackos, 1999]. Therefore, the aim of good HIT design should be that workers are able to start working on a HIT immediately after accepting² it. That is, the action the worker is supposed to do should be obvious and self-explanatory in a way that the need for instructions or user guides is minimized.

2nd Principle: Anchor the Tool in the Task Domain

In working on a HIT, a worker always has a goal in mind. This is probably earning money in the first place, and should coincide with the task he or she is supposed to do. In a minimalist interface design, the designer should “ensure that the users’ goals are well understood by the interface designers so that the interface makes a clear connection between the users’ goals and the tasks required to achieve the goals” [Hackos, 1999]. Applied to the design of HITs,

²On MTurk, workers are shown a preview of a HIT with disabled functionality first. If they want to work on it, they have to accept the HIT, which is done by clicking the “Accept HIT”-button.

this leads to two challenges that the requester has to face. The first is to make the task of the HIT the workers' goal. That is, the requester has to make sure that title, description and interface of the HIT are informative and obvious enough that workers know what to do in order to earn money. The second challenge is to support workers to achieve this goal. Action-oriented design should "build on the users' prior skills, knowledge, and experience" [Hackos, 1999]. For the proofreading task this means the interface should be implemented so that it reminds workers of proofreading. This can be achieved by applying one of the aforementioned proofreading interface metaphors, so workers would probably be able to guess what to do in order to annotate and correct errors.

3rd Principle: Support Error Recognition and Recovery

In the best case, there is nothing that could go wrong in using a proofreading interface. In reality, errors still happen, either due to weak spots in the design, or due to wrong input by the user. The idea of the third principle is to "assist users in preventing the error in the first place" [Hackos, 1999]. Considering the design of HITs, this also means to prevent workers' submissions from being rejected, since the rejection of assignments can be seen as the consequence of an erroneous solution. Workers should be prevented from submitting empty values, for example, with a JavaScript that disables the submit button unless the worker has changed anything or filled in all required form fields. Plus, erroneous input by the worker should be prevented. Here, default values and completion proposals are helpful. The worker has to know what is wrong, so the designer should "create informative, helpful, and courteous error messages" [Hackos, 1999]. For instance, if the submit button is disabled, it should be labeled with a short note what the worker has to do in order to submit the HIT.

4th Principle: Support Reading to Do, Study, and Locate

This principle is about user guides and help instructions. In HIT design, these can be much shorter and less complex than in application interface design, due to the small size of the task. But it still should be mentioned, since HITs often provide a short task instruction. Most important is that "user guides should be as brief as possible" [Hackos, 1999]. A proofreading HIT just requires a short description in a few sentences, or even less, in case the interface is designed to be self-explanatory. If the task itself is not obvious on the first glance, a short visual example (e.g. an image, or an animation) may be helpful for the worker to understand what to do.

In summary, good HIT design needs (1) a minimalist and preferably self-explanatory interface that applies the appropriate interface metaphor, (2) mechanisms that prevent errors, and wrong or missing input, and (3) minimized but meaningful instructions.

3.2 Our Proofreading UI Implementations

In this section we describe the implementation of three different UIs of proofreading HITs. We show how we implemented the interface metaphors of annotation and editing, and how we applied the minimalism design principles to our proofreading HITs.

In the previous section we determined two interface metaphors that are suitable for proofreading, namely annotation and editing. To test the impact of these metaphors to proofreading results, we implemented two different proofreading UIs, each applying one metaphor. We further want to test the impact of the length of the text that workers have to correct, so we implemented two versions of the UI that applies the editing metaphor for different text lengths. In the end, we had three UIs that we call after their interface metaphor and text length: *annotating a paragraph*, *editing a paragraph*, and *editing a sentence*. In the following, we present each implementation in more detail.

Annotating a Paragraph

The metaphor of annotation means the action of highlighting some wrong words in a text and providing a comment alongside that contains an error type, an explanation and/or a correction. The goal of our proofreading UI that applies the annotation metaphor is to have all errors detected, and therefore all wrong words highlighted. Furthermore, the best possible correction proposals should be obtained.

On electronic files, the most natural action is probably to select erroneous text passages with the mouse. In Microsoft Word and most PDF annotation software, the user has to click on a menu or the highlighted text to open a comment balloon, before he or she can write a comment. This is reasonable since word processors and PDF annotation software support a variety of actions other than proofreading. For example, in Microsoft Word, you can format text after highlighting it, or in annotation software, the annotator may wish to just highlight some text without commenting it.

In our case, the worker should only have one option after highlighting: writing a correction. According to the first minimalism design principle, the action

should be instantaneously available to the user. So, the comment input field should appear immediately after highlighting, and place the worker's cursor right into it. Figure 3.5 shows a screenshot of our proofreading UI after the user selected some text.

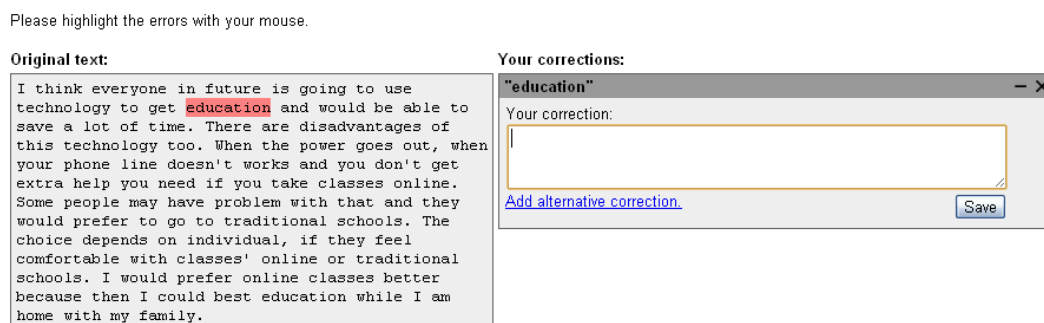


Figure 3.5: Proofreading UI for annotating a paragraph.

We implemented this interface as follows: The box that contains the erroneous paragraph consists of two `div` boxes, one superimposed on the other, with exactly the same content, height, width and position. The only difference is that the background of the upper one is transparent, and the background of the lower one is grey. In this way, it just looks like one single box, but is indeed relevant to the highlighting process, as described below.

If the user selects some text with the mouse, a JavaScript catches the `onMouseUp` event, and creates a JavaScript `Range` object out of the selection. Then, the start and end character position and the string representation of the selected text are retrieved. Each is saved as hidden input field values of the HTML form. With this information the highlighting is done: `span` elements surround the selected text passage at the respective start and end positions in the lower `div` box so that it is highlighted with a particular background color via CSS. Here, the reason for the use of two superimposed `div` boxes gets clear: The upper box represents the text without any `span` tags, which is important for retrieving the correct character positions of the `Range` object. Those `span` elements just occur in the lower `div` box, where the highlighting actually happens. Users can just select the text of the upper `div` box, so the selection will never contain any `span` tags, which might cause problems otherwise.

But placing the right opening and closing `span` tags in the text is not trivial. Our goal is to place highlightings anywhere in the text, no matter if they overlap each other, and to have the active highlighting (the one related to the annotation the user is currently editing) in a brighter color. This may look as shown in figure 3.6.

Please highlight the errors with your mouse.

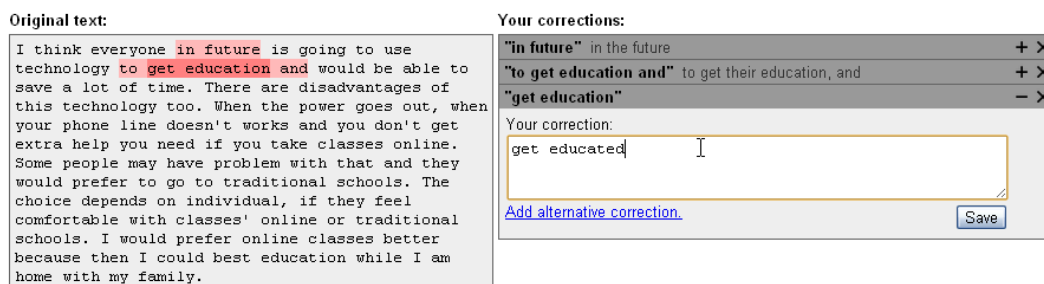


Figure 3.6: Proofreading UI for annotating a paragraph, with overlapping highlights.

So, when a user selects the same text passage several times, and selections overlap each other, the placement of `span` tags gets difficult. For solving this problem we came up with a solution that we call the *character mask*. This is understood as a JavaScript array with the same length as the number of characters in the text. Each character of the text is mapped to the value of the array at its corresponding index. Initially, each value of the array is 0. If the user selects some text, the values at the corresponding indices of the selected characters are incremented. Each time the highlighting changes, the character mask is parsed. If an array field has a different value than its predecessor, the respective closing and/or opening span tag is placed in the text at this index.

If no text is highlighted, there is an empty correction box on the right side with a short note that the user should select some text with the mouse, to encourage the user to do the right thing.

Editing a Paragraph

The metaphor of editing means to change text to eliminate the errors it contains. The goal of a proofreading UI that applies this metaphor is to have an error-free text in the end. In pen and paper form this is done by striking out wrong passages and writing the right versions above. Though, if a text contains many errors and requires heavy correction, the preferred method may be rewriting of whole passages.

In word processors, this can be done by just editing the text, that is, changing the text so that wrong words are replaced with their right counterparts. This is what is done in every text editor. The most simple form of providing a simple text editor in a WebUI is an HTML `textarea` with a predefined value that contains the text. We use this as our proofreading UI, as shown in figure

3.7.

Edit the text and correct all errors and passages with bad style.

```
I think everyone in the future is going to use technology to
get education and would be able to save a lot of time. There
are disadvantages of this technology too. When the power goes
out, when your phone line doesn't work and you don't get
extra help you need if you take classes online. Some people
may have problem with that and they would prefer to go to
traditional schools. The choice depends on individual, if they
feel comfortable with classes' online or traditional schools.
I would prefer online classes better because then I could best
education while I am home with my family.
```

Figure 3.7: Proofreading UI for editing a paragraph.

One thing that is not trivial in `textareas` is to adjust the height dynamically to its value. Because for each HIT of this UI a different text is loaded dynamically into the `textarea`, we need to alter the height of it dynamically to avoid having scroll bars at its side. We solved this problem with a hidden `div` box, in which the text is loaded, too. We retrieved the height of this `div` box and set the height of the `textarea` to this value, plus some additional space so that scroll bars do not appear even if a user introduces corrections that are a bit longer than the original.

We meet the fourth minimalism design principle with keeping instructions short and simple. The third principle is met by adding a JavaScript that prevents users from submitting the text without having made any changes. This JavaScript checks if the text is equal to the original value. If it is, this means that the worker has not changed anything, so the submit button is disabled and labeled with the note: “First correct the text”.

Editing a Single Sentence

As its name implies, the purpose of our third UI is to proofread a single sentence. Here, users do not edit the text itself, but provide a rewritten version of the sentence in a separate input field. This is in order to provide an even higher degree of freedom, and to encourage users to revise the whole wording. Additionally, users are asked to indicate what types of errors they found. Figure 3.8 shows how this interface looks like.

To meet the 1st minimalism design principle, the user can instantaneously access the task. Short descriptions instruct the user what to do in order to accomplish the task.

Each of these three UIs provides a different means for proofreading. To figure out which UI leads to best proofreading results, we did experiments and compared their outcomes, as described in chapter 4.

Original sentence:

These knowledge are extremely useful, can help us to look after the body, causes these tendency not to be able to turn the disease.

Your proofreading task:

Which type(s) of error does the original sentence contain?

Your corrected version of this sentence:

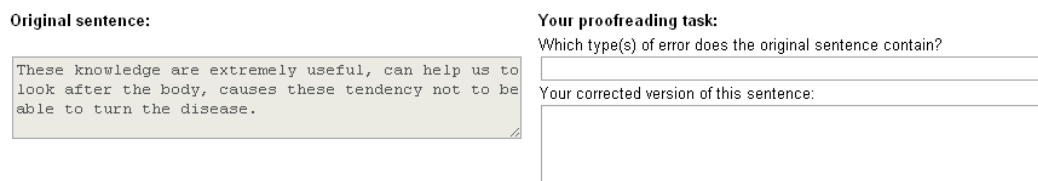


Figure 3.8: Proofreading UI for editing a sentence.

3.3 A Backend UI for Result Reviewing

Our UIs are implemented for an input text of the size of a sentence up to a paragraph. If these UIs are used to proofread text of a size that is significantly longer, the text is split into several segments that meet the required length. Each segment is used as a HIT's input text, so the whole original text is posted on MTurk as a batch of HITs. To achieve best crowdsourcing results, each HIT should be assigned to multiple workers. For example, to gain conclusive results in our experiments (see chapter 4) we had each HIT be accomplished by up to 10 workers. So, the longer a text gets and the more conclusive results should be, the more the number of assignments increases.

Requesters get the results of accomplished HITs as a table, either on the web interface of the MTurk platform, or as a CSV document. Each value submitted by a HIT's HTML form field results in a cell of this table, and each assignment results in a row. As may be imagined, this table might soon lose clarity. For example, the HTML form of our *annotating a paragraph* HIT does submit each selection the worker made, each position of a selection, each correction proposal, and a flag whether it was saved or deleted. So, result tables easily get big, complex and confusing.

Nevertheless, a requester has to review each assignment in order to approve or reject it, which decides whether or not a worker gets paid. To make this task feasible, we developed two reviewing UIs as backend interfaces for requesters: one for the *annotating a paragraph* HIT, and another for the *editing a paragraph* HIT. These reviewing UIs provide a clear presentation of the results of a batch of HITs, and a facility for quick reviewing.

A Reviewing UI for Annotating a Paragraph

The *annotating a paragraph* UI produces the most complex results, which include:

- HIT ID
- assignment ID

- worker ID
- original text
- the text of each selection
- the position of each selection
- the state of each selection (if it was saved or deleted by the worker)
- the text of each selection's correction (can be more than one)
- the state of each correction (saved or deleted)

We obtain this information from an MTurk results CSV document, and generate a UI based on HTML, CSS and JavaScript that adopts the look of the proofreading UI. It consists of two columns: on the left-hand side the erroneous text is displayed, and on the right-hand side the corrections are displayed as folding boxes. In addition, each selection and each assignment has a button for approving or rejecting it. A button on the bottom of the reviewing UI then generates the CSV the MTurk API takes as input to process approvals and rejections of assignments.

The results of all assignments can be arranged in different ways: (1) showing each text with all corrections that had been introduced to it, or (2) showing all assignments of a text separate from each other, or (3) showing the assignments of a particular worker. We call the first *document view*, the second *assignment view* and the third *worker view*. In our reviewing UI the user can switch from one view to another anytime.

Figure 3.9 shows a screenshot of a result visualization of one of our experiments with the *annotating a paragraph* UI. Here, each text is shown with the selection of all its assignments. Like in the proofreading UI, selections are highlighted in the text, and annotations are displayed as boxes at the right hand side. These boxes contain the correction proposals made by the worker. Assignment ID and worker ID are hyperlinks to the respective assignment in the assignment view or worker view. With clicking on the green tick or the red cross, reviewers can approve or reject a selection with its corrections. If the majority of selections of an assignment have been approved, the assignment is approved and hence the worker gets paid, otherwise the assignment gets rejected leaving the worker without payment.

Highlightings in the text get brighter the more workers annotated a given text segment. A pale red means a segment was just found erroneous by only one worker, whereas a segment in bright red indicates that all workers found the text erroneous. Hence, the brightness of a highlighting is an indication of

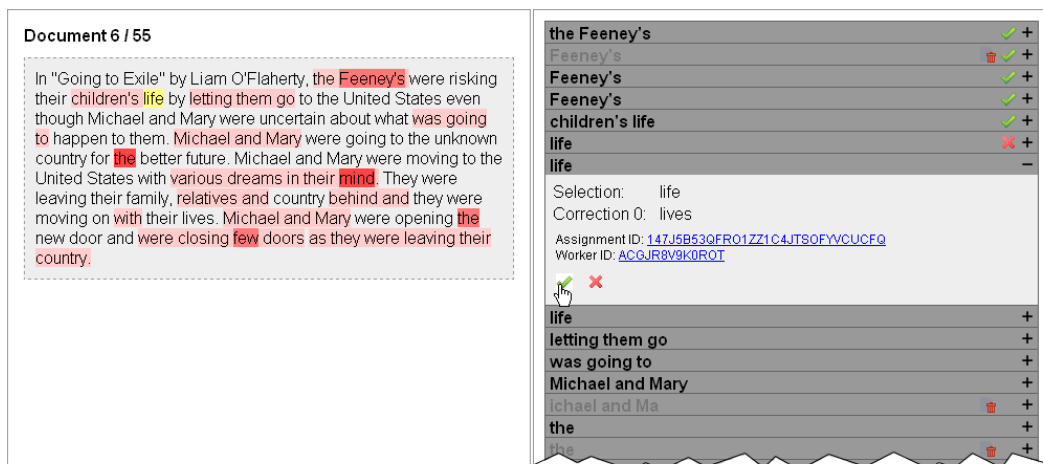


Figure 3.9: Reviewing UI for annotating a paragraph, document view.

error probability: The more workers annotated a text passage, the higher is the probability that it contains an error.

Figure 3.10 shows results in the assignment view. Here, for each assignment each selection with its corrections is shown. The worker ID is linked to the worker view that shows all the worker's assignments. An assignment can be approved or rejected by clicking on the green tick or the red cross below the text.

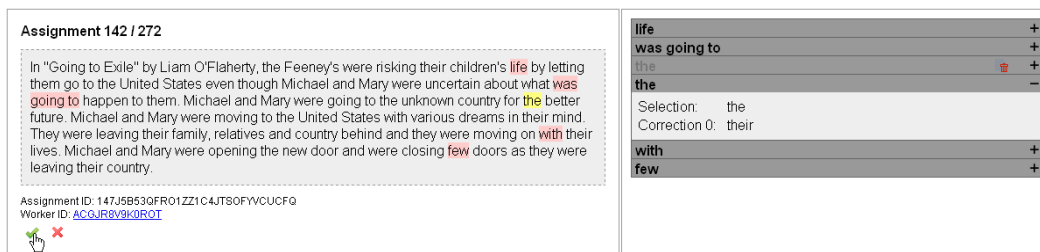


Figure 3.10: Reviewing UI for annotating a paragraph, assignment view.

Since the worker view is just a selection of the assignments that are related to the selected worker, the worker view looks the same as the assignment view. As an additional feature, reviewers can block the worker whose assignments are currently displayed in the worker view. If a worker is blocked, he or she is not allowed to work on the requester's HITs on MTurk anymore. In this reviewing UI all assignments of a blocked worker are rejected.

Our Reviewing UI for Editing a Paragraph

Compared to the results of the *annotating a paragraph* HIT, the results of the *editing a paragraph* HIT comprise much fewer values. The important ones are the original text and the corrected text. In the reviewing UI, for each assignment the original text is placed on the left hand side, while the corrected text is placed on the right hand side. For displaying differences between the text, the Myers Difference Algorithm [Myers, 1986] calculates all *deltas*. A delta represents a difference in the text, and provides information about the position of the change, and if it was a deletion, an insertion, or a substitution. To make differences visible at first glance, all deltas are highlighted in both texts. Deletions are red, insertions are green, and substitutions are yellow. Figure 3.11 shows a screenshot of such a comparison of an original and a corrected text.

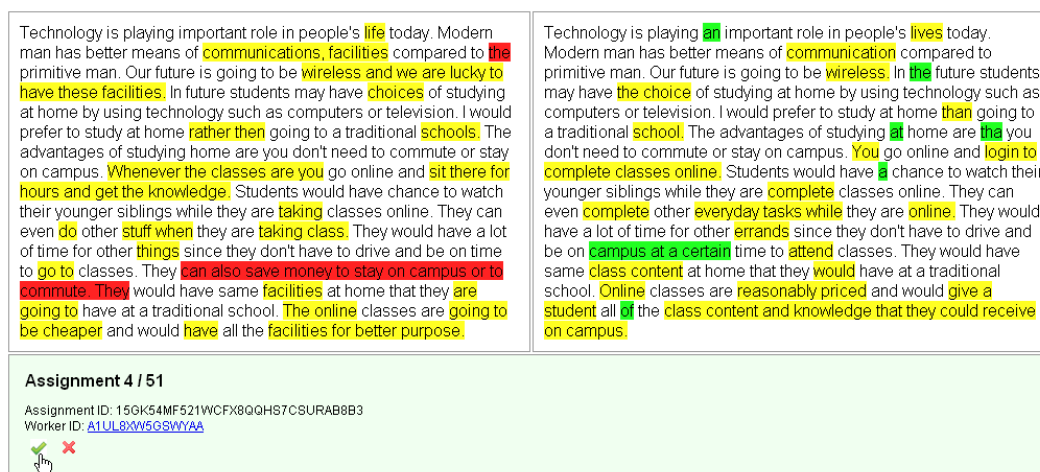


Figure 3.11: Reviewing UI for editing a paragraph.

Each assignment can be approved or rejected by clicking the respective buttons, similar to the former reviewing UI. Likewise, there's a worker view available by clicking on the worker ID below an assignment. In the worker view, all assignments of the respective worker are shown, and a feature is given to block the worker.

In doing our experiments we had to review 54 HITs with 5 assignments each, resulting in a total of 270 assignments, several times for each proofreading UI. After that experience, we are convinced that doing proofreading with MTurk does not work without a reviewing UI that represents results in a clear and revealing manner. This is not only important for deciding which workers get paid and which do not, nor only for evaluating results of our experiments,

but simply for making use of the proofreading results, too. No one can benefit from proofreading via crowdsourcing without an insightfully visualized accumulation of the annotations or edits made by the crowd. We made a first attempt to visualize results so that they get reviewable, but we think that there is much more potential for future work in developing suitable UIs that allow for even easier and quicker reviewing, and helping the writer decide even better which corrections are appropriate and which are not.

Chapter 4

Evaluation of our Proofreading UIs

Once we have implemented our proofreading UIs, we can conduct experiments to determine which UI is best suitable for proofreading. This means, we figure out which UI produces the best results—either by comparing results to each other, or by comparing results to a gold standard. For these experiments we distinguish the input parameters of (1) proofreading UI, (2) input text, and (3) required worker qualification, and the output parameters of (1) error positions, and (2) correction proposals. By varying the input parameters in our experiments, we can keep under review how the output parameters react. This gives us the chance to study the qualities of proofreading via crowdsourcing, and to make findings that are important for both its further development and its application.

In this chapter we describe what our input texts are (section 4.1), how we conducted our experiments (section 4.2), how we measure the quality of results (section 4.3), and what the evaluation results of our experiments are (section 4.4).

4.1 Writing Error Corpora

In this section we describe the input texts that we used for our experiments. We describe our source for those texts—English learner corpora—and elaborate on the properties of the corpora we used to conduct our experiments.

For our experiments we need a collection of texts that contain errors of which we know where they are located. We use these texts as input, and let them be corrected by workers on MTurk. Then, we analyze the results and see how many errors have been detected, so we can assess the quality of

proofreading. These texts should contain spelling errors, grammar errors, and style errors as well as semantic errors. Ideally, they are already annotated, so that we know where the errors are, and what their corrections are.

This is the case with so-called English learner corpora. These are collections of writing samples (mostly essays) by English learners, that were originally collected for linguistic research on learning English language, namely writing error analysis and studies of the influence of first languages on learner errors [Leacock et al., 2010]. For this purpose, many corpora are annotated with so-called *error tagging systems*, which describe positions and types of errors, and provide a correction proposal. These corpora contain errors that have been naturally introduced, and therefore represent a sample of various naturally distributed error types. In contrast to artificially generated errors, this is a comparatively good source for validating our proofreading system for real-world usage.

There is a variety of English learner corpora with essays written by English learners with different first language backgrounds. Over the years, there have been several attempts to create more or less complete lists of existing English learner corpora, for example, by [Leacock et al., 2010] (pp. 27), [Pravec, 2002], and [Tono, 2003]. But not all of these corpora are available or suitable for our experiments. Many of them are not error annotated, and most error tagging systems are very sophisticated. Those tagging systems are described in more detail, for instance, in [Díaz-Negrillo and Fernández-Domínguez, 2006]. To give an example, figure 4.1 shows an error annotated sentence of the Tswana Learner English Corpus (cf. [Rooy and Schafer, 2002]).

```
<p> Poverty is (GAA)# %% 0 $a$ (WM)# %% 0 $situation$ (LS)# %%  
whereby $in which$ (GAA)# %% 0 $the$ standard of living is low or  
when the physical Quality of (FS) %% life $Life$ Index < this looks  
like a technical term to me, hence I have gone for capitalization  
throughout DL> (GVN)# %% are $is$ not (LS) %% satisfied $achieved$  
(QM) %% 0 $. $ (LS)# %% eg $Constituents$ of (GAA)# %% 0 $the$  
physical Quality of (FS) %% life $Life$ Index are health, (FS) %%  
Nutrition $nutrition$ and employment.
```

Figure 4.1: Error annotated sentence of an essay of the Tswana Learner English Corpus.

Since we neither need that much annotation information, nor distinguish between so many error types, we had to extract the information we actually need (error positions and error types, as described in section 2.1). To keep our pre-processing efforts within limits, we used two relatively small corpora: (1) the *English as a Second Language 123 Mass Noun Examples* (ESL123), which contains no error annotation but is small enough for a subsequent manual

annotation, and (2) the *Montclair Electronic Language Database* (MELD), that contains error annotations that just provide information about the errors' locations and possible corrections.

ESL 123 Mass Noun Examples

The ESL123 corpus consists of 123 sentences found on the world wide web. The sentences were collected during research in automatic writing error detection using phrasal statistical machine translation, see [Brockett et al., 2006]. They contain grammar errors in the form of mass noun and count noun confusion. An example for a sentence of this corpus were the mass noun “knowledge” was confused with a count noun is:

“We are more exposed to all knowledges of the World.”

The ESL123 corpus consists of the erroneous sentences and proposals for their correct counterparts. It is not error annotated, so we did an error annotation subsequently. According to our analysis, the original sentences of this corpus comprise 4 to 38 words, and 15 words on average. They consist of 1 813 words in total, 358 of which contain errors, resulting in an error ratio of 19.75%.

[Brockett et al., 2006] state that 110 of the sentences contain mass noun / count noun confusions, and 13 of them contain no errors. In our subsequent error analysis we just found 4 sentences being entirely without any errors, and 62 containing mass noun / count noun confusions. We further found that they contain all kinds of errors, comprising spelling errors, style errors, semantic errors, and multiple kinds of grammar errors, like wrong articles, subject-verb-disagreements, and others.

We used all 123 sentences of this corpus as input texts for our experiment with the *editing a sentence* UI.

Montclair Electronic Language Database

The MELD corpus consists of 18 essays written by learners of the English language that address two kinds of topics: (1) a debate essay about computer-aided learning as a replacement for traditional schools, and (2) a review of the short story “Going into Exile” by Liam O’Flaherty. The corpus is manually annotated with a simple tagging system that provides information about error positions and correction proposals [Fitzpatrick and Seegmiller, 2001]. Due to this manual annotation, some errors have been introduced to both annotation

tags and correction proposals. We revised the annotation, so that these annotation errors were eliminated, and we could automatically generate original and corrected versions of the texts.

We then divided these essays into 54 paragraphs, the length of which is 14 to 418 words, and 124 words on average. The corpus is composed of 6 659 words in total, 503 of which are errors, resulting in an error ratio of 7.55%. Since the essays were written by all levels of second language learners of multiple first language backgrounds, the paragraphs comprise various kinds of writing errors (mainly spelling, grammar, and style errors), and differ in their error ratio. An example for a paragraph (original version) of the MELD corpus is:

“Also if you study at traditional school Montclair State University you will be able to make friends and to have a fun school, if you make friends and the school is fun you will be able to finish school instead if you stay at home you don’t be able to make friends.”

We used the original versions of these paragraphs as input texts for our experiments with the *annotating a paragraph* UI and the *editing a paragraph* UI.

4.2 Experimental Methodology

In this section we present our experimental methodology. We specify our input and output parameters, and introduce our experiments that we conducted to evaluate our proofreading UIs.

We ran six experiments with different proofreading UIs on MTurk. We extracted the original, unannotated versions of the texts from the writing error corpora described in section 4.1 as input for our HITs, and had workers correct them. Each Hit was assigned to and proofread by multiple workers.

After all assignments were submitted by the workers, we downloaded the results as a CSV file. We parsed the results CSV and visualized it with our reviewing UI. With this UI, we approved or rejected each assignment, and hence decided whether to pay the respective worker or not. Then, we compared the results to the gold standard, and applied the statistical measures as described in section 4.3. The results of the evaluation of these experiments are presented in section 4.4.

With our experiments we want to research the impact of multiple factors on the quality of proofreading via human-based computation. In doing so, we want to figure out (1) which factors lead to better results compared to others, and (2) which factors lead to results that are more approximate to our gold

standard. Therefore, we have to conduct several experiments with different input parameters, and evaluate the output parameters.

Our input parameters are:

- Proofreading UI with different
 - Interface parameters: Annotation, Editing
 - Supported text lengths: Sentence, Paragraph
- Different levels of the workers' qualifications, such as
 - Being a U.S. resident
 - Having an approval rate¹ of at least 95%
 - No qualification

The output parameters of our HITs are:

- Original text
- Annotations, containing
 - error position
 - selected text
 - correction proposals
- Corrected text

On these output parameters we apply the performance measures described in section 4.3. With these measures we want to estimate which input parameters have a better or worse impact on proofreading results.

We conducted six experiments, each with a varying input parameter. Table 4.1 gives an overview of these experiments and their parameters. Herein we will reference the experiments with their IDs given in this table.

We expect that experiment #6 leads to better results than the others, even if it has less qualification requirements than experiments #4 and #5. This is because the number of assignments is crucially bigger, and since crowdsourcing works best if the group of workers is large and diverse, a proofreading HIT assigned to many workers is likely to lead to better results than one assigned to a few qualified workers.

[Kosara and Ziemkiewicz, 2010] state that 73% of workers are from the USA, so some HITs are probably solved by workers with English as a second language. We expect that the access restriction to U.S. residents as a qualification in experiment #5 leads to better results than the preceding, since here the HITs are mostly proofread by native speakers.

¹A worker gets an approval if an assignment he worked on gets approved by the requester.

ID	Corpus	Proofreading UI	qualification	Assgm. per HIT
#1	ESL123	Editing a sentence	none	3
#2	MELD	Editing a paragraph	none	5
#3	MELD	Annotating a paragraph	none	5
#4	MELD	Annotating a paragraph	95% approval rate	5
#5	MELD	Annotating a paragraph	U.S. resident	5
#6	MELD	Annotating a paragraph	none	10

Table 4.1: Our experiments on MTurk

4.3 Performance Measures

In this section we introduce the measures we used to assess the results’ quality of our proofreading experiments. We compare the results of both error detection and error correction to our *gold standard* (the annotated English learner corpora mentioned in section 4.1) and applied statistical measures on it. For error detection, we use the measures of *precision*, *recall*, and *F-measure*. For error correction, we used the *Levenshtein distance* algorithm to calculate the edit distance. We further introduce two statistical measures that were developed for fields of application other than proofreading, and apply them to the measurement of proofreading results. These are *Cohen’s kappa* for error detection, and the *Bilingual Evaluation Understudy* (BLEU) for error correction.

Measuring the quality of proofreading is not trivial. It is indeed questionable if this quality is measurable at all. Natural language is ambiguous, and there will always be different opinions on what is common language usage and what is not. When we let multiple workers proofread the same text, we will most probably get different error positions and correction proposals that indeed are valid results. For example, when workers edited the erroneous sentence

“And we can learn many knowledge or new information from TV.”

from the ESL123 corpus, we got three different results:

Worker 1: “We can learn more from the TV.”

Worker 2: “Watching television provides both increased knowledge and new information on many subjects.”

Worker 3: “We can also gain much new knowledge and information from the television.”

In comparison, the correct sentence of our gold standard was:

“And we can gain a lot of knowledge and new information from TV.”

So, if we want to compare the results of our experiments to our gold standard, we have to be careful with the interpretation of measurements. For example, a low precision does not indicate poor proofreading results. It is even possible that a correction is better English than our gold standard. If we wanted to measure “quality” per se, we had to rate each single result manually. For our evaluation results and further discussion on this topic, see section 4.4.

If we want to evaluate our proofreading UIs, we need to measure the output parameters of our experiments, as mentioned in section 4.2. In doing so, we have to distinguish between evaluating error detection and evaluating error correction. For error detection we need error positions, for error correction we need correction proposals. While these are given in the results of the *annotating a paragraph* UI, we need to subsequently calculate the edit difference in the results of the *editing a paragraph* UI and the *editing a sentence* UI. We do this by using the Myers Difference Algorithm [Myers, 1986]. From this algorithm we can retrieve the changes made to the text, and information about their position, the changed text, and if the edit operation was a deletion, an insertion, or a substitution. We take these positions as error positions, and the texts of changes as correction proposals.

Measures for Error Detection

If we compare the error positions of the results of our experiments to the gold standard, we can apply the statistical measures *precision*, *recall*, and *F-measure*. Given that both gold standard and workers’ results are, in a broader sense, ratings of humans that tell if words (or sentences) are erroneous or not, the application of a measure of inter-rater agreement might be interesting. We use *Cohen’s kappa* to measure this agreement between gold standard and error detection.

We can apply these measures in multiple levels of granularity. In grammar checking, a *sentence-wise* measurement is reasonable, since in most cases it is sufficient to determine whether a sentence is grammatical or ungrammatical. In style checking and spell checking, it is important to have a more precise information about the error location within the sentence, though. So, a *word-wise* measurement is suitable here. We will apply both levels of granularity in the evaluation of our experiments. We will not apply a *character-wise* measurement, since it is not our ambition to detect wrong letters within words, but

rather to tell wrong words and right words apart. Plus, a character-wise level would bias precision and recall, since it would introduce an unwanted weight of faultiness of a word according to its number of characters. An exception is the Levenshtein distance (see below), which works on character-level.

In the following, we will describe the aforementioned statistical measures and their calculation. All descriptions are made on the word-wise level. The calculation on the sentence-wise level works analogously.

When calculating precision and recall, we compare the errors found by workers in our experiments, herein called *found errors*, to the errors in our gold standard, herein called *gold errors*. Found errors that match gold errors are *true positives*. Words that are not identified as errors by workers, and are not marked as errors in the gold standard either, are *true negatives*. Found errors that are not marked as errors in the gold standard are *false positives*, while gold errors that have not been identified as errors by workers are *false negatives*. To give an example, figure 4.2 shows a visualization of the comparison of gold errors to found errors.

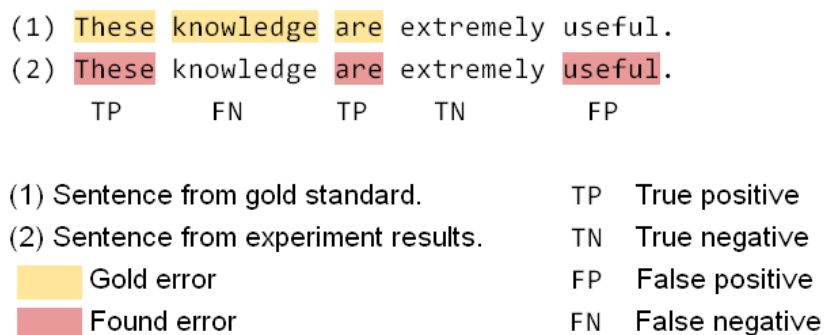


Figure 4.2: Comparison of gold errors to found errors.

In this figure, the original sentence “These knowledge are extremely useful” from the ESL123 corpus is compared to the correction: “This knowledge is extremely beneficial”. Gold errors are highlighted in yellow, and found errors, retrieved by the edit difference, are highlighted in red.

Definitions

Let E_G be the set of all words being gold errors e_g , and E_F the set of all words being found errors e_f , with E_G^C and E_F^C being their complements. The set of true positives TP , the set of true negatives TN , the set of false positives FP , and the set of false negatives FN are then defined as follows:

$$TP = E_G \cap E_F$$

$$TN = E_G^C \cap E_F^C$$

$$TP = E_F \setminus E_G$$

$$TP = E_G \setminus E_F$$

Precision

Precision describes the ratio of the size of the set of true positives to the size of the set of found errors.

$$precision = \frac{|TP|}{|E_F|} \quad (4.1)$$

Recall

Recall describes the ratio of the size of the set of true positives to the size of the set of gold errors.

$$recall = \frac{|TP|}{|E_G|} \quad (4.2)$$

F-measure

The F-measure is the harmonic mean of precision and recall.

$$F\text{-measure} = \frac{2 \textit{precision} \textit{recall}}{\textit{precision} + \textit{recall}} \quad (4.3)$$

Cohen's kappa

Cohen's kappa coefficient, κ , is a statistical measure for inter-rater agreement defined by Cohen in 1960 (cf. [Cohen, 1960]). Given two raters R_1 and R_2 who classify objects into two categories A and B , the probabilities p for their ratings are:

		R_1		
		A	B	Total
R_2	A	p_{11}	p_{12}	p_{A2}
	B	p_{21}	p_{22}	p_{B2}
	Total	p_{A1}	p_{B1}	1

The observed level of agreement p_0 is calculated as:

$$p_0 = p_{11} + p_{22}$$

The probability of random agreement p_e is calculated as:

$$p_e = p_{A1} p_{A2} + p_{B1} p_{B2}$$

Cohen's kappa coefficient κ is then calculated as:

$$\kappa = \frac{p_0 - p_e}{1 - p_e} \quad (4.4)$$

Measures for Error Correction

Both of the corpora we used for our experiments provide correction proposals that we can use as a gold standard. We can compare some statistical measures to calculate the similarity of the experiment's results to this gold standard. Results that are quite similar to the gold standard are expected to have a high probability to be good and valid correction proposals. Further, it is interesting to estimate how much a text has been edited. We expect corrections that are quite different to the original to be more creative corrections that possibly enhance the style of the text. Therefore, we want to measure the difference of corrections to their original versions.

To measure the difference between original text and corrected text we used the *Levenshtein distance* to calculate the edit distance. To measure the similarity to our gold standard, we apply the *Bilingual Evaluation Understudy* (BLEU) algorithm, which was developed by [Papineni et al., 2002] for evaluating machine translation. It calculates the similarity between a translation proposal and a machine translation result. In doing so, it takes into account that a possible re-structuring of a sentence or phrase does not alter its meaning or quality. Since our correction proposals can be seen as a translation proposal of erroneous text to its correct equivalent, BLEU is an appropriate measure to evaluate these corrections.

Levenshtein distance

The Levenshtein distance is an algorithm developed by Levenshtein in 1966 (cf. [Levenshtein, 1966]). It measures the similarity of two String representations of texts at character-level. It counts how many operations have to be done at a minimum in order to change one string into another. An operation is a change that has to be introduced to one character. Formally, the Levenshtein distance is calculated as follows:

Let m be the length of the first string u , and n the length of the second string v . The cells $D_{i,j}$ of matrix D of the size $m \cdot n$ are then defined as:

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} + 0 & \text{if } u_i = v_j \\ D_{i-1,j-1} + 1 & \text{if character is substituted} \\ D_{i,j-1} + 1 & \text{if character is inserted} \\ D_{i-1,j} + 1 & \text{if character is deleted} \end{cases} \quad (4.5)$$

with $1 \leq i \leq m$, and $1 \leq j \leq n$,

and the special cases $D_{0,0} = 0$, $D_{i,0} = i$, and $D_{0,j} = j$.

The complexity of the Levenshtein distance algorithm is $O(nm)$.

BLEU

The Bilingual Evaluation Understudy (BLEU) is an algorithm that calculates the similarity of two texts that takes into account that texts may be equivalent despite re-structuring (cf. [Papineni et al., 2002]). BLEU calculates a modified precision score p_n for each n -gram length. Applied to the sentence-wise evaluation of correction proposals, the BLEU formula reads as follows:

Let S_c be a sentence that has been corrected by workers in our experiment, and S_g the corresponding correct sentence from our gold standard. The BLEU score is then calculated as

$$Score = \exp \left\{ \sum_{n=1}^4 0.25 \log(p_n) - \max \left(\frac{L_g}{L_c} - 1, 0 \right) \right\} \quad (4.6)$$

with

L_g = the number of words in S_g

L_c = the number of words in S_c

$$p_n = \frac{n(S_c, S_g)}{n(S_c)}$$

where

$n(S_c)$ = the number of n -grams in S_c

$n(S_c, S_g)$ = the number of n -grams in S_c with co-occurrence in S_g

There are even more measures that could be valuable for the evaluation of error detection and error correction. There are further basic statistical measures that can be applied, like *true negative rate* and *accuracy* for the evaluation of error detection, and *word error rate* for the evaluation of error correction. There are

a vast number of algorithms that calculate the difference or similarity of texts, too. A few examples are the *cosine similarity*, the *Hamming distance* and the *Jaccard similarity coefficient*. Even the aforementioned algorithms have improvements, such as the *Damerau–Levenshtein distance* for the Levenshtein distance, and the *NIST* algorithm (cf. [Doddington, 2002]) for BLEU. In this thesis, we confine ourselves to the measures introduced in this section, which we found to be the most basic. The evaluation with further measures, with which the evaluation of evaluation measures would gain in importance, is left for future work.

4.4 Evaluation Results

In this chapter, we evaluate the experiments described in section 4.2 using the performance measures described in section 4.3. We present the evaluation results and discuss their meaning for assessing the quality of proofreading. We further provide some statistics about costs and working time spent on these experiments.

After we conducted our experiments and retrieved their results, we applied several statistical measures to them, as described in section 4.3. With these measures we want to quantify error detection and error correction. To measure error detection performance we calculated precision, recall, F-measure and Cohen’s kappa by comparing the positions of errors detected by workers to the error positions of our gold standard. To measure error correction performance we compared the corrected texts to both their uncorrected versions and the correct versions in our gold standard. To do the former, we calculated the edit distance using the Levenshtein distance. To do the latter, we calculated the similarity of corrections to our gold standard using BLEU.

Measuring Error Detection Performance

For each experiment we calculated precision, recall and F-measure. These measures give an idea of how error detection agrees with the errors in our gold standard. They describe how precise an error detection matches the gold standard (precision), how many of the gold standard errors were covered (recall), and how big this agreement is regarding both of these measures (F-measure).

We calculated precision, recall and F-measure at both word-wise and sentence-wise granularity level. In our evaluation we focus on the word-wise level, since it is more our concern which erroneous words or passages have been found, rather than which erroneous sentences. Nevertheless, we present the

Granularity level	Experiment						Measure
	#1	#2	#3	#4	#5	#6	
Word-wise	0.26	0.28	0.21	0.18	0.20	0.20	Precision
Sentence-wise	0.97	0.85	0.86	0.81	0.85	0.85	
Word-wise	0.90	0.76	0.63	0.83	0.85	0.91	Recall
Sentence-wise	1.0	0.96	0.86	0.94	0.99	0.98	
Word-wise	0.40	0.41	0.32	0.30	0.33	0.33	F-measure
Sentence-wise	0.98	0.90	0.86	0.87	0.92	0.91	

Table 4.2: Precision, recall and F-measure of error detection.

results in both word-wise and sentence-wise granularity level. The results of our evaluation with recall, precision, and F-measure are found in table 4.2.

Experiment #1 shows highest overall values. Since precision, recall and F-measure are all measures that do not regard true negative rates, the interpretation of experiment #1 having best proofreading results is not quite valid. Cohen’s kappa is a measure that does factor true negatives in, and as table 4.3 shows, the values of experiment #1 are crucially smaller. Nevertheless, these high precision and recall values indicate that workers probably do a better job if they focus on a rather small text segment.

Regarding those experiments that use input texts of paragraph size (#2–#6), experiment #5 shows the highest values in precision and F-measure. It shows that the editing a paragraph UI leads to more precise results than the annotating a paragraph UI. This is probably the case because in this UI workers edited less, which may be an indicator for the phenomenon that even though an editor presents a higher degree of freedom to the user, he or she does not rewrite large text passages, but rather picks the most obvious words that have to be corrected. Regarding these results, we can deduce that a higher degree of freedom for text editing does not encourage but rather discourages the user to rewrite large text passages. Yet, this hypothesis needs to be tested in further experiments that focus on this phenomenon, which remains as future work.

The comparison of experiments #3 and #4 shows a similar effect in which reduced work leads to a higher precision. These two experiments differ in their worker qualification requirement: #3 does not require any qualification, while #4 restricts the access to its HITs to workers that have an approval rate of at least 95%. Our evaluation results show that #3 has a higher precision than #4, but a lower recall. This is probably because the missing qualification requirement of #3 leads to a group of workers who might be less serious about

Granularity level	Experiment						Measure κ
	#1	#2	#3	#4	#5	#6	
Word-wise	0.1659	0.2218	0.1544	0.1012	0.0808	0.0752	
Sentence-wise	0.0	0.1473	0.5302	0.3571	0.2051	0.1847	

Table 4.3: Cohen’s kappa of error detection.

their job than it is the case in #4. If workers wanted to reduce the amount of work, they would just annotate the most obvious errors. This would lead to an error detection that is probably more precise compared to our gold standard. In contrary, workers in #4 obviously annotated larger text passages, seemingly more concerned about a common wording and good style. In doing so, they covered more of the errors in our gold standard, which leads to a higher recall.

The highest recall in experiments #2–#6 occurs in experiment #6. In this experiment we assigned each HIT to 10 workers instead of just 5. The results show that the more people annotate a text, the more errors they find. This is exactly what crowdsourcing is about: a larger crowd leads to a higher coverage in error detection, which is certainly more important for proofreading than the precision of the detection of few predefined errors.

Regarding the experiments with 5 assignments per HIT that use the annotating a paragraph UI (#3–#5), experiment #5 has the highest overall values, especially in recall. Since in #5 the qualification requirement was U.S. residency, this means that proofreading results apparently benefit from an access restriction to a group mostly consisting of native English speakers.

Another interesting fact is that our human-based computation approach seems to significantly outperform the automatic proofreading approach of [Brockett et al., 2006]. Brockett et. al. collected the ESL123 corpus to evaluate their approach that utilizes phrasal statistical machine translation to detect and correct writing errors. As stated in their paper, their algorithm reached a sentence-wise recall value of 0.62, while ours reached a sentence-wise recall of 1.0. Even though this number is not significant individually, it nonetheless proves that humans find more existing errors than their algorithm, or change the text so that errors are eliminated.

Unlike precision, recall, and F-measure, Cohen’s kappa is a measure that factors in the rates of true positives, false positives, false negatives, and true positives as well. Regarding each value, the inter-rater agreement coefficient, κ , is calculated. The results are presented in table 4.3.

Here, experiments #2 and #3 show the highest values. This is due to a higher agreement to the gold standard, and confirm the observations stated

Experiment		Measure
#1	#2	
24.99	69.15	Mean distance
0	1	Minimum
184	542	Maximum
5.48	10.75	Standard deviation

Table 4.4: Levenshtein distance between original text and corrected text.

above. Similarly to the precision values in table 4.2, this is an indicator for a smaller amount of work, or frankly speaking, for a certain laziness of workers.

Measuring Error Correction Performance

The experiments that use the annotating a paragraph UI (#3–#6) may provide precise error position data, but rather less useful values for measuring error correction. Even though these experiments provide a correction value for each annotated error, it is often the case that these correction proposals do not fit into the annotated text passage. Workers often propose corrections for bigger text passages than they marked as errors, and in several cases workers provide some information other than a proper correction proposal. Eventually, the benefit of this UI is that authors who use this UI to let workers proofread their text can choose between multiple correction proposals and decide for themselves which one is the most appropriate. Anyway, we decided to just measure the text correction of those experiments that used the editing UIs: experiments #1 and #2. Here, we can compare the corrected text to (1) the erroneous original text, and (2) to the respective correct text of our gold standard.

To measure the edit distance between original text and the worker’s correction, we calculated the Levenshtein distance. The results comprising the mean distance, minimum and maximum values, and standard deviation are shown in table 4.4.

These results show that more editing operations have been made in experiment #2 than in #1. This is not surprising, since the texts in #1 are sentences while the texts in #2 are paragraphs. In #2 there is simply more text that can be changed, which naturally leads to a greater number of changes. So these results are not quite conclusive. In a further evaluation of experiment #1 we therefore tried to find out if a higher Levenshtein distance indicates a higher quality of corrections. For this purpose we manually rated the correction qual-

Experiment		Measure
#1	#2	
0.48	0.67	Mean BLEU
0.0	0.23	Minimum
1.0	0.93	Maximum
0.44	0.16	Standard deviation

Table 4.5: BLEU of corrected sentences compared to our gold standard.

ity of each assignment with values from 0 to 2, with 0 being the poorest quality and 2 being the best quality. We then calculated the correlation between Levenshtein distance and quality assessment and came to a correlation coefficient of 0.06. This low value rather indicates that there is no convincing correlation between the amount of editing and the correction’s quality.

To measure the similarity of corrections and the correct texts given in our gold standard we applied the BLEU algorithm. BLEU calculates the similarity of two text passages (in our case sentences) while regarding possible co-occurrences. Table 4.5 shows the mean BLEU of experiments #1 and #2, their minimum and maximum values, and their standard deviation.

In comparison, experiment #2 has a higher mean BLEU than experiment #1. This means that corrections of #2 are more similar to our gold standard, regardless if a sentence is re-structured in some way. The different standard deviations here show that the sentences in #1 have been edited in a way that they are more different to the gold standard than the sentences in #2. This corresponds to the aforementioned assumption that short texts encourage workers to rewrite them, whereas larger texts displayed in a text editor discourage workers to rewrite longer text passages. Another possible explanation is that the error rate of the ESL123 corpus used in #1 is considerably higher than the error rate of MELD used in #2 (ESL123: 19.75%, MELD: 7.55%, see 4.1). So, workers were more encouraged to re-write larger text segments in #1 than in #2, merely due to the higher amount of given errors.

To see all these measures as indicators of quality is highly questionable. They are rather measures for the agreement between error detection or error correction results and the gold standard. A higher agreement may be an indicator for higher proofreading quality, but a low agreement is not compulsorily a sign for poor proofreading quality. A proper assessment of the proofreading quality can only be made by manually reviewing and rating all results. Such a quality assessment could also help evaluating the different performance measures. But since this is a crucially labor-intensive and time-consuming task,

Experiment						Measure
#1	#2	#3	#4	#5	#6	
3.68	3.50	11.00	12.50	4.70	9.85	Total cost [\$]
1 813	2 223	6 659	6 659	2 223	2 223	Number of words in original texts

Table 4.6: Costs of our experiments compared to number of words and rejection ratio.

Experiment						Measure
#1	#2	#3	#4	#5	#6	
13.7	8.5	28.1	28.5	9.7	16.8	Total working time [h]
398	100	269	275	94	197	Number of assignments
2	5	6	6	6	5	[min] per assignment

Table 4.7: Working time and hourly wage of experiments.

such further evaluation is beyond the scope of this Bachelor thesis and is left for future work.

Costs and Working Time

Besides the quality of error detection and error correction, there are some other values that should be analyzed to evaluate if human-based computation is a good way for doing proofreading. An interesting factor for comparison to the traditional way of consulting a professional proofreader is how much it costs. For our experiments that comprised 21 800 words in total we paid a sum of \$45.23. Table 4.6 shows the single values of each experiment, comparing the costs to the number of words.

In experiments #1–#5 we paid about \$2 per 1 000 words on average, and about \$4 per 1 000 words in experiment #6. According to the Society for Editors and Proofreaders (SfEP) a minimum hourly wage of about \$30 is standard.² Compared to this, proofreading via crowdsourcing is quite inexpensive. In table 4.7 we further present the working time that has been spent by workers on each experiment.

The values of total working time can be seen as the amount of working

²SfEP’s suggestions of minimum freelance wages can be accessed at http://www.sfep.org.uk/pub/mship/minimum_rates.asp

time that has been saved. Each of our experiments was conducted in less than 24 hours. That the value of total working time can even exceed this time is due to the simultaneous work of many people. Imagine that the author or the person charged with proofreading would have to spend this amount of time, the enormity of time saving becomes clear. Regardless of the amount of time spent in the consequent reviewing process (which the writer has to spend either way), the amount of working hours undertaken by workers is the most valuable benefit of crowdsourcing.

Chapter 5

Conclusion

In this thesis, we examined the qualities of human-based computation as an approach for semi-automatic proofreading. We developed three different user interfaces for proofreading tasks on MTurk, and one for reviewing the proofreading results. We conducted six experiments with these different proofreading UIs and input texts with different properties. By evaluating these experiments we found that workers do less re-writing with the *editing a paragraph* UI than with the *editing a sentence* UI, and cover more errors with the *annotating a paragraph* UI as more workers are assigned to a HIT. We further found that the access restrictions for HITs to US residents as qualification requirement helps to increase the quality of error detection. Further evaluation of the quality of proofreading results and of proofreading performance measures are left for future work, as well as further development of proofreading UIs and reviewing UIs. The following statement does not only summarize the value of crowdsourcing for proofreading, but also illustrates how proofreading via MTurk works: for this purpose we had this thesis' last paragraph corrected with our *editing a paragraph* UI, and visualized deletions in red and crossed out, and insertions in green.

When talking about crowdsourcing as a way ~~for~~ of proofreading, sooner or later the question ~~rises~~ ~~arises~~ ~~if~~ ~~as~~ ~~to~~ ~~whether~~ the anonymous crowd can be trusted. To many people, it seems unlikely that a group of strangers barely qualified for proofreading would provide high quality error detection and correction. In this thesis, we evaluated proofreading via crowdsourcing, and results show that crowdsourcing is indeed a good and cheap resource for this kind of work. ~~But~~ ~~even~~ ~~regardless~~ ~~Regardless~~ of the quality of proofreading results, crowdsourcing brings ~~some~~ further ~~values~~ ~~value~~ for the writer than ~~the~~ traditional ~~ways~~ ~~methods~~. The crowd is able to perform a high amount of working hours in a relatively short time ~~period~~ for an ~~amazing~~ ~~amazingly~~ low price. In the end, the diversity of the crowd leads to a selection of multiple

annotations and corrections, which ~~this~~ is the actual value of crowdsourcing. ~~There~~ Currently, there is no ~~other~~ tool ~~and no~~ or method of accumulating so many different proposals for corrections to one text that is ~~that as easy simple~~, fast, cheap, and easy to use. We hope that, inspired by our work on that issue, other and better tools will be developed ~~and~~ to make proofreading easy and affordable for everyone.

Bibliography

- [Bernstein et al., 2010] Bernstein, M. S., Little, G., Miller, R. C., Hartmann, B., Ackerman, M. S., Karger, D. R., Crowell, D., and Panovich, K. (2010). Soylent: A word processor with a crowd inside. In *Proceedings of the 23rd annual ACM symposium on User interface software and technology*, pages 313–322. ACM.
- [Brabham, 2008] Brabham, D. C. (2008). Crowdsourcing as a model for problem solving. *Convergence: The International Journal of Research into New Media Technologies*, 14(1):75.
- [Bredenkamp et al., 2000] Bredenkamp, A., Crysman, B., and Petrea, M. (2000). Looking for errors: A declarative formalism for resource-adaptive language checking. In *Proceedings of the 2nd International Conference on Language Resources and Evaluation*.
- [Brockett et al., 2006] Brockett, C., Dolan, W. B., and Gamon, M. (2006). Correcting esl errors using phrasal smt techniques. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the ACL*, pages 249—256, Sydney, Australia. Association for Computational Linguistics.
- [Butcher et al., 2006] Butcher, J., Drake, C., and Leach, M. (2006). *Butcher’s copy-editing: the Cambridge handbook for editors, copy-editors and proof-readers*. Cambridge University Press, 4 edition.
- [Carroll, 1998] Carroll, J. M. (1998). *Minimalism Beyond the Nurnberg Funnel*. Massachusetts Institute of Technology.
- [Cherry and Vesterman, 1981] Cherry, L. and Vesterman, W. (1981). *Writing Tools: The STYLE and DICTION Programs*. Bell Telephone Laboratories.
- [Cohen, 1960] Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and psychological measurement*.

- [Díaz-Negrillo and Fernández-Domínguez, 2006] Díaz-Negrillo, A. and Fernández-Domínguez, J. (2006). Error tagging systems for learner corpora. *Revista española de lingüística aplicada*, (19):83.
- [Doddingtion, 2002] Doddingtion, G. (2002). Automatic evaluation of machine translation quality using n-gram co-occurrence statistics. In *Proceedings of the second international conference on Human Language Technology Research*, pages 138–145. Morgan Kaufmann Publishers Inc.
- [Domeij et al., 2000] Domeij, R., Knutsson, O., Carlberger, J., and Kann, V. (2000). Granska - an efficient hybrid system for swedish grammar checking. In *Proceedings of the Twelfth Nordic Conference in Computational Linguistics (NoDaLiDa)*, pages 49–56.
- [Erickson, 1990] Erickson, T. D. (1990). Working with interface metaphors. *The Art of Human-computer Interface Design*, pages 65–73.
- [Fitzpatrick and Seegmiller, 2001] Fitzpatrick, E. and Seegmiller, M. S. (2001). The montclair electronic language learner database. In *Proceedings of the International Conference on Computing and Information Technologies*.
- [Fromkin et al., 2003] Fromkin, V., Rodman, R., and Hyams, N. (2003). *An Introduction to Language*. Wadsworth, 7 edition.
- [Garside, 1996] Garside, R. (1996). The robust tagging of unrestricted text: The bnc experience. *Thomas and Short*, pages 167–180.
- [Gentry et al., 2005] Gentry, C., Ramzan, Z., and Stubblebine, S. (2005). Secure distributed human computation. In *Proceedings of the 6th ACM Conference on Electronic Commerce*, pages 155–164. ACM.
- [Granger, 2003] Granger, S. (2003). Error-tagged Learner Corpora and CALL: A Promising Synergy. *CALICO Journal*, 20(3):465–480.
- [Hackos, 1999] Hackos, J. T. (1999). An application of the principles of minimalism to the design of human-computer interfaces. *Common Ground*, 9:17–22.
- [Hammond and Fogarty, 2005] Hammond, M. O. and Fogarty, T. C. (2005). Co-operative oulip-ian generative literature using human based evolutionary computing. *Proceedings of GECCO-2005*.

- [Helfrich and Music, 2000] Helfrich, A. and Music, B. (2000). Design and evaluation of grammar checkers in multiple languages. In *Proceedings of the 18th conference on Computational Linguistics*, volume 2, pages 1036–1040. Association for Computational Linguistics.
- [Horton and Chilton, 2010] Horton, J. J. and Chilton, L. B. (2010). The labor economics of paid crowdsourcing. In *Proceedings of the 11th ACM Conference on Electronic Commerce*, pages 209–218. ACM.
- [Horwood, 1997] Horwood, T. (1997). *Freelance Proofreading and Copy-editing - a guide*. Action Press, Virginstow, 2 edition.
- [Howe, 2006a] Howe, J. (2006a). Crowdsourcing: A definition. *Crowdsourcing: Tracking the Rise of the Amateur*. (weblog, 2 June). URL (accessed February 26, 2011): http://crowdsourcing.typepad.com/cs/2006/06/crowdsourcing_a.html.
- [Howe, 2006b] Howe, J. (2006b). The rise of crowdsourcing. *Wired*, 14(6).
- [Islam and Inkpen, 2009] Islam, A. and Inkpen, D. (2009). Real-word spelling correction using google web 1t n-gram data set. In *Proceeding of the 18th ACM Conference on Information and Knowledge Management*, pages 1689–1692. ACM.
- [Kittur and Kraut, 2008] Kittur, A. and Kraut, R. E. (2008). Harnessing the wisdom of crowds in wikipedia: Quality through coordination. In *Proceedings of the ACM 2008 Conference on Computer Supported Cooperative Work*, pages 37–46. ACM.
- [Kosara and Ziemkiewicz, 2010] Kosara, R. and Ziemkiewicz, C. (2010). Do mechanical turks dream of square pie charts. In *BELIV Workshop*, pages 373–382.
- [Lakoff and Johnson, 1980] Lakoff, G. and Johnson, M. (1980). *Metaphors We Live By*. The University of Chicago Press.
- [Leacock et al., 2010] Leacock, C., Chodorow, M., Gamon, M., and Tetreault, J. (2010). *Automated Grammatical Error Detection for Language Learners*. Morgan & Claypool, 1 edition.
- [Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710.

- [Lightbound, 2005] Lightbound, P. M. (2005). *An Analysis of Interlanguage Errors in Synchronous/Asynchronous Intercultural Communication Exchanges*. PhD thesis, Universitat de València.
- [Little et al., 2009] Little, G., Chilton, L. B., Miller, R., and Goldman, M. (2009). Turkit: Tools for iterative tasks on mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, pages 29–30. ACM.
- [Mudge, 2010] Mudge, R. (2010). The design of a proofreading software service. In *Proceedings of the NAACL HLT 2010 Workshop on Computational Linguistics and Writing*, pages 24—32. Association for Computational Linguistics.
- [Myers, 1986] Myers, E. W. (1986). An o (nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266.
- [Naber, 2003] Naber, D. (2003). A rule-based style and grammar checker. Master’s thesis, Universität Bielefeld, Technische Fakultät.
- [Paolacci et al., 2010] Paolacci, G., Chandler, J., and Ipeirotis, P. G. (2010). Running experiments on amazon mechanical turk. *Judgment and Decision Making*, 5(5):411–419.
- [Papineni et al., 2002] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.
- [Pravec, 2002] Pravec, N. A. (2002). Survey of learner corpora. *ICAME Journal*, 26:81–114.
- [Richardson and Braden-Harder, 1988] Richardson, S. and Braden-Harder, L. (1988). The experience of developing a large-scale natural language text processing system: Critique. In *Proceedings of the Second Conference on Applied Natural Language Processing*, pages 195–202. Association for Computational Linguistics.
- [Rooy and Schafer, 2002] Rooy, B. V. and Schafer, L. (2002). The effect of learner errors on pos tag errors during automatic pos tagging. *Southern African Linguistics and Applied Language Studies*, 20(4):325–335.
- [Surowiecki, 2005] Surowiecki, J. (2005). *The Wisdom of Crowds*. Anchor Books, 1 edition.

- [Takagi, 2001] Takagi, H. (2001). Interactive evolutionary computation: Fusion of the capabilities of ec optimization and human evaluation. In *Proceedings of the IEEE*, volume 89, pages 1275–1296. IEEE.
- [Tetreault et al., 2010] Tetreault, J. R., Filatova, E., and Chodorow, M. (2010). Rethinking grammatical error annotation and evaluation with the amazon mechanical turk. In *Fifth Workshop on Innovative Use of NLP for Building Educational Applications*, page 45. Association for Computational Linguistics.
- [Tono, 2003] Tono, Y. (2003). Learner corpora: Design, development and applications. In *Proceedings of the Corpus Linguistics 2003 Conference*, pages 800–809.
- [von Ahn, 2005] von Ahn, L. (2005). *Human computation*. PhD thesis, Carnegie Mellon University.
- [von Ahn and Dabbish, 2004] von Ahn, L. and Dabbish, L. (2004). Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 319–326. ACM.
- [von Ahn et al., 2008] von Ahn, L., Maurer, B., McMillen, C., Abraham, D., and Blum, M. (2008). recaptcha: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468.
- [Wilson, 1901] Wilson, J. (1901). *The Importance of the Proof-reader. A Paper read before the Club of Odd Volumes, in Boston, by John Wilson*. Cambridge University Press.