University of Zurich[UZH]

Master's thesis
presented to the Faculty of Arts and Social Sciences
of the University of Zurich

for the degree of
Master of Arts UZH

Treatment of Markup in Statistical Machine Translation

Author: Mathias Müller
Student ID Nr.: 09-746-652

Examiner: Prof. Dr. Martin Volk

Institute of Computational Linguistics

Submission date: January 10, 2017

Abstract

This thesis presents work on markup handling in phrase-based, statistical machine translation. In translation workflows that involve post-editing of machine translation, markup handling is an essential mechanism. It is essential because the documents that are translated frequently contain markup, mostly in the form of inline formatting. Several solutions to the problem of markup handling have been put forward, but there is no consensus as to which strategy should be employed in standard use cases. As a consequence, standard machine translation tools do not have any markup handling method at all, except for *ignoring* markup which can hardly be called a reasonable strategy.

What is needed for a meaningful discourse about markup handling – that could lead to implementations in standard tools – are *comparisons* between different markup handling strategies. In order to facilitate those comparisons, I have implemented five different markup handling strategies in the same machine translation framework. Using those implementations, I have conducted three experiments that compare the usefulness of markup handling strategies. The results of both an automatic and manual evaluation show that identity masking, a method that replaces markup with unique mask tokens for training and translation, is the most promising method. However, alignment masking and alignment reinsertion are not too far behind and should be regarded as viable alternatives.

In summary, the main contributions of this thesis are: a comprehensive survey of existing markup handling solutions, reimplementations of existing and novel solutions in a convenient framework that is available for free and recommendations regarding the choice of markup handling strategy.

Zusammenfassung

Diese Arbeit behandelt Methoden zur Verarbeitung von Markup in phrasenbasierter, statistischer maschineller Übersetzung. In Übersetzungsworkflows die das Post-Editing von maschinellen Übersetzungen beinhalten ist die richtige Verarbeitung von Markup essentiell. Dies, weil die zu übersetzenden Dokumente oft Markup in Form von Inline-Formatierung enthalten. Es existieren einige Ansätze zur Verarbeitung von Markup, aber es herrscht keine Einigkeit darüber, welche Methode in Standardfällen eingesetzt werden sollte. Deswegen ist in Standard-Tools für maschinelle Übersetzung gar keine solche Methode verfügbar, ausser Markup komplett zu ignorieren, was aber nur schwerlich als richtige Verarbeitung von Markup angesehen werden kann.

Was wirklich nötig ist für die Auseinandersetzung mit der Verarbeitung von Markup (die zu Implementationen in Standard-Tools führen könnte), sind *Vergleiche* zwischen verschiedenen Strategien. Um diese Vergleiche zu ermöglichen, habe ich fünf verschiedene Strategien zur Verarbeitung von Markup in demselben Übersetzungsframework implementiert. Mithilfe dieser Implementationen habe ich drei Experimente durchgeführt, die die Performanz der Strategien untereinander vergleichen. Die Resultate der automatischen und manuellen Evaluation zeigen, dass Identity-Masking (eine Methode, die Markup im Training und während der Übersetzung mit Masken ersetzt) die vielversprechendste Strategie ist. Alignment-Masking und Alignment-Reinsertion sind aber fast gleichauf und sollten als wirkliche Alternativen angesehen werden.

Zusammenfassend ist der Beitrag dieser Arbeit folgendes: ein vollständiger Überblick über existierende Methoden zur Verarbeitung von Markup, Reimplementationen von existierenden und neuen Methoden in einem benutzerfreundlichen und gratis erhältlichen Framework und Empfehlungen für die Wahl einer geeigneten Strategie.

Acknowledgements

First and foremost, I would like to thank my supervisor, Martin Volk, for relentless encouragment and his keen eye for misguided thought, not only during the writing of this thesis, but throughout the several years we have known each other. To a large extent, it is due to him that I found my way into the field of machine translation, and for that I am very grateful. Another person I have greatly benefited from is Samuel Läubli who has led me by example in many ways, be they as mundane as writing quality code or as worthy as being an outstanding academic.

That I got really interested in all things XML and markup in the first place is due to another person, Roberto Nespeca, to whom I am also extremely grateful. He, along with Sandra Roth, has shown me the business side of writing software and has kindly provided the data I have used in my experiments. I am further indebted greatly to a number of people who have helped me understand their code or have offered their insights: Achim Ruopp from TAUS, Christian Buck from UEDIN, Eric Joanis from the NRC and Saša Hasan from Lilt.

Thanks to the Moses community for an incredibly useful collection of tools and the insightful discussions on the Moses support mailing list. Thanks to the XML community on Stackoverflow that was my home for a couple of years and that I miss dearly.

Thanks to the people who had the questionable honor of proofreading my work: Rahel Gerber, Benjamin Pelzmann and Phillip Ströbel. If there are errors in my writing I have surely introduced them after their skillful correction. Thanks to Thomas Wismer who has offered very helpful advice on typography and layout.

And last, but certainly not least, my heartfelt thanks to my parents, who have always known the enormous value of education.

The director (with blue contact lenses) says a few long sentences in Japanese.

The translator, a middle-aged woman in a coordinated outfit, translates but it is only a short sentence now.

Bob wonders what she's leaving out, or if that's the way it works from Japanese to English.

—- Sofia Coppola, *Lost in Translation*

Contents

List of Abbreviations

BLEU	Bilingual Evaluation Understudy, a well-known metric in machine translation
CAT	Computer-assisted Translation *or* Computer-aided Translation
DOM	Document Object Model
DTD	Document Type Definition, a schema language for XML
GALA	Globalization and Localization Association
HTML	HyperText Markup Language
LISA	Localization Industry Standards Association
LSP	Language Service Provider
MERT	Minimum Error Rate Training
MT	Machine Translation
NIST	National Institute of Standards and Technology
NLP	Natural Language Processing
OASIS	Organization for the Advancement of Structured Information Standards
OOXML	Office Open XML, a suite of standards used in Microsoft Office products
PDF	Portable Document Format, a popular binary file format
RSS	Rich Site Summary, a standard for web feeds
SAX	Simple API for XML, event-driven XML parsing
SMT	Statistical Machine Translation
SVG	Scalable Vector Graphics, a web standard for vector graphics
TEI	Text Encoding Initiative, a digital humanities standard and community
TER	Translation Edit Rate
TM	Translation Memory *or* Translation Model
TMX	Translation Memory eXchange, a structured format for translation memories
W3C	Word Wide Web Consortium, a standardization body
XDM	XQuery 1.0 and XPath 2.0 Data Model
XLIFF	XML Localization Interchange File Format
XML	eXtensible Markup Language, a popular markup standard
XSLT	eXtensible Stylesheet Language Transformations, a programming language

1 Introduction

If there is ever a thing that translators despise, it is when something of great importance is "lost in translation". In documents that are translated as I write, *markup* is one of those important ingredients – and yet, paradoxically, it is routinely lost when *machines* do the translation. I, for one, am intrigued by this fact and I have set the goals for my thesis work accordingly: firstly, to explain how exactly markup is lost in machine translation and secondly, to enable research into how markup handling in machine translation should be done. But before, I am well aware that I have to explain why the loss of markup is a problem in the first place.

Explaining why the loss of markup is a problem is straightforward enough to summarize it here, without giving away too much of the discussions later on. To put it bluntly, markup is lost in machine translation because machines translate *text*, while humans translate *documents*. Documents are different from raw text in that they have structure and information encoded in them that go beyond the text itself. To give a specific example, additional information found in a document might be *formatting*. Formatting as a concept is quite universal, virtually every type of document offers formatting in one way or another. But not only is formatting (and everything else that lives in documents apart from the text) ubiquitous, most of it is also encoded with the same basic mechanism, *markup*.

So far, I have established that documents are the unit of translation for human translators and that those documents likely contain markup. Human translators and marked-up documents get on well with each other and there would be no catch so far, if it were not for machine translation. The catch is that most translation systems are ill-equipped to deal with markup in a proper way, especially if they are out-of-the-box and trained with standard tools. This is unfortunate because industrial translation workflows typically involve machine translation systems, even more so after several studies have proven that they increase translator productivity.

In fact, in many translation projects, pre-translation of the source segments is an obligatory step and human translators, instead of translating from scratch, will *post-edit* those pre-translations. Given that machine translation is so commonplace, one could be forgiven to expect that machine translation tools for translators woud be truly assistive, in the sense that they would match the needs of translators very closely, but this does not always hold true. With regard to markup it certainly does not, as machine translation is lacking a standard solution that is grounded in empirical research and, short of a highly customized solution for markup handling, will either ignore markup or treat it improperly.

Spinning all loose ends into a single strand again, markup being lost in machine translation is problematic (and, by extension, a problem worth solving) because machine translation is an integral part of human translation, because human translation involves the translation of markup and because machine translation has no standard solution for markup translation.

After this rather elaborate motivation for the problem at hand, I will briefly comment on my first goal, namely on explaining *how* markup is lost in machine translation. But first, it is imperative that I make two important clarifications. Firstly, what I call "losing markup" does not only describe situations where markup from the source language is completely *absent* from the translation. Rather, loss of markup entails any modification to it during translation that renders it unusable. And since XML, the only markup language this thesis is concerned with, imposes strict rules on XML documents, there are many ways to destroy XML markup. Secondly, until now I have made it seem like machine translation in its entirety loses markup, but such a claim would, of course, be a bold hyperbole and downright wrong.

A more realistic assessment is that, with standard tools, markup indeed falls through the cracks and is not seen as content that should be translated. For instance, this is the case for standard tokenization.

There is no generally accepted solution to markup handling as there is for other important problems such as word alignment. General acceptance of one method or another would require comparisons between all methods that have been put forward, but such comparative studies are missing, and on top of that, the actual published solutions are few and far between. Therefore, markup handling in machine translation can rightly be described as an underresearched topic.

Keeping in mind those circumstances, my second and most important goal follows very naturally: my thesis work should provide implementations of existing and new markup handling strategies in a way that it would facilitate empirical comparisons between the methods. Nothing in my implementations is outrageously novel, but it resulted from a careful survey of the literature and all existing solutions. All implemented methods are embedded into the same machine translation framework[1] so that they can be compared in controlled experiments. I have conducted a number of experiments myself and will report the results in the thesis.

The observations from my experiments have resulted in a number of recommendations. They are intended to guide interested readers with the choice of markup handling strategy for their own projects and enable them to make informed decisions. For this reason, I have high hopes that my work will lead to less markup being lost in translation.

1.1 Outline

Section 2 adds more substance to the motivation outlined in the introductory paragraphs, but also gives background information. It explains the essential concepts of statistical machine translation, markup languages and translation process research, all of which are necessary to understand later sections. Section 3 discusses all solutions to markup handling that have been developed until today, surveying the published literature and unpublished tools found elsewhere.

Moving on to more practical work, Section 4 describes all strategies for markup handling that I have implemented myself, all inspired by or based on earlier work entirely. For all methods, I have intended to give reproducible code examples that hopefully make clear that my software modules can be reused in other work without effort. Section 5 explains three experiments I have conducted, all of them are comparisons between different methods of markup handling. In addition, this section also introduces the data set I have used in all experiments. Section 6 reports the results of the experiments, and they are discussed thoroughly in Section 7.

Section 8 summarizes the main contributions, findings and limitations of this thesis and puts forward a number of recommendations for future projects. Finally, Section 9 suggests future avenues of research regarding the problem of markup handling in machine translation.

[1] The whole framework my thesis work is embedded into, `mtrain`, will eventually be open-sourced. Until then, it is available for free upon request.

2 Background and Motivation

This section explains the fundamental concepts of machine translation (in particular, *statistical* machine translation; SMT), markup languages (in particular, *XML*) and translation process research (in particular, the notion of *translator productivity*). It is intended mainly for readers unfamiliar with the topics. Readers who already have a firm grasp of all of them are invited to skip ahead as I would not want to delay them unnecessarily.

To this end, Section 2.1 explains, in a nutshell, the fundamental ideas that shape machine translation until today and, in more detail, the workings of phrase-based, statistical machine translation, since I am using this type of translation system in my experiments. From a completely different perspective, Section 2.2 explains what markup languages are and also demonstrates clearly how standard machine translation tools are not able to properly deal with XML markup. Lastly, Section 2.3 very briefly introduces concepts from translation process research and argues that improper markup handling has an impact on translator productivity.

2.1 Statistical Machine Translation

Here, I will explain what machine translation is, how it came into being and what its most important concepts are. It is mostly an introduction to phrase-based, statistical machine translation, not because this paradigm of machine translation is the most cutting-edge, but because it is well described in the literature, and well established in industry. Also, I am using phrase-based, statistical systems in my experiments and thus it makes sense to give background information about them. The real cutting edge of machine translation research is, at the time of writing, neural machine translation, which has gained a lot of momentum recently (much of it due to an influential paper by Devlin et al., 2014).

Machine translation as a concept and its history are described in Section 2.1.1, and it will become evident how phrase-based, statistical machine translation arose from those beginnings. Section 2.1.2 follows seamlessly with a not-so-gentle introduction to the concepts of phrase-based, statistical machine translation. Section 2.1.3 recalls the importance of peripheral processes outside of the core methods of machine translation, i.e. the pre- and postprocessing steps applied to raw parallel text and highlights the fact that markup handling always means changing those auxiliary processes, rather than changing the models themselves.

2.1.1 Advent of Machine Translation and Its Early History

The idea of automatic translation occurred to prescient thinkers as early as the seventeenth century (Hutchins, 2005), but all of those works were theoretical considerations. Actual research into machine translation (MT) did not begin in earnest until the 1950s. It is perhaps not a coincidence that MT would be born in those years, since they saw the first electronic computers being invented (Williams, 1997; Ceruzzi, 2003) and suddenly, there was a practical side to automatic translation.

At that time, it is especially the works of Warren Weaver and Claude Shannon[2] that have set the stage for the inception of machine translation. Warren Weaver has very clearly and for the first time articulated the possibility of using machines for translation and put forward believable proposals, while Claude Shannon has provided the theoretical framework upon which statistical machine translation rests until today.

[2] The informed reader must forgive the chronological disorder. I am aware that Shannon's work during the war is one of the reasons Weaver has composed his *memorandum*, but for the sake of the development of this chapter I am reporting on them backwards.

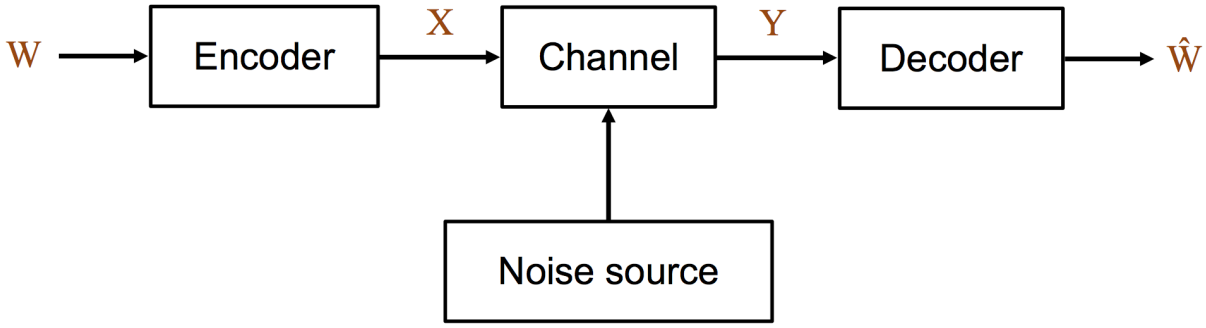

Figure 2.1: Components of any general system of communication, adapted from Shannon (1948, 7) and Manning and Schütze (1999, 69).

In 1949, Warren Weaver distilled his thoughts on machine translation into his famous *memorandum* (Weaver, 1955), the "direct stimulus for the beginnings of research [on machine translation] in the United States" (Hutchins, 2000a, 17). Those beginnings include the appointment of Yehoshua Bar-Hillel to conduct research on MT at MIT in 1951, the organization of the first MT conference and the first public demonstration of an MT system shortly after, in 1954 (Hutchins, 2005). The quintessence of the memorandum are proposals, each of them meant to delineate possible avenues of research:

- *Meaning and Context*: Weaver realized that one of the stumbling blocks of automatic translation is ambiguity. He proposed that the ambiguity of words be resolved by looking at their local context. Approaching ambiguity on the word level in this fashion is promising, but looking back at the text today, it is fair to say that Weaver greatly underestimated the problem.
- *Language and Invariants*: It is argued that, while languages are different at the surface, there might be universal aspects of language that are common to all means of linguistic communication. Translation could be seen as reducing a human language to a universal *interlingua*, and from this common denominator that underlies all languages, generating another surface language.
- *Translation and Cryptography*: This proposal suggests that translation can be likened to cryptography in the sense that the source language of a translation process is nothing else than an "encoded" version of the target language. It also foreshadows an important realization: "Perfect translation is almost surely unattainable" (Weaver, 1955, 22) and it is emphasized that methods should concentrate on reducing the margin of error instead.

The fourth proposal, *Language and Logic* does not seem significant to me, and Weaver devoted very little space to explaining it. In fact, he believed the search for language universals to be the most feasible approach. There was indeed a time when research on language universals and interlingua systems were the prevailing acitvity in the field, in the 1970s and 1980s (Hutchins, 1995b, 11ff), but current research in the last decades has very much focused on the statistical nature of language. This should not be understood as a sign of an immature science, as it is not uncommon for active fields of research to undergo such "shifts of paradigm" (Kuhn, 2012). But since statistical methods (together with neural networks) are currently the dominant paradigm, I will now put *recent* developments in MT research into perspective by explaining how information theory is the basis of recent statistical MT systems.

Claude Shannon came to be known as the "father of information theory" and is widely regarded as having heralded the information age. Among other things, he is credited with pointing out that information can be measured and quantified in "binary digits" (later shortened to "bits") and with introducing entropy as a measure of uncertainty. But most relevant to machine translation is his development of the *noisy-channel model* (Shannon, 1948, 2001). One of the central tenets of the model is that

application	input	output	p(i)	p(o | i)
machine translation	target language	source language	language model	translation model
speech recognition	recognized text	speech signal	probability of text	acoustic model
POS tagging	tag sequence	written text	tag probability	$p(word\|tag)$

Table 2.1: Typical problems in natural language processing, seen as decoding problems (adapted from Manning and Schütze, 1999, 71).

communication can be formalized as

- selecting a *message* from a finite alphabet,
- encoding this message,
- sending the message though a *channel*
- and decoding the message, i.e. attempting to reconstruct the original meaning.

Also, when information is transmitted it is inevitable that the encoded message is distorted to a certain degree, because of the introduction of noise (see Figure 2.1). While this formalization of communication is invaluable to information theory because it introduces crucial concepts like *channels* and their *capacity*, it has informed a more general *change of perspective* in other sciences. This change of perspective came about because many problems thought to be completely unrelated to information theory could be cast as *decoding problems*.

This is certainly true for the field of natural language processing (NLP) which has embraced the noisy-channel model, and many of its problems fit into the model nicely (see Table 2.1 for examples). In all those cases, the model can even be simplified because there is usually no way to influence the encoding step and solving the problem boils down to "discovering" the most likely input, given an output. Thus, we are looking for the following:

$$\hat{I} = argmax_i \; p(i|o) \tag{2.1}$$

Which can conveniently be rewritten using Bayes' theorem, a trivial transformation:

$$\hat{I} = argmax_i \; \frac{p(o|i) \; p(i)}{p(o)} = argmax_i \; p(o|i) \; p(i) \tag{2.2}$$

The rightmost part of Equation 2.2 indicates that the probability $p(o)$ can even be ignored because it is a constant value. In the general case, we are left with having to estimate the distributions of $p(o|i)$ and $p(i)$ which can be derived from training data.

In the specific case of machine translation, this means that French should actually be regarded as distorted English, and that the original English utterance was somehow lost in the process. This might seem an unlikely scenario and you might be tempted to accuse MT researchers of linguistic chauvinism, but it is only meant as a hypothetical situation – a fact which "does not in any way detract from the usefulness of the method" (Mani, 2001, 67).

The next section will focus on the mechanisms and foundations of phrase-based, *statistical* machine translation which for the last decade has been the prevailing paradigm. For a more complete treatise of machine translation as a whole and how the field has developed over time, the reader is referred to the many writings of Hutchins (1986, 1994, 1995a,b, 1997, 2000b, 2001, 2007).

2.1.2 Phrase-based, Statistical Machine Translation

The cornerstone of phrase-based, statistical machine translation (SMT) is the adapted version of the noisy-channel model given in Equation 2.2. Formally, the problem of statistical machine translation can be stated as

$$\hat{E} = argmax_e \; p(f|e) \; p(e) \tag{2.3}$$

Where $\hat{E}$ is the most likely translation, $p(f|e)$ is equivalent to the *translation model* and $p(e)$ is described by the *language model*. For historical reasons, it is a convention in SMT to use f and e to denote the source (**f**oreign) and target (**E**nglish) language, respectively (for a thorough introduction to statistical machine translation in general, see Koehn, 2010).

The translation model $p(f|e)$ is a distribution of probabilities of stretches of text in the source language f, given another stretch of text in the target language e.[3] What could be a useful definition for such stretches of text that should be the building blocks of the translation model? Collecting the probabilities of single words will lead to a model that is rather unexpressive, and entire sentences make the model too sparse (a lot of probability mass would have to be wasted on unseen strings to make it useful).

This is why Koehn et al. (2003) have introduced the *phrase* as the fundamental unit for the translation model. In SMT, there is no linguistic definition for phrases, they simply mean arbitrary sequences of adjacent words, including punctuation. If we break up f and e into phrases that correspond to each other, then $p(f|e)$ becomes

$$p(\bar{f}_1^I|\bar{e}_1^I) = \prod_{i=1}^{I} \phi(\bar{f}_i|\bar{e}_i) \; d(start_i - end_{i-1} - 1) \tag{2.4}$$

Namely, given a sequence of phrases $\bar{e}_1^I$, the probability of the phrases in the source language $\bar{f}_1^I$ is the product of the translation probabilities of all individual phrase pairs, multiplied by a value from the *distortion model*. The latter is meant to explicitly model the reordering of phrases, and it is essentially a *distance function* that discourages reordering to some degree, assigning a larger penalty as the distance grows. A distance function is not learnt from data, it is an a priori assumption about the nature of the data. To overcome this weakness, it is also possible to learn a proper lexicalized reordering model from data (Koehn et al., 2005; Galley and Manning, 2008).

But the one thing that certainly must be learnt from data is the phrase translation probabilities $\phi(\bar{f}_i|\bar{e}_i)$ in Equation 2.4. Learning those probabilities requires that there is a correspondence between the phrases of f and e, which is not straightforward and can be considered a "hidden" variable. The most prominent solution to this problem has been not to model phrase translation probabilities directly, but to compute alignments between *words* and extract phrases from the word alignment.

Word alignments emerge from data as a by-product of learning a word-based translation model. Introducing models based on single words after phrase-based models might seem like a step back, but phrase-based models have not superseded word-based models. Quite on the contrary, word-based models are an essential stepping stone for any current phrase-based model (Och and Ney, 2003). The most well-known word-based models are the *IBM models* developed at IBM (Brown et al., 1990, 1993)[4]

It is a fundamental principle of the IBM models that they need to be informed by alignments between

[3] Remember that we are actually looking for $p(e|f)$, namely for the probability that e is "generated from" f, but applying Bayes' theorem has flipped the sides again, which has caused a great deal of confusion (Koehn, 2010, 95).

[4] After more than 20 years, the original paper of Brown et al. still has the most elegant and intuitive explanation of both the IBM models and the EM algorithm for parameter estimation, in my opinion.

words and that word alignment should be modelled explicitly (unlike in phrase-based translation). In an IBM model, the translation probability of f becomes the joint probability of f and an alignment between f and e, summed over all possible alignments:

$$p(f|e) = \sum_{a} p(f, a|e) \tag{2.5}$$

and $p(f, a|e)$, in its most general form, is defined as follows. Let m be the length of the string f, let a_j be a specific alignment and

$$\mathbf{p(f,a \mid e)} = p(m|e) \prod_{j=1}^{m} p(a_j|a_1^{j-1}, f_1^{j-1}, m, e)\ p(f_j|a_1^{j}, f_1^{j-1}, m, e) \tag{2.6}$$

The several parts of Equation 2.6 can be described as the length probabilities $p(m|e)$, alignment probabilities $p(a_j|a_1^{j-1}, f_1^{j-1}, m, e)$ and translation probabilities $p(f_j|a_1^{j}, f_1^{j-1}, m, e)$. The actual IBM models 1 to 5 are simplifications of this general formula. For instance, IBM model 1 assumes that the length probability is a fixed, small value ϵ, that all alignments are equally likely and that the translation probability does not depend on the previous words or alignments.

In current phrase-based SMT systems, IBM models are never used directly for translation, but they are indispensable for "revealing" word alignments. The alignments are then used to extract phrases from the data and it is those phrases that the phrase translation model is populated with. A common strategy for phrase extraction is to compute word alignments in both directions (from the source language to the target and vice versa) and then symmetrize those alignments before phrases are extracted from them. In the end, this process will lead to a translation model $\phi(\bar{f}_i|\bar{e}_i)$ learnt from the unaligned[5] data.

After discussing the translation model, I will now describe the other significant part of Equation 2.3, the language model $p(e)$. There are a number of intuitive explanations for what exactly is modelled in a language model: while the translation model is said to focus on the *adequacy* of the translation (the degree to which information is transferred correctly to the target language), the language model is responsible for the *fluency* of the translation. Intuitively, a score from the language model can be interpreted as the likelihood that a specific string would be uttered by a native speaker of that language. Language models typically contain n-grams together with their probability, a frequent upper bound is 5-grams. Since a language model is only concerned with the target language, it can be trained on monolingual data, which is easier to find than parallel corpora. To account for unseen words, language models are usually *smoothed* (see e.g. Chen and Goodman, 1996).

In practice, computing the best translation simply by multiplying $p(f|e)$ with $p(e)$ does not offer enough flexibility. Instead, the translation model and language model should be thought of as interchangeable components of a more general model. Phrase-based SMT systems commonly use a *log-linear model* for this purpose. It can incorporate an arbitrary number of components (called "feature functions") and also introduces weights for each feature function (Koehn et al., 2005). Searching for the best translation then looks like[6]:

$$\hat{E} = argmax_e \sum_{m=1}^{M} \lambda_m h_m(e, f) \tag{2.7}$$

[5] That is, from data that is not word aligned. Sentence alignment is still a prerequisite for word alignment and phrase extraction. But sentence alignment is straightforward, compared to word alignment.

[6] This representation from Koehn et al. of a log-linear model is misleading to a certain degree, because decoders actually compute log probabilities (that is where the name for the model comes from), see e.g. Ortiz-Martínez et al. (2008, 162) for a more accurate equation.

Where h_1 ... h_M are feature functions and λ_m is the weight of each feature function. Combining components in this way is flexible because the weights can be changed without retraining the underlying models, and it is extensible because any other feature function can be added with ease.

In the training phase of phrase-based SMT, the models that underlie the feature functions in the log-linear model are trained, with data coming from a training set. At this point, the weights of all feature functions are initialized uniformly (i.e. each with an equal share of probability mass). The weights are then optimized with respect to another, smaller set of data in the *tuning phase*. The objective aim of the tuning step is to maximize or minimize an automatic metric of translation quality like BLEU (Papineni et al., 2002).

Finally, I would like to say a few words on the evaluation of phrase-based SMT systems. SMT systems are either evaluated with translation metrics that are fully automatic or manually by human experts. Human evaluation is rather expensive, so a lot of research has gone into the development of automatic metrics that approximate – or correlate with – human judgement. In a manual evaluation of machine translation, human subjects are given the machine translation and probably (but not necessarily) also the source segment and are asked to rate various aspects of translation quality, e.g. on a Likert scale. All automatic methods compute, in one way or another, the overlap between the machine-translated segment and one or multiple human reference translations.

This concludes the broad overview over the core methods used in phrase-based SMT I have intended to give. I have explained how the problem of statistical machine translation can be modelled as a noisy channel, which results in two probability distributions that must be learnt from data: the translation model and the language model. The translation model essentially is a distribution of phrase translation probabilities that are derived from word alignment. The language model is only concerned with the target language and can assign a probability score to sequences of n-grams. In phrase-based, statistical machine translation, both the translation model and language model are seen as interchangeable components of a log-linear model which can easily be extended with other abitrary feature functions.

To round off this section, I will now mention actual implementations of the processes I have described above. A lot of the mechanisms are available in **Moses**, an open-source toolkit for statistical machine translation (Koehn et al., 2007). At its heart, Moses contains convenient libraries such as those for computing word alignment, the one that is implemented in Moses is `GIZA++` (Och and Ney, 2003) and `fast_align` (Dyer et al., 2013) is a very prominent alternative. Both of them are variants of the original IBM models.

For language modelling, there is a wide range of implementations: `RandLM` (Talbot and Osborne, 2007), `SRILM` (Stolcke et al., 2002), `IRSTLM` (Federico et al., 2008) and `KenLM` (Heafield, 2011), to just cite a small selection of them[7]. At the time of writing, `KenLM` is the most widely used language model and the most versatile (in my opinion). Tuning the weights of the log-linear model is done with minimum error-rate training (MERT; Och, 2003) or the "margin infused relaxed algorithm" (MIRA; Cherry and Foster, 2012).

To find the best hypothesis at translation time, Moses has a built-in decoder. Other useful decoders include `cdec` (Dyer et al., 2010) and `Thot` (Ortiz-Martínez and Casacuberta, 2014), which is actually a full-flegded SMT toolkit that includes online learning, for instance. Finally, some of the well-established metrics to measure the translation quality of phrase-based SMT systems are: BLEU (Papineni et al., 2002), TER (Snover et al., 2006) and METEOR (Banerjee and Lavie, 2005; Denkowski and Lavie, 2011).

[7] See *http://www.statmt.org/moses/?n=FactoredTraining.BuildingLanguageModel* for a complete list of language models that can be used with Moses.

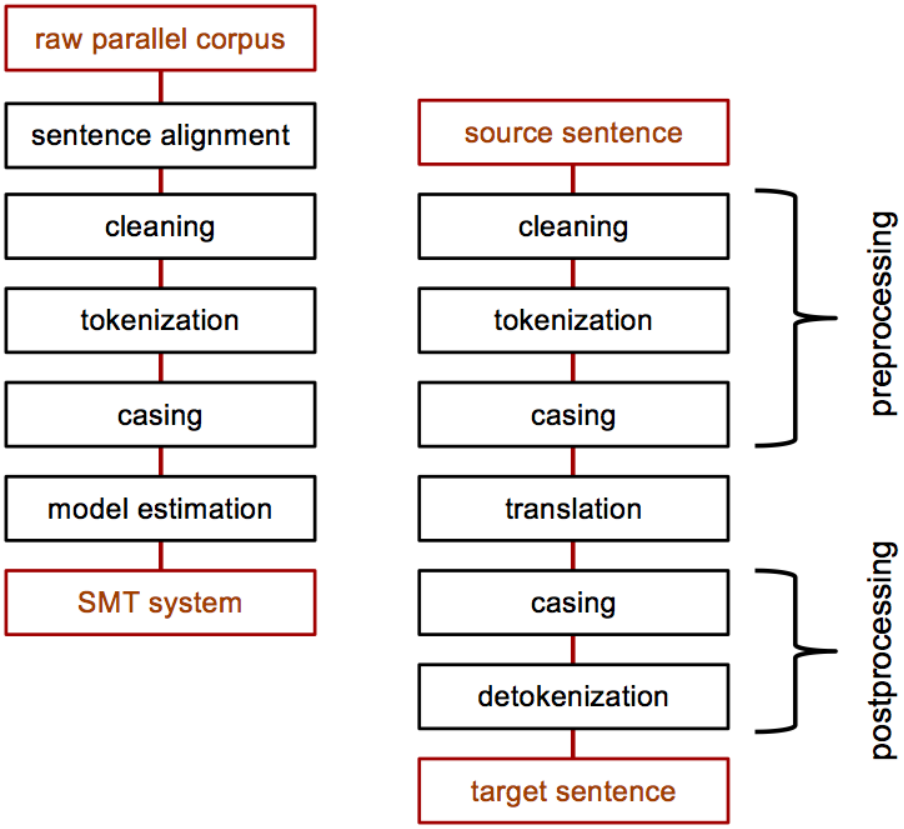

Figure 2.2: Pre- and postprocessing steps typically involved in phrase-based SMT systems for European languages. Left: preprocessing before the models are trained. Right: processing that takes place before and after translation.

2.1.3 Common Pre- and Postprocessing Steps

Raw parallel text is the basis upon which translation systems are built, but a number of steps are required before the text can be used to train a model or be fed to a decoder for translation. These steps that are applied to each segment of input and precede both training and translation are called *preprocessing steps*. Likewise, a number of *postprocessing steps* are applied after translation. Figure 2.2 shows a pipeline of a phrase-based SMT system that is very typical for European languages. Other languages and writing systems have different requirements. For instance, Chinese text is not tokenized, but *segmented* and the concept of casing does not apply to Chinese. But in many cases, such a pipeline is an instructive approximation.

It should be noted that the model training mechanisms discussed in the previous Section 2.1.2 are now represented as a single box, "model estimation". But all other steps before and after those core methods are equally important, especially because upstream changes in the preprocessing may have a fundamental impact on all the downstream processes in such a pipeline. For instance, the impact of trying a different tokenization method can be more incisive than changing a feature of the translation model. The meaning of some of the steps shown in Figure 2.2 is not obvious and I will explain them now:

- **sentence alignment**: splits a continuous stream of input text into smaller pieces that should roughly correspond to sentences, e.g. using Hunalign (Varga et al., 2007). In many cases, raw parallel corpora

are already provided in a sentence-aligned format so that this step can be skipped often.

- **cleaning**: an umbrella term for sanitizing input data, for instance by removing markup from the input, or removing sentences that are too long, or deleting strings specific to the data set that are known to cause trouble later on.
- **casing**: lowercasing the input text can help to make the statistics less sparse because it will collapse the counts for tokens that were capitalized for some reason, e.g. appeared at the beginning of sentences and others that did not. After translation, casing must be restored in the target sentence of course.

When I discuss the actual contributions of this thesis in the later chapters, it will become evident that the problem of markup handling, in all cases, is solved in the periphery of the system. More precisely, markup handling is implemented as pre- and postprocessing steps rather than as more fundamental changes to model estimation or decoding.

2.2 Markup Languages and Their Uses in Machine Translation

Markup languages are useful in a wide variety of contexts, so it is not surprising that SMT systems have uses for them and, on the other hand, face challenges that have to do with markup. This section will properly introduce the concept of markup languages (in Section 2.2.1) and I will argue that XML should not only be regarded as additional information, but also as translatable content (in Section 2.2.2). Finally, Section 2.2.3 will investigate how XML markup is treated by default in Moses, a widely used machine translation framework.

2.2.1 Markup Languages, Giving Special Consideration to XML

An intuitive way of grasping the concept of a markup language is the following[8]. When resources are stored in our digital age, they are not stored merely as raw data. For many applications, data is only useful if information is added on top of the actual content. Additional information can either mean meta data (for example, the time and place a certain resource has come into being) or any description, interpretation or analysis of – even imposing structure on – the data. It is wise to uphold a strict separation between the actual data and any additional information, much in the same way as web development tells us to always separate content from presentation.

Markup is a technique, usually with a finite set of key words or instructions to make the result machine-readable, that allows annotating the data with additional information while making sure that the two dimensions are never confused. The result is a structured document written in a markup language like XML or HTML. Figure 2.3 shows an example of an XML document, where pieces of information about a certain kind of fish, "cichlids", are embedded into XML elements and enriched with attributes. In this work, the only markup language I will be using and describing is XML. For an introduction to HTML please look elsewhere, although many of the concepts and rules of XML also apply to HTML since they share a common ancestor: SGML.

XML is shorthand for "eXtensible Markup Language" and is an official standard endorsed by the W3C, the "Word Wide Web Consortium" – an intergovernmental and independent body that develops and maintains web standards. XML is widely used for protocolled information interchange, as a versatile storage format and e.g. for text annotations in the digital humanities. XML is a so-called *meta*

[8] This introduction to markup languages is adapted from an unpublished technical report of mine, on the subject of the feasibilty of stand-off markup in TEI documents, a standard used in digital humanities.

```
<?xml version="1.0" encoding="UTF−8" ?>
<cichlids>
        <cichlid ID="c1">
                <name>Zeus</name>
                <color>gold</color>
                <teeth>molariform</teeth>
                <breeding_type>lekking</breeding_type>
        </cichlid>
        <cichlid ID="c2">
                <name>Nemo</name>
                <color>green</color>
                <teeth>molariform</teeth>
                <breeding_type>harem</breeding_type>
        </cichlid>
</cichlids>
```

Figure 2.3: Simple example of a complete XML document (inspired by Barlow 2002)

markup language because it defines the building blocks of documents and general rules for how documents can be composed, but it says very little on the *semantics* of those components. For instance, the XML standard defines the existence of elements and attributes and how they interact with each other, but it does not define what a certain element or attribute name means. Defining a vocabulary (a set of allowed names) is done individually for each application, and it is only with a well-defined vocabulary that elements and attributes can be ascribed meaning.

Since it is likely that some of the readers are unaware of the scale at which XML permeates the internet and computing world, I will give a few examples of actual XML vocabularies:

- **RSS**: Internet feeds are structured according to the RSS standard, which is an XML vocabulary. If you subscribe to a news feed, you are sent a payload of XML markup.
- **SVG**: All major browsers support a specific type of vector graphic: SVG (scalable vector graphics), which again, is XML markup.
- **OOXML**: Text processing software typically stores your documents in an XML format. For instance, documents saved with a recent version of Microsoft Word are stored as "Office OpenXML", a zipped-XML standard approved by OASIS. At the time Microsoft introduced this format, there already was a standard called "OpenOffice XML" used in free software like LibreOffice – which has led to confusion over and over again.
- **TEI**: A community based in the digital humanities curates the "Text Encoding Initiative" standard, an elaborate XML vocabulary and tools for the encoding of scholarly texts.

Many more examples could be given, but suffice it to say that XML and related markup languages are an omnipresent phenomenon and that research in text processing frequently involves coming to terms with XML. But why is XML a viable alternative to e.g. comma-separated data that is much less verbose or a binary format that uses much less space? In the following paragraph, I will summarize the advantages of using XML to store or exchange information.

By now, XML is a well-tried standard that underwent virtually no changes since 1996 when most of today's rules were laid down. Most applications are still expecting XML version 1.0, version 1.1 has

introduced a few changes to the standard, but its use never caught on. The stability of the standard means that XML comes with a wide range of co-standards that can be used in conjunction with XML itself: XML Schema, XML Namespaces, XPath, XQuery and XSLT, to name the most important of those standards. But the fact that XML is a very stable standard also implies that there is rarely a need to reinvent the wheel: many programming languages and environments provide libraries for XML processing by default. High-level languages like Perl, Python, Java or C++ are capable of dealing with XML out of the box, without writing any additional code.

But the true unique selling points of XML that set it apart from other means of storage or interchange are the following:

- the format is versatile in the sense that both machines and humans can read it, unlike binary formats (PDF for instance),
- XML is a strict markup language, with tight rules for the structure of documents,
- using XML together with a schema language gives developers fine-grained control over the structure and content of documents.

It is especially the rules for XML documents that make XML worthwile. As I have explained above, some of the rules apply to all XML documents, and the property of respecting all those rules is called *well-formedness*. Checking the well-formedness of an "alleged" XML document is a very straightforward process and it creates a useful bottleneck for robust processing: ill-formed documents can be rejected very early on in the application. For instance, if an application receives a payload of XML from a server, it could notice at the start of processing that a part of the server's answer is missing if the XML in it is not well-formed.

For individual use cases, the structure and content of XML documents can be controlled in much more detail, using a schema language like DTD, XML Schema, RNG or Schematron. A schema language defines a set of allowed names and the structural organization of a compliant document, rules that go beyond the general well-formedness of XML documents. If a documents complies with the rules defined in such a schema, it is said to be *valid* with respect to this particular schema. This makes XML processing even more robust, since the application only needs to process the specific subset of XML documents for which it was written. As an example, consider that a client and a server need to exchange information. Any communication between them will be greatly facilitated if they speak the same language, i.e. if they negotiate with a mutual protocol. This protocol could be written in a schema language, a real-world example for this is XML-RPC.

But the question I have not answered fully yet is how markup is related to machine translation. The answer is two-fold: on the one hand, markup is a vital part of documents that are processed with machine translation (see Section 2.2.2) and on the other hand, machine translation frameworks have internal uses for XML, for instance in decoding (see Section 2.2.3).

2.2.2 XML as Translatable Content

The purpose of this section is to establish that XML does not only "mark up" text but can actually be content that is translated by human translators. I will argue that there is a need for machine translation systems to treat XML as content and translate it.

While "SMT researchers often give themselves the luxury of pretending that only pure text matters" (Joanis et al., 2013, 73), today's translation professionals typically do not work on raw text. Thinking that translators are only concerned with how to best get the meaning across languages is a romantic idea, but nowadays, the duties of translators are defined by business needs. The business needs in turn are defined by the customers that the translation industry services. What those customers expect from

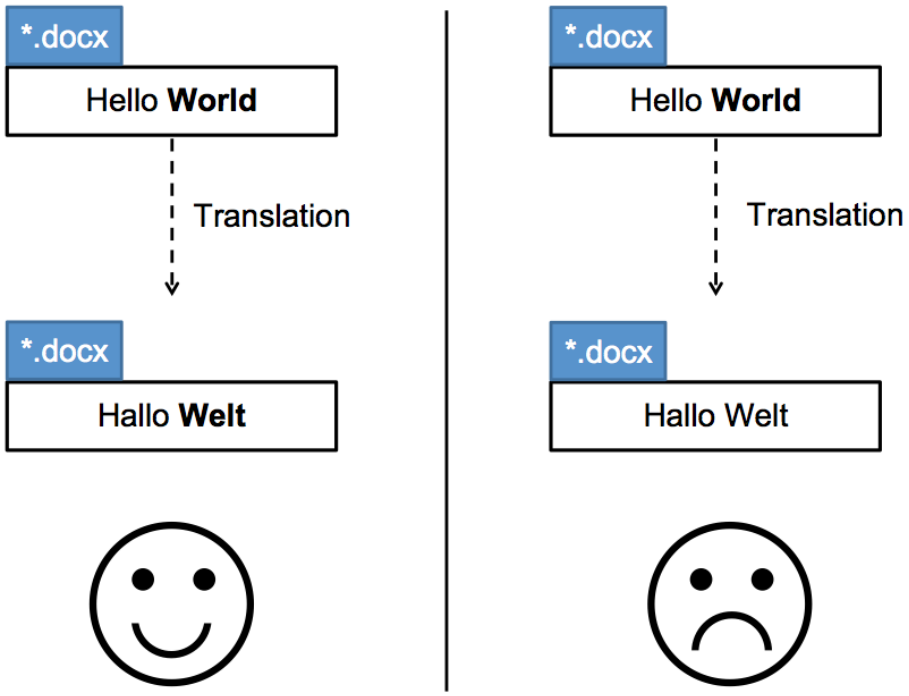

Figure 2.4: Expectation of customers regarding the translation of formatted content

a language service provider (LSP) is not the translation of text only, but the translation of *documents* that encapsulate the text.

Companies have translatable content in a variety of document types, including HTML, Microsoft Word and PDF, all of which have formatted content. Their expectation clearly is that if there is formatting in the source language, that formatting should carry over to the target language (see Figure 2.4 for a rather light-hearted illustration). Information about formatting is vital in this case and it also carries a part of the meaning intended by the author. A translator must respect the formatting, or put another way, they need to translate formatting exactly in the same way they translate the text itself.

But to know what it means to preserve the formatting in documents that have it, we first need to discuss how formatting is represented in those documents. It can be encoded in an arbitrary way of course, but a common solution is to use markup for formatting. This is the case for documents in Microsoft Word, which I would like to use as an example.

Figure 2.5 gives an example of a small Microsoft Word document that is stored in the Office OpenXML format. A *.docx file is actually a zip archive that can be unzipped to reveal its content. Among the files in such a folder, there is a file named `document.xml` that holds the main content of the document. In this case, the only thing that was typed into a newly created Word document is "Hello **World!**", with exactly the formatting shown here. In the abbreviated XML document shown in the figure, you can see that "World!", the string that is formatted in bold, is accompanied by a `<w:b>` element inside a `<w:rPr>` element (the so-called "run properties"). The `<w:b>` element informs a text processing application that any text that appears next to it must be set in bold. Therefore, if the formatting of this document should be kept in a translation, it is actually the `<w:b>` element that must be preserved.

Yet, in many translation workflows, translators do not work on the source formats directly, but on a an intermediate format that can hold all the necessary information and be converted back to the original format once the translation is done. Having such a standard simplifies workflows consid-

```
<?xml version="1.0" encoding="UTF−8" standalone="yes"?>
<w:document mc:Ignorable="w14 wp14">
    <w:body>
        <w:p w:rsidR="00F677AB" w:rsidRDefault="00392E8B">
            <w:r>
                <w:t xml:space="preserve">Hello </w:t>
            </w:r>
            <w:r w:rsidRPr="00392E8B">
                <w:rPr>
                    <w:b/>
                </w:rPr>
                <w:t>World!</w:t>
            </w:r>
            <w:bookmarkStart w:id="0" w:name="_GoBack"/>
            <w:bookmarkEnd w:id="0"/>
        </w:p>
    </w:body>
</w:document>
```

Figure 2.5: The main body of a minimal Microsoft Word document (*.docx) that only contains "Hello **World!**", after unzipping, without namespace declarations and section properties

erably because it unifies the plethora of different input formats and because it facilitates exchanging translation data. The most elaborate and widely used such standard is XLIFF (OASIS XLIFF Technical Committee, 2008), which stands for "XML Localisation Interchange File Format" and, needless to say, it is an XML standard that defines an XML vocabulary. XLIFF 1.2 is supported by a great many tools (Filip and Morado Vázquez, 2013), both commercial computer-assisted tools (CAT) like Trados and free frameworks like Okapi, and support for the recently published XLIFF 2.0 is growing (Morado Vázquez and Filip, 2014).

XLIFF is used alongside with another popular XML standard, TMX (Translation Memory eXchange)[9]. TMX has a slightly different purpose, namely to make it easier to exchange translation memories between different tools and projects. But in principle, both XLIFF and TMX address the same basic need: the ability to efficiently work on or exchange translation data without losing critical information. So, if translators actually work on[10] XLIFF documents, what does the Word document shown in Figure 2.5 look like, once a translator gets to see it and how is the formatting preserved exactly?

An XLIFF 1.2 version of the Word document is shown in Figure 2.6, converted to XLIFF with *Rainbow*, a tool from the Okapi framework, a "free, open-source and cross-platform set of components and applications designed to help your localization and translation processes"[11]. Now, the formatting is repre-

[9] TMX was initially developed by the Localisation Industry Standards Association (LISA), but the most recent version TMX 1.4b is now hosted by the Globalization and Localization Association (GALA), here: *https://www.gala-global.org/tmx-14b*, see e.g. *http://www.opentag.com/tmx.htm* for more information.

[10] Or, rather, that CAT tools actually work on. The translators themselves might have a view or user interface that abstracts completely from the underlying format.

[11] See *http://okapiframework.org/wiki/index.php?title=Main_Page* for more information and *https://bintray.com/okapi/Distribution/* to download for free.

```
<?xml version="1.0" encoding="UTF-8"?>
<xliff version="1.2" >
  <file original="word/document.xml" source-language="en-us" target-language="de-de"
  datatype="x-undefined">
    <body>
      <trans-unit id="NFDBB2FA9-tu1" xml:space="preserve">
        <source xml:lang="en-us">Hello <g id="1" ctype="x-bold;">World!</g></source>
        <target xml:lang="de-de">Hello <g id="1" ctype="x-bold;">World!</g></target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Figure 2.6: Content from a Word document converted to XLIFF 1.2, only showing essential parts without additional files and namespace declarations in the outermost element

sented with a <g> element, a generic indicator for actual inline markup that was present in the original document. The type of inline markup is indicated by the `ctype` attribute, whose value is "x-bold;" in this case, a user-defined type for bold face text. Thus, in XLIFF, one of several means to represent inline formatting is by using <g> elements in the translation units directly. The document shown in Figure 2.6 can finally be read by current CAT tools and it could be part of a translation assignment sent to a translator by an LSP, for instance.

To sum up this section, XLIFF and TMX are the predominant formats in the translation industry, and virtually all commercial products can process those formats. When converting to XLIFF or TMX, the formatting of the original format is kept or referenced. Because of this, inline elements are actually a part of the content that needs to be translated. If machine translation is to be a convenient tool that assists human translators, it must be able to treat inline markup properly.

2.2.3 Inability of Standard Machine Translation Tools to Treat XML as Content

In this section I will discuss how current, out-of-the-box SMT systems treat markup if it is present in the training or translation input. It will become clear that the developers did not intend their systems to translate XML markup, but had other uses in mind for markup. Throughout the section I will discuss the capabilities and limitations of Moses, currently the most popular SMT framework by far and will not mention other systems that might be better equipped to translate XML as content.

How does a standard, out-of-the-box phrase-based Moses system cope with markup in the input? We will further develop the examples given in Figures 2.5 and 2.6 which are valid use cases for machine-assisted translation. In an XLIFF file, the only information relevant for automatic translation is the content of the `<source>` element. As soon as this content is extracted from XLIFF, a number of preprocessing steps are applied to it (see Section 2.1.3 for more detail). After preprocessing, the string is handed to the decoder for translation, together with a reference to trained models on the file system. After decoding, the raw translation is postprocessed to undo many of the preprocessing steps and for final touches.

Table 2.2 shows a number of pre- and postprocessing steps usually performed when a sentence is translated with a standard, phrase-based Moses system. Tokenization and lowercasing (and their coun-

original	`Hello <g id="1" ctype="x-bold;">World!</g>`
after tokenization	`Hello < g id = " 1 " ctype = " x-bold ; " > World ! < / g >`
after escaping	`Hello &lt; g id = &quot; 1 &quot; ctype = &quot; x-bold ; &quot; &gt; World ! &lt; / g &gt;`
after lowercasing	`hello &lt; g id = &quot; 1 &quot; ctype = &quot; x-bold ; &quot; &gt; world ! &lt; / g &gt;`
after translation	`guten tag &lt; code &amp;lt; ept id = &amp;quot; 1 &amp;quot; ctype = &quot; &quot; &quot; x-bold ; &quot; &gt; world money &lt; / g &gt; !`
after recasing	`Guten Tag &lt; Code &amp;lt; ept id = &amp;quot; 1 &amp;quot; ctype = &quot; &quot; &quot; x-bold ; &quot; &gt; World Money &lt; / G &gt; !`
after detokenization	`Guten Tag &amp;lt; Code &amp;lt; ept id = &amp;quot; 1 &amp;quot; ctype = "" "x-bold;" &gt; World Money &amp;lt; / G &gt;!`
after unescaping	`Guten Tag < Code &lt; ept id = &quot; 1 &quot; ctype = "" "x-bold;" > World Money < / G >!`

Table 2.2: Effect of out-of-the-box Moses tools on text that contains inline XML markup

terparts in postprocessing, detokenization and recasing) are quite universal for European languages, and escaping is a hard requirement because of the decoder's internals, as we will see. The table highlights several problems that arise if there is XML markup in an input string during translation:

- Tearing apart XML markup in this way during tokenization dramatically increases the number of tokens in each segment and this will lead to worse word alignments. It is a well-known fact that the quality of word alignments decreases as the sentence length (in tokens) grows. This also leads to more segments being discarded from training because of their length, as most systems have a threshold for the maximum number of tokens in a segment (again, because of word alignment).
- Standard lowercasing and recasing do not make sense either if the input is XML, since capitalization in XML names is significant, i.e. it is case-sensitive. Lowercasing the name of an XML element changes the markup in a fundamental way, and there is no way to guarantee that the recasing step will consistently restore the initial casing. Also, all characters of XML names can be capitalized, while lowercasing and recasing are primarily concerned with the casing of the first letter of tokens.
- XML markup should definitely be transferred to the target language, but it should not be translated in a probabilistic fashion with candidates from the phrase table, and least of all should this be done character by character. The translation of XML markup is fixed in most cases, except perhaps for attribute values which could be translatable under certain circumstances, but not in the general case.

So, on the whole, the typical chain of steps taken before, during and after translation in a standard, phrase-based Moses system does not deal properly with XML markup. In fact, the translation might be utterly useless if there was markup in the source sentence. There are two feasible solutions for the problems discussed so far: either "hide" XML markup in training and translation by replacing it with an innocuous placeholder, or remove all markup before training and translation, and try to reinsert afterwards. Additionally, individual tools in the chain (e.g. tokenization) can be made aware of markup and customized to either treat markup in the right way or ignore it alltogether. Later sections will explain all approaches in detail.

While it is clear that a standard Moses system is unable to treat XML markup as *content*, it has other uses for markup, in several places. There is, for example, a number of ways to provide the decoder with additional information about the translation[12]:

- **forced translations**: parts of the input sentence can be wrapped in XML and a fixed translation can be forced onto the decoder, or an additional translation can be suggested to the decoder, together with its probability if needed.
- **zones and walls**: the user or application that requests a translation can set constraints on the reordering of the output sentence. A **wall** is a hard boundary which tokens cannot cross during reordering. A **zone** wraps a number of tokens that should only be moved together, and cannot move individually (Koehn and Haddow, 2009). Both zones and walls are represented as unescaped XML elements.
- **alternate weight settings**: for a single input sentence, alternate weight settings can be specified in XML markup to override the default setting for each feature function of the log-linear model. Using this mechanism, the influence of features can be changed on the level of individual sentences.

These are some of the decoding features that are implemented using XML, but XML is also used in a few other places. Most notably, XML is used if Moses is run in server mode. A Moses server implements the XML-RPC protocol to communicate with a client, which must also adhere to this protocol. The payload of an XML-RPC request or answer is XML.

So, although there are many internal uses for XML in the Moses framework, a standard Moses system is incapable of treating XML as content that should be translated. Additional layers of pre- and postprocessing must take care of handling XML markup in suitable ways, all of which I will explore in Sections 3 and 4. But before, the next section will explain how markup handling in machine translation ties into research on translation processes.

2.3 Translation Process Research

Machine translation is used widely, not only in academic institutions, but in companies that have a need for effective translation. Especially for technical domains, it is extremely common to pre-translate content with an MT system and then have a human translator check the result and correct errors. Incidentally, this led to translators being called "post-editors" very frequently. Since machine translation and human translation now have a common cause, their goals have started to align in recent years and in the course of this, the effectiveness of MT systems has been called into question because evaluations typically did not involve real translators. But in the meantime, several studies have made up for this lack of validity and have proven the effectiveness of MT in realistic translation contexts.

Here, I will explain shortly what has been done to make MT blend in with the translation industry (in Section 2.3.1), what translator productivity is (in Section 2.3.2) and how translator productivity is reduced if XML markup is lost in translation (in Section 2.3.3).

[12] See *http://www.statmt.org/moses/?n=Advanced.Hybrid*, *http://www.statmt.org/moses/?n=Advanced.Domain* and Koehn and Senellart (2010) for more detailed explanations.

2.3.1 Bridging the Gap Between MT and CAT Tools

While considerable progress has been made to integrate MT into CAT tools and the translation industry in general, the tools are still rough around the edges and academic successes do not automatically carry over to industry. An early attempt at bridging the gap between MT and CAT tools is Koehn and Senellart (2010). They recognize that until then, TM and SMT tools "have been developed very much in isolation" (Koehn and Senellart, 2010, 21) and they outline ways to combine TM pre-translation with an SMT system, for cases where the fuzzy match score[13] from a TM is too low.

MT has adapted to the post-editing setting in a variety of other ways. Several projects were dedicated to developing interactive machine translation that assists post-editors in real time (Koehn, 2009; Alabau et al., 2013; Federico et al., 2014) and to adapt to new information from a human user by adapting the models online (e.g. Martínez-Gómez et al. 2012). Handling formatted input with inline XML markup properly is another step into the general direction of being useful for translators and the translation industry as a whole.

2.3.2 Translator Productivity

In a CAT tool context, automatic metrics of translation quality like BLEU do not necessarily say much about the usefulness and impact of the MT tool. The only real and measurable outcome of the translation process is the *productivity* of the translator, which for example can mean time taken per segment or how many characters needed to be post-edited in a machine-translated output. If there is a relationship between automatic SMT metrics and translator productivity, the onus is on MT and translation process research to prove it (O'Brien, 2011).

But rather than investigate the aforementioned relationship, researchers have taken to measuring directly whether MT systems increase the productivity of translators. After a few initial experiments that suffered from unrealistic experimental conditions, such as that the translators had to work with an interface they were not familiar with or had to "think out loud" (Krings, 2001), several studies have produced sound proof that MT is cost-effective and increases productivity (Green et al., 2013; Läubli et al., 2013).

2.3.3 Impact of Wrongly Translated Markup on Translation Processes

If translator productivity is the only outcome that is relevant in the end, one can of course take this one step further and inquire what exactly it is that causes a translator to be less productive. Put another way: which aspect of an MT system is in the way of being productive, what is missing? And by giving one of many possible answers to this question, I can finally touch on the *raison d'être* of my thesis and explain why it solves a problem that is worth solving.

One of many answers, then, is that one of the reasons translators are less productive is the translation of markup. Or rather, the inexistence of standard solutions for transferring markup to the target language so that post-editing would actually require less effort than checking back to the original markup and reinsert it manually. Several authors have already pointed out that might be the case: Tezcan and Vandeghinste (2011, 56) argue that an MT system should handle XML markup correctly to avoid inefficient translation workflows. In the same vein, Joanis et al. (2013, 74) say that "post-editing SMT output without the formatting information found in the source may represent a serious loss of productivity".

[13] A fuzzy match score is a metric commonly used in CAT tools to estimate the similarity between entries of a TM and an untranslated segment. It does not compute exact matches, hence the name "fuzzy". If the score of a TM entry is high enough (a common threshold is 85 %), the segment is pre-translated with the TM entry.

From a slightly different perspective, Escartín and Arcedillo (2015) have conducted experiments to find good predictors for post-editing productivity. Their goal was to test whether the fuzzy match score (FMS) correlates with productivity, as it would be easier to interpret than traditional MT metrics. In their conclusions, they state "that inline tags have a big impact on productivity, a fact which is not reflected in any of the known metrics and which has not yet received much attention in research" (ibid., 142).

I agree with their assessment and, as I see it, this further piles on the evidence that markup handling in machine translation is a problem worth solving.

3 Related Work

In this section, I will present ealier approaches to solving exactly the same problem, that is, handling XML markup in machine translation. I have tried to be very thorough and therefore this is, to the best of my knowledge, a complete list of solutions, both published in an academic sense or otherwise disseminated. There are remarkably few published papers that deal with this topic and not all of them present experimental results or a clear algorithm to follow. I will also mention work that has no record of academic publications but is nevertheless publicly available as open-source software and comment tentatively on how commercial products have solved this problem.

3.1 Published Literature

There are only a handful of scientific papers published about markup handling in statistical machine translation. A small handful in fact, it is the main topic of only four published works, and an additional one discusses such a method at length, even though markup handling is not the main focus in that case. I will report on each of those works individually, with a clear focus on the proposed *methods*, rather than the data sets or the results. The methods should be general enough to not depend on any specific data set or XML vocabulary and the results, while they are worth mentioning, are not comparable between papers. Finally, I offer fair criticism in each case, explaining the merit of the proposed methods and their disadvantages.

3.1.1 Du et al. (2010)

Du et al. (2010) present three methods to process TMX markup in an SMT system and compare the performance against a "customized SYSTRAN system", which presumably implies a rule-based system. They also have an extensive evaluation protocol that includes automatic metrics and human ratings of the results. The three different methods they propose are (using the original names given in the paper):

- **Markup transformation**: the SMT system is trained without the markup. Before translation, TMX elements are replaced with `<zone>` elements (see Section 2.2.3). The decoder is invoked with the `-t` option which instructs it to report the segmentation, i.e. how the output sentence is made up of phrases from the phrase table. After translation, the markup is restored based on the original source sentence that contains the markup, the target sentence and the segmentation.
- **Complete tokenization**: Do nothing, let the standard tokenizer rip apart (also see Section 2.2.3) the markup and escape it, both during training and translation. By far the easiest method, and worth testing since their paper was the first published work to ever discuss this issue.
- **Partial tokenization**: The third method also keeps the markup during training and translation, but the tokenization process is less aggressive in the sense that opening and closing tags of XML elements are kept together as a single token. For instance, while complete tokenization would turn `<ph>` into `< ph >`, partial tokenization does not split this tag.

Du et al. (2010) report the following overall results: partial tokenization worked best according to a number of different automatic metrics (BLEU, NIST[14], METEOR and TER), when translating from French to English, albeit by very small margins. When translating from English to French, complete tokenization worked best according to three out of four metrics. Again, for most of the figures reported,

[14]National Institute of Standards and Technology, but also stands for a popular machine translation metric. See Doddington (2002) for a detailed description.

the difference between complete and partial tokenization is negligible. Human subjects that were asked to rate samples of translations with markup in them said that more than half of the segments only required minor corrections during post-editing.

The analysis of the findings and the sources of error that ensues is exemplary, but all methods are problematic and their success can only be very limited. I will briefly comment on each of the three methods and explain what the limitations are:

- **Markup transformation**: for sentences that contain several tags, this method will severly hinder reordering and decoding will approximate monotonicity. For language pairs that have very different word order, this might affect the overall translation quality. In general, removing markup from the training corpus and before translation is a worthwile idea, as it does not burden the core modules of the SMT system with the markup at all; a dedicated module can deal with the markup. But this module should work with word alignments if possible because they are more precise than phrase segmentation[15].
- **Complete tokenization**: As I have discussed earlier, letting the standard tokenizer split markup is a questionable idea, since this dramatically increases the number of tokens per sentence. As a consequence, more overly long sentences are removed from training and word alignment quality will be worse for the remaining corpus. Also, if the decoder is free to choose translations for each of those one-character tokens and reorder them, one cannot hope to ever be able to put the markup back together. This is unsatisfactory because the translation for XML markup is actually known beforehand, namely it should not be translated at all. Also, the detokenizer would have to be able to merge the split XML markup after translation, which the standard version in Moses cannot do.
  As the final straw, complete tokenization of markup jeopardizes the results of automatic metrics of translation quality like BLEU because the metrics are biased towards longer translations.
- **Partial tokenization**: Everything about complete tokenization above applies, but having a custom tokenizer that keeps together XML content is a good approach. Then, any downstream tool (tokenization always is one of the first preprocessing steps) can still parse and use the markup. But then, a lot more thought has to go into modifiying the tokenizer. For instance, their partial tokenization method does not keep together the attributes of an element, as far as I can see.

Keeping in mind that this paper by Du et al. (2010) is the first published work on this topic, they very successfully set the stage and frame the problem, propose three methods and test them in well thought-out experiments. They are aware that "in future work, we have to find a more effective and efficient way to facilitate TMX-based SMT system [sic]" (ibid.). Although they envision a solution for TMX input, all of their methods seem to be general enough to work with other XML vocabularies.

3.1.2 Zhechev and van Genabith (2010)

In Zhechev and van Genabith (2010), implementing a method to handle markup is not the primary goal, but they do present an approach to solve this problem. The actual goal of the paper is to supplement a TM with SMT, in case there is no good match in the TM for a new source segment. The novel idea presented in the paper is that the parts of a sentence that should be translated are identified with subtree alignment, but I will only explain how they dealt with markup in the input texts.

The authors are aware that a standard SMT system should not be trusted with processing XML markup on its own, saying that "letting any MT system deal with these tags in a probabilistic manner

[15] In 2010, Moses did not provide an option to report word alignments yet, so the authors of this paper would have had to align the source and target sentence themselves after translation.

can easily result in ill-formed, mis-translated and/or out-of-order meta-tags in the translation" (ibid.). What they have implemented to avoid those problems is the first incarnation of replacing markup with placeholders: they replaced XML tags with IDs that act as a placeholder for the actual markup. All IDs were unique on a global level, i.e. throughout the whole corpus.

They do not present any results that could be related to their approach to markup handling, as is expected because for them, handling markup was only an auxiliary tool. Nothing in the results or discussion indicates that the way of handling markup could have had an influence on their experimental results. However, their system actually outperforms the baseline system for FMS values between 80 and 100 %, which means that tree-based structural alignment is a promising approach.

Even if the authors do not evaluate or otherwise comment on the markup handling method they have developed, it is clear that it will lead to a high number of unique IDs in the training data. Each time a new XML tag is encountered, a new ID is created for it and it is replaced with this ID. Again, it is unclear what exactly happens to attributes in this process. In any case, replacing all occurrences `<ph>` with something like `@@4673@@` means that the chance of out-of-vocabulary words during translation is high and that there is very little evidence for individual IDs. If an ID is seen only very few times in training, the models do not have sound statistics for this token.

3.1.3 Hudík and Ruopp (2011)

Hudík and Ruopp (2011) is a very serious attempt at solving the markup handling problem, but no experimental results are reported. They see their work as a follow-up to Du et al. (2010), trying to improve the method "markup transformation" (see Section 3.1.1). They improved the method in the sense that they do not introduce `<zone>` elements into the input that could hinder reordering and they streamlined the algorithm that reinserts markup into translated text.

In this case, XLIFF is used as the input format, where the patterns of XML markup are different from the ones found in TMX markup (in general, XLIFF has tidier markup with fewer nested structures). Their approach is to remove XML markup completely before training, so that the trained models have nothing to do with XML markup. During translation, XML is also removed, but the original version of the input file is kept for reference. Using this reference and the phrase segmentation reported by Moses, the markup is reinserted into the translation.

Unfortunately, there are no experiments in this paper and the authors did not critically evaluate their approach or compare it to other methods. To make up for this lack of scientific rigour, the source code of the project is publicly available[16] and the reinsertion algorithm is described very clearly with pseudo code in the paper. This has helped me understand the code and reimplement a similar solution. When looking at the code more closely, it becomes evident that it is clearly only meant for a specific XML vocabulary, *InlineText*, an intermediate XML format that can be converted to and from XLIFF. This means that although the proposed method is general enough, the actual implementation only works if the input is InlineText.

Like Du et al. (2010), Hudík and Ruopp (2011) rely on phrase segmentation to reinsert markup into the translation, which means that markup can only be reinserted between phrases. The exact location where markup should be reinserted does not have to coincide with a phrase boundary, however and that is why the placement of tags will sometimes be off because of the fuzzy nature of phrase segmentation. Because of this limitation of phrase segmentation, the code was later changed and now uses word alignment instead to guide the reinsertion process[17].

[16] Inspect the code here: *https://github.com/achimr/m4loc*, and find more information here: *http://www.digitalsilkroad.net/m4loc.pdf*.

[17] Achim Ruopp told me he changed the code to make it work with word alignments as soon as Moses offerred the option to report

3.1.4 Tezcan and Vandeghinste (2011)

The work of Tezcan and Vandeghinste (2011) originated at roughly the same time as Hudík and Ruopp (2011), the papers were even submitted to the same conference. The former borrows ideas from the pioneering work of Du et al. (2010) and Zhechev and van Genabith (2010). Tezcan and Vandeghinste (2011) propose four different methods of handling markup in SMT (again, using the original names given in the paper):

- **full tokenization**: do nothing at all, let the standard tools deal with markup even though they are ill-equipped, equivalent to "complete tokenization" in Du et al. (2010).
- **full markup normalization**: all XML elements are replaced with one single placeholder, `@tag@`. Models are trained with data that contains the placeholder, and placeholders are inserted before translation. After translation, markup is reinserted, presumably with phrase segmentation (it is not stated clearly in the paper).
- **role-based markup normalization**: similar to the second method, but there are several placeholders, based on the names of elements. Roughly equivalent to the method of Zhechev and van Genabith (2010). The rationale for this method is: "as different tags are used in different contexts, just like words, this method helps to distinguish certain contextual differences while creating better phrase and word alignments" (Tezcan and Vandeghinste, 2011, 57).
- **role-based markup normalization, xml input**: similar to the third method, except that the translation of the placeholder is forced by wrapping it into XML and suggesting the translation to the decoder in this way.

In contrast to Hudík and Ruopp (2011), the authors do not explain the algorithms they used to put the markup back into the translated content and there seems to be confusion about phrase segmentation. As earlier papers have done, they run the Moses decoder with the `-t` option to report segmentation together with the translation. They claim that the "additional information stored in the output shows the alignment of source and target tokens" (Tezcan and Vandeghinste, 2011, 58), but that is not really accurate. In a case as simple as the one they use as an example in the paper: `this |3-3| is |2-2| a |0-0| house |1-1|` you could be forgiven for thinking that the numbers are source and target tokens, but they actually report how the translation is made up of phrases from the phrase table. Consider a slightly different example, translated by a different system: `es handelt sich hier um eine |0-2| test |3-3|` where it is obvious that the numbers are not aligned tokens. I cannot imagine that their approach worked well if they thought of phrase segmentation as word alignment and programmed their tool accordingly.

Regarding the results they obtained, they show that the third method, role-based markup normalization, works best in all cases, across all metrics that were measured (BLEU, NIST, METEOR). The differences between the methods are rather small in many cases, so that it is not always justified to say that the third method performed better than everything else. In particular, full tokenization performs as good as all the other methods, which I do not find surprising, given that segments with more tokens are more likely to achieve high scores and given that full tokenization greatly increases the number of tokens. SMT systems are known to be able to cope with a considerable amount of noise and still generalize well, but I am still looking forward to testing for myself the performance of full tokenization compared to other, more sophisticated methods.

word alignments along with the translation (personal communication by email). The newest version that uses word alignments is here: *https://github.com/achimr/m4loc/blob/master/xliff/reinsert_wordalign.pm*.

3.1.5 Joanis et al. (2013)

At the time of writing, Joanis et al. (2013) is the most serious and authoritative treatise of this subject. They clarify the terminology that should be used to describe approaches to markup handling, which was a little fuzzy until then: they divide all approaches into "one-stream" and "two-stream" approaches. Two-stream approaches keep a copy of the original input text during translation and pass the original along the first "stream", while in the other stream, the markup is removed and this cleaned string is given to the decoder for translation. One-stream encompasses all kinds of approaches that do not remove the markup from the text completely. Put another way, two-stream approaches remove and reinsert, while one-stream approaches replace or mask information.

What they themselves propose is a two-stream approach; markup is stripped prior to training and translation, and reinserted after translation. They motivate the need for a solution to the tag handling problem with examples that are spot-on:

> In the two-stream approach, the decoder may initially translate "**Hang up** the phone!" into German as "Lege das Telefon auf!" Depending on how the tag reinsertion rules are written, the final translation might be "**Lege** das Telefon **auf**!", "Lege das Telefon auf!", or even "**Lege das Telefon auf!**" In the one-stream approach, we can easily tell the decoder not to break up a contiguously-tagged word sequence. The decoder would probably produce "**Lege auf** das Telefon!", which exactly reproduces the formatting of the original, but has unidiomatic word order. (Joanis et al., 2013, 74)

The example above clearly illustrates the problems that arise in a two-stream approach that must account for any amount of reordering in the output sentence. Their approach to tag insertion roughly works as follows:

- identify a sequence of tokens in the source sentence that is enclosed by a matching pair of XML tags (source tag region, STR),
- find the phrases in the target sentence that correspond to all of those tokens in the source sentence (target covering phrases, TCP),
- find all phrases in the source sentence that correspond to the target covering phrases (source phrase region, SPR),
- follow a set of rules that are designed to do the right thing in most cases. If there is doubt, they rather include more tokens than necessary between a matching pair of tags. In clear cases, for instance if the STR and SPR are identical and all tokens in the TCP are adjacent to each other, markup is guaranteed to be inserted perfectly.

For a complete overview of all rules, the interested reader is referred to the paper itself. Their implementation works with both TMX and XLIFF input documents and they make use of both phrase segmentation and word alignment to reinsert markup into the translations. In some cases, using phrase segmentation is sufficient to make an informed decision and faster than processing the word alignment in all cases, which might be one of the reasons they use both.

They performed a "mini evaluation" of their approach, manually annotating roughly 1500 segments. The results showed that "most tags are placed correctly" (ibid., 79), because 93 % of TMX tags and 90 % of XLIFF tags were perfect according to the human annotators. In a similar evaluation by segments instead of tags, around 80 % of segments were either perfect or only had spacing errors in them, some of which might be irrelevant (either because whitespace is insignificant in some places in XML documents or because whitespace does not matter in the specific use case).

The authors themselves identify an important limitation of their work, namely that they "do not

carry out an experimental comparison between the one-stream and two-stream approaches, though this would certainly be a worthwile next step" (ibid., 74). Such an evaluation would indeed be advisable, and the goal of the current thesis is exactly that: providing reimplementations of different approaches and comparing them to each other in controlled experiments.

3.2 Community Efforts

Apart from published scientific papers, a few communities and individuals have made an effort to solve the problem of handling markup in machine translation. In some cases, there is no documentation or description available, but the source code of those tools is available for free and I have inferred some of the information presented in this section simply by looking at the code.

First of all, the work described in Hudík and Ruopp (2011) is actually part of an open-source project, *Moses for Localization* (M4loc; Ruopp 2010). The project was initiated and led jointly by Moravia, Digital Silk Road and the LetsMT! Consortium[18]. The aim of the project was to lower the barriers for the localization industry to integrate Moses systems into their workflows, which, among other things, meant to find a viable solution to the markup handling problem. All of the code is open-source[19] and has a free license for both academic and commercial purposes. In its entirety, M4loc provides tools to preprocess TMX or XLIFF, to train Moses systems and offers a REST API which can be queried to request a translation. The tools rely on modules from the Okapi framework[20], a more extensive library of tools for localization and translation, but in many places, the only input format allowed is *InlineText*, an XML format specific to the Okapi framework, which is a limiting factor.

A few years later, another solution was put forward in the context of the *Matecat* project (Federico et al., 2014). Matecat is a versatile CAT tool that has a web frontend and a Python server as the backend that wraps the functionality available in Moses. Christian Buck and Nicola Bertoldi have developed a solution for markup handling to be used in Matecat and they called it *tags4moses*. The code is available online[21] although, unfortunately, it was never used in production[22]. The solution in tags4moses borrows ideas from published papers described in Section 3.1, but some aspects are novel. What happens to input strings is, in this order:

- the segment is tokenized by a markup-aware tokenizer which keeps together opening and closing tags of elements,
- then, the segment itself is stripped of markup, but the information on the markup is stowed away in an unescaped XML element that will be handed to the decoder and ignored,
- the decoder will translate the text without any markup (and, by implication, the models were trained without any markup in the training data) and all markup will be passed through the decoder untouched,
- the markup restoration step will word-align the source segment and the translation and the markup will be reinserted into the translation. All elements are numbered to avoid confusion if there are multiple tags with the same name in a segment.

[18] Find more information about those companies and organizations here: *http://www.moravia.com*, *http://dsr.nii.ac.jp/about.html.en* and *https://www.letsmt.eu*, respectively.

[19] Find it online here: *https://github.com/achimr/m4loc/*.

[20] See *http://okapiframework.org/* and *http://okapiframework.org/wiki/index.php?title=Main_Page*.

[21] See *https://github.com/christianbuck/matecat_util/tree/master/code/tags4moses*.

[22] In fact, Christian Buck told me I am the first person in a long while to inquire about the code (personal communication by email).

Now that word alignment is available as an option in the Moses decoder, it is no longer necessary to word-align segments again during markup handling, which makes parts of the tags4moses code obsolete, but everything else is still relevant and the method works well. For instance, it already implements a resilient HTML parser that is capable of processing malformed markup and handle it in a meaningful way.

Another tool that apparently includes code to handle markup is a Perl script developed by Hervé Saint-Amand as a contribution to Moses[23]. It is a CGI script that translates web pages and is aware of HTML markup. There is no description of the actual tag handling method that is implemented, but looking at the code, it seems that markup is replaced with placeholders, and the original markup is reinserted after translation. Or, rather, markup is reinserted while the original HTML page is reconstructed. Identifying the parts of a web page that should be translated is not exactly a trivial task and this script is an interesting alternative to translating pages with Google Translate, because the translation engine can be customized.

A further project I would like to mention is the Wikimedia `ContentTranslation` tool used to translate wiki articles[24]. Apertium (Forcada et al., 2011) is used as the MT backend, and the developers correctly point out that not handling HTML markup at all leads to erroneous output. Their solution is to remove all markup from the source segment and reinsert it later with the help of "subsentence alignment information", which can be interpreted either as segmentation or word alignment information. As far as I know, this project is the only example of markup handling used together with a rule-based system. In fact, it is even the only project that does not use libraries from Moses to train the systems.

Very recently, the problem of markup handling has surfaced again as part of the ModernMT project[25] that investigates, among other things, how the scalability of MT systems can be improved and how systems can adapt to a target domain in a flexible way, without the need to train separate systems for different domains. The code is available for free on Github[26], and it includes a so-called "tag injection API", developed by Nicola Bertoldi[27]. The only description of "tag injection" is in the first technology assessment report of the ModernMT project (Germann et al., 2016). According to this report (ibid., 23ff):

- the tag manager removes markup and significant whitespace from the input segments, but retains this information,
- it receives the translation from the decoder and invokes a separate, external word aligner to align the cleaned input segment and the translation,
- based on symmetrized word alignments, the tool reinserts markup and spaces into the translation.

Reinserting markup in this way is certainly a good approach, and using a separate word alignment step makes it more independent of the underlying MT system, because systems other than Moses might not be able to report word alignment. At the time the report was written (May 2016), the tag manager had an overall error rate of about 25 %, including errors that only concern whitespace characters. This is an encouraging result and the methods implemented in ModernMT might turn out to be the most successful and cutting-edge solution to the problem at hand.

[23]See *https://github.com/moses-smt/mosesdecoder/blob/master/contrib/web/translate.cgi* for the code and *http://www.statmt.org/moses/?n=Moses.WebTranslation* for a description.

[24]See *http://thottingal.in/blog/2015/02/09/translating-html-content-using-a-plain-text-supporting-machine-translation-engine/*.

[25]See *http://www.modernmt.eu/*.

[26]See *https://github.com/ModernMT/MMT*.

[27]I was informed about this by Achim Ruopp (personal communication by email), but Nicola Bertoldi could not be reached by email for an explanation of what methods he actually implemented in the ModernMT project.

3.3 Treatment of Markup in Commercial Products

Several commercial providers claim that their tools handle markup in the translation process. Since machine translation is often used together with CAT tools, it makes sense to also discuss those CAT tools and look at whether they can interface correctly with a translation system that has markup handling.

Since the tools are closed-source, it is difficult to obtain conclusive information, let alone find out what method each tool uses exactly. Therefore, I have focused on answering two simple questions regarding the handling of markup in those tools:

- When the tool interfaces with an external MT engine, does it send to the translation server only the text contained in the segment or a whole segment, together with its inline elements? If only the text is sent to the MT system, it cannot be expected to handle markup in a meaningful way.
- Is pre-translation with TM entries aware of the markup? For instance, if the text of a new, untranslated segment is identical to the text in a TM segment, but the inline elements are different, does this lower the match rate of this pair? Can the user configure what happens if markup is involved?

I will answer those questions by simply listing common translation tools together with anything I could find out about their way of handling markup:

- **Across Language Server**: The tool can be connected to a machine translation API in several ways, and can be configured to send whole segments, including the inline elements. This is true at least for different versions of XLIFF, I am not sure about working with other formats. Pretranslation with a TM is aware of the markup and can intelligently adapt the markup of a new segment if its text is identical to a TM segment.[28]
- **Lilt**: In the case of Lilt, it is not really accurate to describe MT as an external component because the tool is best described as an interactive machine translation tool rather than a CAT tool with MT plugins. It prides itself on intelligent tag placement in order not to burden translators with having to manipulate the markup. Markup is removed from the source segment before machine translation, and reinserted later, a method I have discussed already several times in this thesis. Using Lilt terminology, the tags are "projected" onto the translation as a postprocessing step. Lilt is also aware of markup in TM entries.[29]
- **Trados SDL**: Probably the most widely used CAT tool at the time of writing, it connects to MT servers in a flexible way, sending the whole segment including inline elements if there is evidence that the remote system can handle markup, sending only the text otherwise. Trados is aware of XML content during pretranslation if XML formats are involved.[30]
- **MemoQ**: Unfortunately, I could not find any conclusive information about markup handling in MemoQ and nobody at Kilgray Translation Technologies was available to comment on the capabilities of the tool.[31]

[28] According to Christian Weih, Across Systems GmbH (personal communication by email), also see *http://www.my-across.net/en/*.

[29] According to Saša Hasan, Research Scientist at Lilt Inc. (personal communication by email), also see *https://lilt.com*.

[30] According to Petra Dutz, Senior Sales Representative at SDL (personal communication by email), also see *http://www.translationzone.com*.

[31] See *https://www.memoq.com/en/*.

4 Implementation of Strategies for Markup Translation

As the survey of the literature in Section 3 shows, there are a variety of different approaches, each with its own strengths and weaknesses. Because of stark differences in the experiments and evaluations of those methods, it is impossible to compare them in a meaningful way. On top of that, some of the methods are not explained very well or the actual code of the implementation is not available.

In order to enable better comparisons of those methods, I have reimplemented or modified several approaches mentioned in the literature. Those reimplementations will serve as the basis for the experiments I report on in Section 5.2, but first I will explain them in depth in the current section. All of the programming code is available for free upon request and is part of a larger framework called **mtrain**[32] that offers convenient tools for training and using MT systems. `mtrain` is written purely in Python and there are plans to open-source it in the near future.

I will first elaborate on `mtrain` in general and give a broad overview in Section 4.1. Then I give a detailed explanation of how tokenization can be modified to be aware of markup (in Section 4.2). Moving on to the core methods of markup handling I have implemented, Section 4.3 explains the masking strategies and Section 4.4 details the reinsertion strategies available in `mtrain`. All sections include clear code examples that the readers are encouraged to try for themselves, after obtaining and installing `mtrain`.

4.1 Integration of the Thesis Work into a Larger MT Framework

In general, `mtrain` is a tool for fast deployment of machine translation systems. Its design principles are:

- flexible and modular style, so that components can be used individually or repurposed easily,
- standard MT tools are *wrapped* whenever possible instead of modified (e.g. the tokenizer included in Moses), because modifications create unnecessary and hard-to-find dependencies,
- be a *convenience* wrapper: if the user does not configure, make educated guesses based on the data, do something useful and keep the user interface uncluttered,
- get as much information as possible from directories that contain trained engines instead of burdening the user with options.

All of those principles also apply to the markup handling code I have written. In the course of enabling markup processing in `mtrain`, I have worked on the following:

- changes to tokenization process,
- ways to have the decoder process report alignment and segmentation,
- force translations with decoder markup,
- methods to replace markup with placeholders and reverse this process,
- methods to remove markup before translation and reinsert it later,
- many other changes that were a necessity, e.g. extensive automatic evaluation or reading from error streams without deadlocks.

[32] To a large extent, `mtrain` was conceived and programmed by **Samuel Läubli** and I would like to make sure he is given credit for it. I have only contributed exactly the parts I describe in this section.

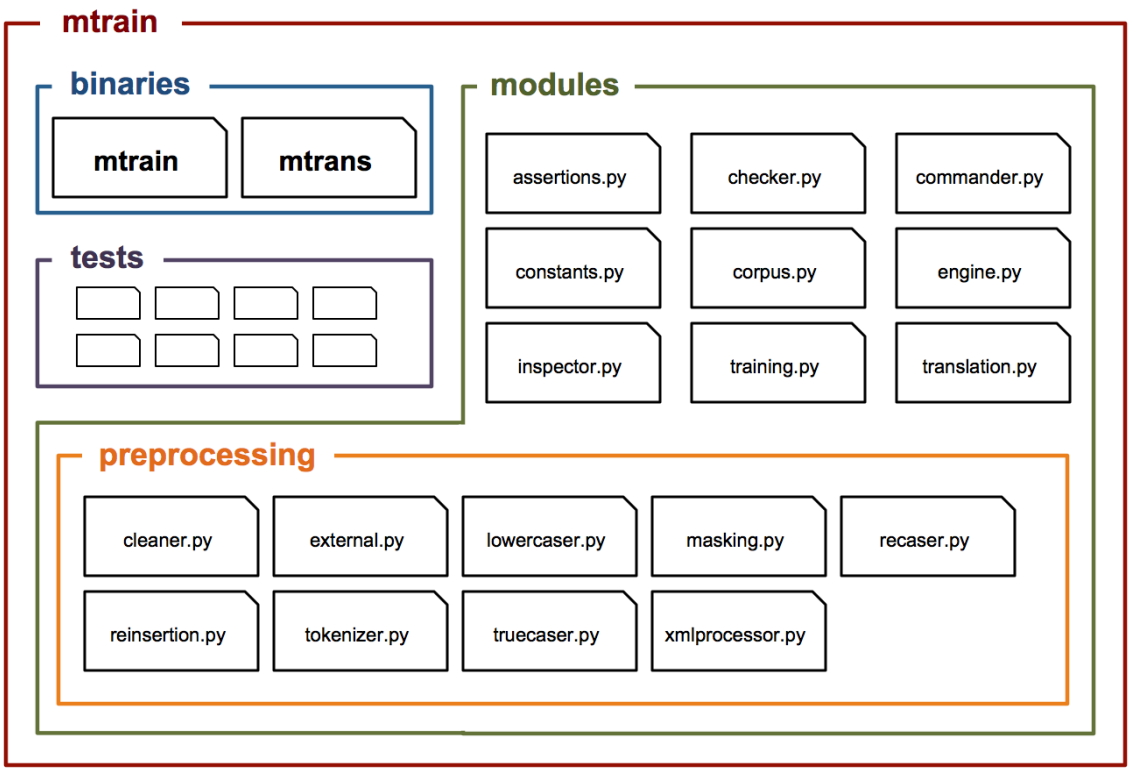

Figure 4.1: Modules of the `mtrain` framework as of December 2016

Figure 4.1 shows an overview of all components currently implemented in `mtrain`[33]. There are two "binary" modules that, once `mtrain` is installed properly[34], have user interfaces to coordinate training or translation. `mtrain` includes a suite of regression tests that can be run automatically to make sure that the code is working exactly as intended. The remainder of `mtrain` is divided into modules that each have a distinct purpose. This thesis is not the place to discuss all of the modules at length, but it includes code to preprocess raw parallel text, train statistical models with the Moses toolkit, handle casing, tokenization, execute and monitor external processes, tune a system, evaluate, interact with a Moses decoder for translation – and the latest addition, process markup.

Most of the work I describe in later sections can be found in `xmlprocessor.py`, `masking.py` and `reinsertion.py`, but changes in many other modules were also necessary. In the following sections, I will always show what modules need to be imported for each code example, so that interested readers can rerun examples for themselves.

The partitioning of the code already hints at the actual methods I have implemented, namely both strategies to replace markup with placeholders before training and translation (which I will henceforth call **masking**) and others that remove markup and reinsert it later (which I will call **reinsertion**). Figure 4.2 shows an overview of all the strategies I have implemented. In total, there are five different ways in which markup can be processed in the training and translation phases, most of them based on existing methods: the only novel approach is *identity masking*, the four other strategies are reimplementations

[33] Very recently, too late to redesign the figure with the module overview, I have added another module called `evaluator.py` which allows more extensive and faster evaluation.

[34] The code for `mtrain` can be cloned with git and then installed with a Python 3 version of pip, a package manager for Python. See Appendix A for installation requirements.

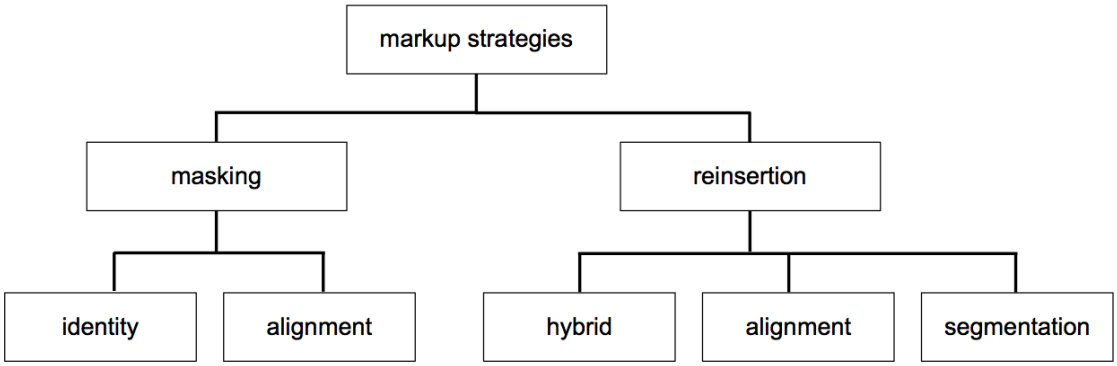

Figure 4.2: Overview of implemented strategies to process markup

of existing solutions. The remainder of this section explains all strategies in detail. The two different versions of masking are explained in Section 4.3 and the three reinsertion methods are explained in Section 4.4.

Technically speaking, `mtrain` has yet another strategy for markup handling found in the literature: treating it as normal text. Du et al. (2010) and Tezcan and Vandeghinste (2011) have tested this method, where it was called "complete tokenization" and "full tokenization", respectively. However, it would perhaps be an overstatement to describe it as proper markup handling, since the preprocessing and translation will completely obliterate the markup.

The five proper strategies (and an additional, improper one) I have implemented do not cover all the methods and their variations I have described in Section 3. Apart from the fact that more strategies would probably overwhelm the users, it was a conscious decision not to write code for them. I will briefly list methods that are not provided for in `mtrain`, together with an explanation for why this is so:

- **replace markup with zones (Du et al., 2010)**: as I have explained in Section 3.1.1, there are theoretical objections to this method and in practice, it did not perform well. Zones complicate or prevent reordering and longer, more coherent phrases from the phrase table are potentially discarded because of the interfering zones. Also, zones are a very specific feature of the Moses decoder: there is no guarantee that other SMT frameworks offer the same functionality and therefore, the method lacks generalizability.
- **replace markup with an ID and keep a global index (Zhechev and van Genabith, 2010)**: some aspects of the method are unclear, e.g. how attributes are dealt with. If attributes and their values are discriminative, that is, if different attributes lead to a different ID, then essentially, one needs to build an index of all actual markup *strings*, which does not scale well. The method is also very similar to the two variants of masking that I have implemented, except that they work in a sentence-wise fashion.
- **role-based markup normalization (Tezcan and Vandeghinste, 2011)**: this is also a variant of replacing markup with placeholders, i.e. masking. The name of an XML element is used as a placeholder, which is somewhere between using the same placeholder for all occurrences of markup (alignment masking, see Section 4.3.2) and using unique IDs on a sentence level (identity masking, see Section 4.3.1), so that I have omitted this intermediate version. Also, the number of different IDs would depend very much on the input data as it corresponds to the number of distinct element names.
- **word alignment as a separate step after translation (tags4moses, ModernMT)**: the alignments reported by Moses are accurate enough in general, and there is no reason to assume that an external

word aligner has better performance. In tags4moses, word alignment is an external script because, at that time, Moses was not capable of reporting word alignments. As far as I can see, word alignment is also computed separately in ModernMT, and I would be interested to know why this is so.

- **passing the markup through the decoder (tags4moses)**: finally, I did not write code that allows wrapping markup in unescaped XML and instructing the decoder to pass through XML. This approach has the potential advantage that the process that receives the translation from the decoder and does the postprocessing does not need to be informed by the process that did the preprocessing, about the markup that was removed from the source segment. But in the case of `mtrain`, keeping the original markup in memory until the decoder returns the translated segment is simple, and does not have a large memory footprint either, because the processing is done sentence by sentence.

original	`Hello <g id="1" ctype="x-bold;">World!</g>`
after markup removal	`Hello World!`
after tokenization	`Hello World !`

original	`Hello <g id="1" ctype="x-bold;">World!</g>`
after masking	`Hello __xml__ World! __xml__`
after tokenization	`Hello _ _ xml _ _ World ! _ _ xml _ _`

original	`Hello <g id="1" ctype="x-bold;">World!</g>`
after masking	`Hello @@xml@@ World! @@xml@@`
after tokenization	`Hello @ @ xml @ @ World ! @ @ xml @ @`

Table 4.1: Interaction between standard tokenization and markup handling strategies. Top: markup is removed and tokenization works correctly in this case. Middle: masking removes markup before tokenization, but the tokenizer then tokenizes the mask tokens instead. Bottom: varying the appearance of the mask tokens does not change the result.

4.2 Tokenization Strategies

As mentioned several times already, standard tokenization gets in the way of markup processing[35] (see Section 2.2.3). In general, there are three solutions to this problem, namely: customize the tokenizer, change the pipeline so that markup is already taken care of when tokenization takes place or find a way to instruct the standard tokenizer to be aware of markup. Modifying the code of the standard tokenizer is out of the question because it is one of the stated design goals of `mtrain` to wrap tools instead of changing them and to avoid unnecessary dependencies (see Section 4.1).

Secondly, moving the tokenization step past markup processing only produces correct results if the markup handling strategy is a variant of reinsertion. In that case, all markup is removed from the string and the tokenizer does not need to see the markup. But if the strategy is masking, then the markup processing will introduce mask tokens into the strings and we are essentially facing the same problem: tokenization will not tear apart the markup, but it will tear apart the mask tokens almost certainly, considering the fact that mask tokens should be unlikely strings (see Table 4.1 for an illustration). Thus, the only real solution to the problem is to instruct the standard tokenizer to not tokenize markup, which is what I have done.

Fortunately, the standard Moses tokenizer already offers a means to influence the tokenization. It accepts a parameter `-protected`, which lets the user point to a file with regular expressions, one per line. Anything that matches such a "protected pattern" will be left untouched by the tokenizer. `mtrain` includes functions that write those patterns to an engine directory and an external tokenizer process can be launched in such a way that it will respect the patterns. Listing 4.1 shows sample code that imports the `masking` module and writes a single protected pattern to a file, namely an expression that matches XML. Parsing XML with regular expressions is not recommended in general, but it is the only way to instruct the tokenizer not to split up markup and translation segments should only contain a subset of XML node types. For instance, CAT tools that prepare XLIFF documents abstract most of the markup

[35] Some of the readers might be aware that the standard Moses tokenizer actually has an option `-x` to "skip XML", but this option only applies when there is an outermost XML element that encompasses everything else, and then the whole string is returned untokenized, which is not helpful.

```
>>> from mtrain.preprocessing import masking
>>> masking.write_masking_patterns('test.dat', markup_only=True)

>>> from mtrain.preprocessing.tokenizer import Tokenizer
>>> standard_tokenizer = Tokenizer("en")
>>> markup_aware_tokenizer = Tokenizer("en", protect=True,
    protected_patterns_path="test.dat", escape=False)

>>> standard_tokenizer.tokenize(
    'Hello <g id="1" ctype="x-bold;">World!</g>', split=False
    )
'Hello < g id = " 1 " ctype = " x @-@ bold ; " > World ! < / g >'

>>> markup_aware_tokenizer.tokenize(
    'Hello <g id="1" ctype="x-bold;">World!</g>', split=False
    )
'Hello <g id="1" ctype="x-bold;"> World ! </g>'
```

Listing 4.1: Sample code showing the differences between standard and markup-aware tokenization. Awareness of markup is achieved by means of a file with protected patterns. For better readability, tokenized strings are not escaped.

```
>>> from mtrain.preprocessing.reinsertion import tokenize_keep_markup
>>> tokenize_keep_markup('Hello <g id="1" ctype="x-bold;"> World ! </g>')
['Hello', '<g ctype="x-bold;" id="1">', 'World', '!', '</g>']
```

Listing 4.2: Splitting a tokenized string into its tokens, but keep together opening and closing XML elements. Implemented in `mtrain` as a fast and efficient SAX parser.

out of the translation unit and it is highly unlikely that processing instructions or CDATA sections are found in such a segment.

Returning to what happens in Listing 4.1, after writing the patterns file, the sample code imports the `Tokenizer` class and creates two instances of the class, one of them interacts with a standard Moses tokenizer, the other one with a tokenizer that respects the pattern that protects markup. The rest of the lines show the different outputs, depending on whether the tokenizer is aware of markup or not. The second type of tokenizer is invoked if it is followed by markup masking.

A further difficulty that manifests itself in the context of tokenization is that the tokenizer returns a string and it is implied that single spaces separate tokens from each other. This is not the case for XML markup, since there is whitespace e.g. between the opening tag of an element and its first attribute. Simply splitting the tokenized string by whitespace will not split the string into tokens correctly. Therefore, in case markup processing is enabled, a string is split into its tokens in a different way, using a customized SAX parser.

A SAX parser is a lightweight XML parser that processes markup as a stream of events. As it encounters XML content, it reports each and every piece as an event that a programmer can choose to react to. In many languages, this behaviour is implemented with callbacks that are invoked once a certain type of event is detected. I have repurposed such a parser by defining its callback functions in a way that gradually builds a list of tokens. So, it is a method to split a string into its tokens while it keeps together opening and closing XML element tags. Listing 4.2 is a short code example that shows how to invoke this function in `mtrain`.

This concludes the discussion of markup-aware tokenization. The methods explained above ensure

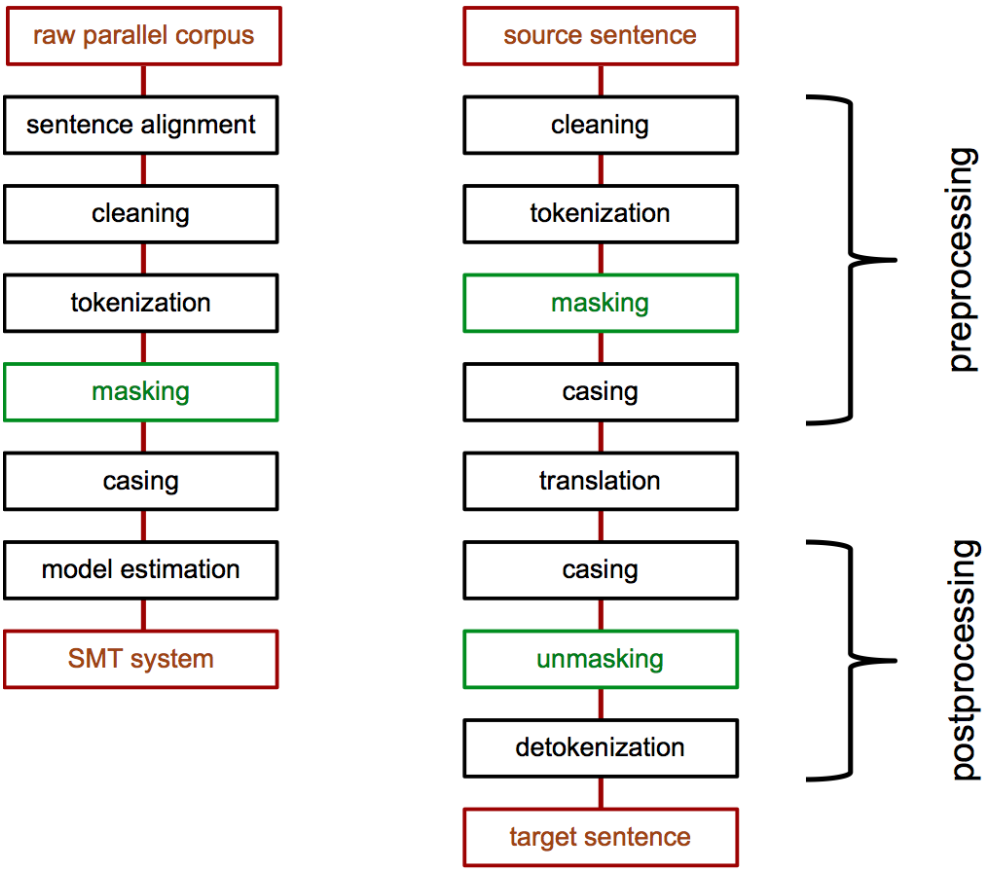

Figure 4.3: Pipeline of common preprocessing steps for training (left) and pre- and postprocessing steps for translation (right), adapted for a scenario that includes masking (highlighted in green).

that tokenization treats markup properly so that all subsequent processing steps can still identify it. One of those subsequent steps can be masking, which is what I will introduce in the next section.

4.3 Masking Strategies

Masking means that certain substrings of the source segment are "hidden" from the training script or decoder in the sense that they are replaced with a placeholder, or mask token. The method is actually not specific to markup processing: all kinds of strings can be masked, it is just that masking markup is a common application of masking. Other uses for masking include numbers, email addresses or URLs. The masking mechanisms in `mtrain` are flexible enough to allow the masking of any substring, given that it can be described with a regular expression. In the remainder of this section, I will discuss mainly the masking of markup, but readers should keep in mind that the method generalizes very well.

Apart from the two variants of masking I will describe shortly, `mtrain` also allows forcing the decoder to translate mask tokens. Once mask tokens are inserted into the source segment, they can be turned into constraint decoding directives that suggest to the decoder that the translation of a mask token is, of course, the unchanged mask token. This functionality enables testing whether enforcing the translation of mask tokens has an influence on translation quality in general or on the recoverability of masked markup (see Section 5.2.3). If a mask token is lost during translation, the original content that the mask

```
>>> from mtrain.preprocessing.masking import Masker
>>> masker = Masker('identity')
>>> masker.force_mask_translation("Hello __xml_1__ World ! __xml_2__")
'Hello <mask translation="__xml_1__">__xml_1__</mask> World ! <mask
translation="__xml_2__">__xml_2__</mask>'
```

Listing 4.3: Wrapping mask tokens in unescaped XML that forces their translation onto the decoder (or lets the value of the `translation` attribute compete with entries from the phrase table, depending on how the decoder was invoked).

corresponds to cannot be reinserted. Listing 4.3 shows how forcing mask translation can be achieved in `mtrain`. A `Masker` object must be created first because the function `force_mask_translation` depends on the exact masking strategy, "identity" in this case. The purpose of the sample code is to show that unescaped markup can be used to influence the decoder at translation time.

4.3.1 Identity Masking

The first masking strategy I have implemented is **identity masking**. The method is novel because it is not described in any publication and, to my knowledge, there is no project with open-source code that uses it[36]. However, I cannot imagine that I am the first one to consider this simple modification of alignment masking.

I have called it identity masking because it retains the identity of all mask tokens in a sentence, by assigning unique IDs to all mask tokens. The mask tokens are only unique within each sentence, but given that phrase-based systems currently translate only one sentence at a time, being unique on the sentence level is enough. For document-level decoding or a different machine translation paradigm alltogether, the merit of this method would have to be re-evaluated.

Identity masking is simple: continuous stretches of markup are replaced with mask tokens of the form `__xml_ID__`[37], where ID starts at 0 and then increments if there is more markup in the sentence. It is important to note that the masking step does not parse the XML markup, since most segments are not well-formed documents, but XML fragments (see Section 4.4.1). Masking does not know about pairs of opening and closing element tags, tags are treated as separate entities (entities in their general sense, not XML entitites), as nothing could be gained from knowing about the relationship between opening and closing tags.

For the training phase, replacing the markup with mask tokens is enough and the original content does not need to be kept. But for translation, the original markup must be kept in memory so that the placeholders can be replaced with the original again after translation. Thus, in addition to the masked string, masking in `mtrain` always returns a mapping between the mask tokens and the original content (see Listing 4.4 for a code example).

An `Masker` object in `mtrain` is able to return the mapping because it uses a custom match object that can track changes made to strings. For an arbitrary regular expression, it returns the first matching substring (as a regular match object would do) that is then substituted with the mask token, but also

[36] There is one single online resource that mentions identity masking, a collection of short descriptions of markup handling methods by TAUS, see *https://www.taus.net/knowledgebase/index.php?title=Handling_formatting_information_in_a_parallel_corpus_for_statistical_machine_translation*.

[37] Representing placeholders with underscores is not my own idea, but inspired by the masking step in scripts written by Rico Sennrich and Mark Fishel that were used in the Finnova MT project.

```
>>> from mtrain.preprocessing.masking import Masker
>>> masker = Masker('identity')
>>> masker.mask_segment('Hello <g id="1" ctype="x-bold;"> World ! </g>')
('Hello __xml_0__ World ! __xml_1__', [('__xml_0__', '<g id="1"
ctype="x-bold;">'), ('__xml_1__', '</g>')])
```

Listing 4.4: Code example of applying identity masking to an input segment that contains markup. The masking function call returns both a string where mask tokens are inserted and a mapping between the mask tokens and the original content.

```
>>> from mtrain.preprocessing.masking import Masker
>>> masker = Masker('identity')
>>> source_segment = 'Message moi a an@ribute.com ou <a>
http://www.statmt.org </a>'
>>> masked_segment, mapping = masker.mask_segment(source_segment)
>>> masked_segment, mapping
('Message moi a __email_0__ ou __xml_0__ __url_0__ __xml_1__',
[('__email_0__', 'an@ribute.com'), ('__xml_0__', '<a>'),
('__xml_1__','</a>'), ('__url_0__', 'http://www.statmt.org')])
# after translation ...
>>> target_segment = 'Email me at __email_0__ or __xml_0__ __url_0__
__xml_1__'
>>> masker._unmask_segment_identity(target_segment, mapping)
'Email me at an@ribute.com or <a> http://www.statmt.org </a>'
```

Listing 4.5: Identity unmasking based on the mapping that is returned by the masking step and the translated target segment. This example also shows that masking is not restricted to markup processing, arbitrary substrings can be replaced.

appends the original substring to a list of tuples that can be retrieved by the calling process as soon as all replacements are done.

After translation, the mask tokens in the target segment need to be removed and the original content must be inserted back into the string. In the case of identity masking, the aforementioned mapping and the target segment are sufficient to perform the unmasking step. Listing 4.5 shows identity unmasking based on the translated segment and the mapping. In this toy example, the translation step in-between masking and unmasking is left out, but is crucial, of course.

Given that a certain mask token is present both in the translation and as an entry in the mapping, identity masking is unambiguous and the original markup can always be restored under those conditions. If a mask token is omitted in the translation (which is possible if the translation of mask tokens is not enforced) then the original markup cannot be inserted in the right place, but `mtrain` chooses to insert the original markup that corresponds to this mask at the end of the segment by default.

On the other hand, the translation might contain mask tokens that are not in the mapping, because the decoder can insert phrases that have mask tokens in them. In that case, the users can decide themselves to remove all superfluous mask tokens from the translation, because there is no original content they could be replaced with. Another possible method to handle uncertain cases is leaving those mask tokens intact and not reinserting unplaceable markup. When a `Masker` object is created in `mtrain`, this behaviour can be configured with a keyword parameter.

```
>>> from mtrain.preprocessing.masking import Masker
>>> masker = Masker('alignment')
>>> masker.mask_segment('Hello <g id="1" ctype="x-bold;"> World ! </g>')
('Hello __xml__ World ! __xml__', [('__xml__', '<g id="1" ctype="x-bold;">'),
('__xml__', '</g>')])

masker.mask_segment('Message moi a an@ribute.com ou <a> http://www.statmt.org
</a>')
('Message moi a __email__ ou __xml__ __url__ __xml__', [('__email__',
'an@ribute.com'), ('__url__', 'http://www.statmt.org'), ('__xml__', '<a>'),
('__xml__', '</a>')])
```

Listing 4.6: Alignment masking, a masking strategy that replaces all XML element tags with the same mask token (top). However, different masking patterns still lead to different mask tokens (bottom).

4.3.2 Alignment Masking

As a variation of identity masking, I have implemented a masking method that uses fewer distinct mask tokens and reintroduces markup based on the word alignment reported by the decoder. I have chosen a middle way between identity masking (see Section 4.3.1) and an extremely generic mask token that reduces all matching patterns to `__mask__`, because the mask tokens do not have an ID, but the mask tokens are based on the name of the masking patterns. For instance, a masking pattern for email addresses could be `[\w\-\_\.]+\@([\w\-\_]+\.)+[a-zA-Z]{2,}`, and it could be named `email`. Substrings that match the pattern would then be replaced with `__email__`. Alignment masking is similar to role-based markup normalization (Tezcan and Vandeghinste, 2011), but it introduces fewer mask tokens, because mask tokens are grouped by match pattern, not by XML element name.

Listing 4.6 shows a simple case of alignment masking, where all substrings that match the same pattern are replaced with the same mask token. Therefore, there are two mask tokens of the form `__xml__` in this example. Alignment masking reduces the number of distinct mask tokens in the data, namely, doing away with all the mask tokens that in identity masking (see Section 4.3.1) had IDs above 0. This may have a positive effect on the quality of machine translation because for each individual mask token, there is more evidence in the data. But on the other hand, alignment masking complicates the unmasking step.

Unmasking is more complicated because there is no unambiguous correspondence between mask tokens in a translated segment and the mapping anymore (as there was in identity masking). If a segment contains several occurrences of the same mask token, it is unclear which entry in the mapping it belongs to. That is why alignment unmasking can only be achieved with additional information from the decoder. The Moses decoder naturally lends itself to two possible ways of supplying this additional information: alignment and segmentation. As I have argued in Section 3.1.3 already in the context of segmentation reinsertion, segmentation reported by the decoder is less valuable for markup handling than alignment. One would generally not resort to segmentation to help handle markup if alignment information is available. Thus, I have not implemented segmentation unmasking, but alignment unmasking.

Its reliance upon alignment means that alignment unmasking might fail because of bad or incomplete alignment, but if the alignment between mask tokens in the source and target segment are perfect and if all mask tokens are actually translated, then successful unmasking is still guaranteed. One such perfect case is shown in Listing 4.7, where it is evident that alignment is relevant to the method.

```
>>> from mtrain.preprocessing.masking import Masker
>>> masker = Masker('alignment')
>>> masked_segment = 'Message moi a __email__ ou __xml__ __url__ __xml__'
>>> translated_segment = 'Email me at __email__ or __xml__ __url__ __xml__'
>>> mapping = [('__email__', 'an@ribute.com'), ('__url__',
'http://www.statmt.org'), ('__xml__', '<a>'), ('__xml__', '</a>')]
>>> alignment = {0:[0], 1:[1], 2:[2], 3:[3], 4:[4], 5:[5], 6:[6], 7:[7]}
>>> masker.unmask_segment(masked_segment, translated_segment, mapping,
alignment)
'Email me at an@ribute.com or <a> http://www.statmt.org </a>'
```

Listing 4.7: A case of successful alignment unmasking. The unmasking step crucially depends on alignment information reported by the decoder. Unmasking succeeds in this case because all mask tokens are present in the translation and because the alignment is perfect.

```
if there are no identical mask tokens in the mapping:
  perform identity unmasking
else:
  split source segment into tokens
  split target segment into tokens
  for each source token in the masked source segment:
    if the source token is a mask:
      if the source token has an alignment to a target token:
        if the target token is also a mask:
          replace the target token with the leftmost occurrence of this mask
          in the mapping
          continue, do not look for further alignments of the source token
  if there are still mask tokens in the target segment:
    remove all of them if requested
  if there are still entries in the mapping:
    append markup from them at the end of the target segment if requested
```

Listing 4.8: Pseudo Python code for alignment unmasking. If there actually are no identical mask tokens in the mapping, alignment unmasking can fall back to identity unmasking, which is more robust, because it does not rely on word alignment.

I would like to discuss briefly how alignment unmasking works internally. Roughly, this is how I have implemented it: both the source and target segment are split into tokens. Then the algorithm (see Listing 4.8 for more elaborate pseudo code) loops through all of the source tokens. If a source token is a mask, and if it is aligned to a target token that is also a mask, then replace this target token with the original content from the leftmost occurrence of this mask token in the mapping. Order is preserved in the mapping because it is a list of tuples, so the leftmost entry will always be the right one if the source tokens are processed from left to right. Remaining mask tokens in the target segment can be removed if this is requested by the user, and likewise, unused entries in the mapping can be inserted at the end of the target segment. Mask tokens can remain in the target segment because the decoder introduced additional mask tokens or because the alignments are misleading. Mapping entries can be unused because of bad word alignment, too, or because a mask token from the source segment was not translated at all.

At this point, the most important facts for readers to remember are that I have implemented two variants of masking, identity masking and alignment masking. Identity masking assigns mask tokens with unique IDs, while alignment masking relies on word alignment for unmasking.

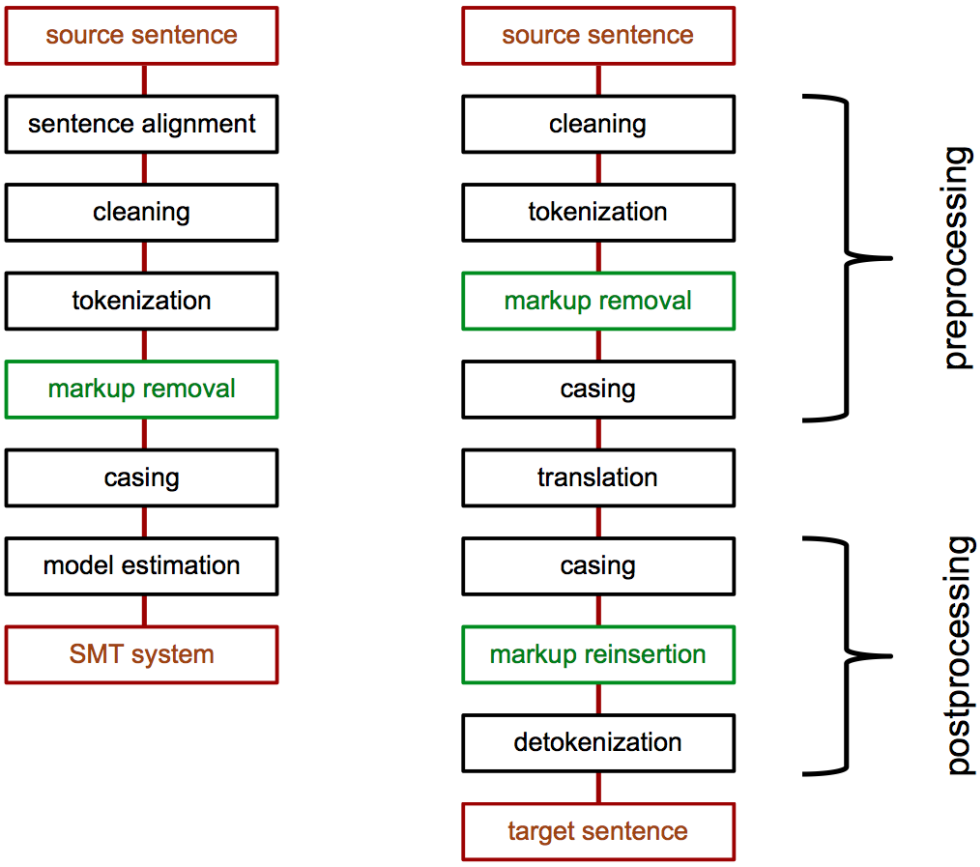

Figure 4.4: Pipeline of common preprocessing steps for training (left) and pre- and postprocessing steps for translation (right), adapted for a scenario that includes markup reinsertion (highlighted in green).

4.4 Reinsertion Strategies

The collection of strategies I present in this section are fundamentally different from masking. Reinsertion methods do not replace markup with mask tokens, but remove markup entirely from the training data and from each source segment that needs to be translated. After translation, the markup is reinserted into the target segment, hence the name **reinsertion**. Figure 4.4 shows how a standard pipeline of processing steps needs to be modified to include markup reinsertion in training and translation. Removing the markup before training implies that the models do not contain any markup at all and that reinsertion can be used with any trained SMT system, regardless of whether markup is in the training data or not.

Further, unlike masking, reinsertion, as I have implemented it, is specific to markup. `mtrain` does not allow the stripping and reinsertion of something other than XML markup. Reinsertion could in principle be more general, but it appears to me that there would not be any application for it in the context of phrase-based SMT. After all, a reinsertion mechanism is used because of a specific shortcoming of the decoder, namely its inability to process markup (see Section 2.2.3) and the fact that XML is a strict language that must be transferred faithfully, while everything else can be translated directly.

All approaches to reinsertion remove markup from segments before training and translation, so an efficient method of removing XML from strings is a requirement, and I will discuss this method in

```
>>> from mtrain.preprocessing import xmlprocessor
>>> processor = xmlprocessor.XmlProcessor('strip')
>>> processor._strip_markup('Hello <g id="1" ctype="x-bold;"> World ! </g>')
'Hello World !'

>>> processor._strip_markup('Neue Testaspekte 6192 und <bpt id="1">&lt;rpr
id=&quot;6&quot;&gt;</bpt>6193<ept id="1">&lt;/rpr id=&quot;6&quot;
transform=&quot;close&quot;&gt;</ept>', keep_escaped_markup=False)
'Neue Testaspekte 6192 und 6193'
```

Listing 4.9: Top: removing markup from segments by wrapping them in an outermost XML, parsing and returning only the text nodes of this XML document. Bottom: recover more of the original string by also removing escaped markup (actual example from my data set).

Section 4.4.1. I have put into practice three different reinsertion strategies: reinsertion based on segmentation (see Section 4.4.2), reinsertion based on alignment (see Section 4.4.3) and a hybrid method that uses both segmentation and alignment (see Section 4.4.4).

4.4.1 Removing Markup

For reinsertion strategies, removing the markup is an integral part of the preprocessing. The text inside XML elements should be kept, but the element tags themselves should be removed. My approach is to wrap each segment in an outermost (or "root") element, then parse it with an XML parser and return all the text nodes found in each XML-wrapped segment.

Parsing the segment with an XML reader is a good idea because once a segment is parsed into memory, it is represented as a tree of nodes (for instance, a DOM or XDM tree) and an XML application can easily tell apart characters that are element names or attributes (and should be removed) and others that are the text content of elements (and should be kept). Any other method of identifying the relevant substrings in a string of XML would be more brittle and complicated.

It is necessary to wrap segments in an outermost XML element because XML parsers will reject content if it is not inside a single element that encompasses everything else. As soon as a structure has a single outermost element it is said to be an *XML document* and it is the only kind of document that XML parsers can read. If there is no outermost element, but a string adheres to all other rules of well-formedness (among other things, all elements are closed, all ampersands are escaped, no attributes appear twice on the same element) then it is said to be a well-formed *XML fragment*.

Translatable segments that have inline elements and that were extracted to a line-based format (that Moses can read) typically are XML fragments. In XLIFF, for example, segments reside in `<trans-unit>` elements and must be XML fragments, because XLIFF as a whole is an XML standard and documents must be well-formed.

Returning to how markup removal is implemented in `mtrain`, Listing 4.9 shows how a string can be rid of its markup with an `XmlProcessor` object. Internally, an lxml[38] XML document is constructed from the string, and, as explained above, all the text nodes are extracted from the lxml tree. Regarding the performance of this method, the relevant parts of lxml are implemented in C for speed, and parsing such small-sized documents is reasonably fast. An even faster solution would use SAX parsing, but the difference would perhaps matter only for real-time systems.

[38] A very common XML library for Python, see *http://lxml.de/*.

```
>>> from mtrain.preprocessing.reinsertion import Reinserter
>>> reinserter = Reinserter('segmentation')
>>> source_segment = 'Hello <g id="1" ctype="x-bold;"> World ! </g>'
# markup removal, then translation...
>>> translated_segment = 'Hallo Welt !'
>>> segmentation = {(0,1):(0,1), (2,2):(2,2)}
>>> reinserter._reinsert_markup_segmentation(source_segment,
translated_segment, segmentation)
'<g ctype="x-bold;" id="1"> Hallo Welt ! </g>'
```

Listing 4.10: Reinsertion based on the original source segment that contains the markup, the translation without the markup and segmentation reported by the decoder. Reinsertion fails to reinsert the first element tag in the right place because markup can only be inserted at the boundaries of phrases.

An implication of removing markup in this way is that input segments must be XML fragments. I have decided that it is the responsibility of the user to provide well-formed fragments and that `mtrain` is within its rights to reject segments that are not. If the input is malformed XML, then no method that is aware of XML markup can be expected to work properly and could only produce garbage, because garbage came in. Segments that are not well-formed are excluded from training, in the same way that segments that are too short or too long are excluded. `mtrain` informs the user about the number of malformed segments that were discarded.

Until now, I have discussed how unescaped, directly accessible markup is removed. But segments can also contain escaped markup that an XML parser will interpret as text content and, by extension, will not remove. Since the purpose of markup removal is to recover the original string in the source or target language without markup, it could be argued that the escaped markup could also be removed. The bottom half of Listing 4.9 shows how this can be done in `mtrain`, although I have not used this functionality in my experiments. Besides, it is again the user who is responsible for removing escaped markup when they extract segments.[39]

4.4.2 Segmentation Reinsertion

My implementation of segmentation reinsertion is based on the work of Hudík and Ruopp (2011), discussed in Section 3.1.3. As I have explained earlier, relying on segmentation may lead to worse results than relying on word alignment, but I have still implemented segmentation reinsertion for two reasons. First of all, to put this hypothesis to the test and see whether segmentation reinsertion leads to worse results. To my knowledge, there is no empirical evidence that would corroborate it. Secondly, the reinsertion algorithm is explained very clearly in the paper and there is exemplary pseudo code to follow. Turning the latter into actual code was straightforward, compared to other publications where the explanation of the algorithm left a lot to be desired.

Listing 4.10 shows an example of segmentation reinsertion. A `Reinserter` object is created and the strategy to be used is provided as a keyword argument. The original source segment that contains the markup needs to be retained, and the target segment without markup and segmentation are returned by the decoder. Based on those three inputs, markup is reinserted into the target segment. The example makes it clear that markup cannot always be inserted correctly because of the segmentation. If the

[39] Apparently, I am not alone with having this opinion. For instance, Eric Joanis told me they have treated escaped markup in the same way. That is, as an error on the side of the user that the software is not responsible for (personal communication by email).

```
1  split the source segment into tokens, keeping together element tags
2  split the target segment into tokens
3  for each entry in the segmentation:
4    determine the source and target tokens that are involved
5  determine the indexes of all opening elements in the source segment
6  determine the indexes of all closing elements in the source segment
7  for each target phrase:
8    for each index of the corresponding source phrase:
9      if elements need to be opened at this index:
10       collect those opening elements
11     if elements need to be closed at this index:
12       collect those closing elements
13   output collected opening elements
14   output target phrase
15   output collected closing elements
16 if there are remaining opening elements in the source segment:
17   output them if requested
18 if there are remaining closing elements in the source segment:
19   output them if requested
```

Listing 4.11: Pseudo algorithm of segmentation reinsertion. Lines 13 to 15 show how the actual output is produced and hint at the fact that reinserting markup within target phrases is not possible.

segmentation contains any spans longer than a single token, this means that markup cannot be inserted in the middle of this phrase, only before or after it. To be very precise, in the example in Listing 4.10, the reinsertion is not perfect because the first element tag `<g id="1" ctype="x-bold;">` cannot be inserted between `Hallo` and `Welt` because the two form a phrase `(0,1)`. This is a known limitation of the method and it cannot be overcome if word alignment is not available.

The algorithm I have used for segmentation reinsertion is shown in Listing 4.11, mimicking the code described in (Hudík and Ruopp, 2011, 52) as closely as possible. It determines the indexes of all element tags in the source segment, separately for opening and closing elements so that the well-formedness of the result can be guaranteed. Then it loops through all target phrases and checks if there are any opening elements that should be inserted before a particular target phrase, and if there are closing elements that should be inserted after the target phrase. Finally, if requested by the user, element tags in the source segment that have not been inserted into the target segment can be inserted anyway. If so, again care is taken to first insert opening elements.

Summing up, segmentation reinsertion is imprecise if markup should be inserted within phrases and it is expected that it will be outperfomed by alignment reinsertion, which is the topic of the next section.

```
>>> from mtrain.preprocessing.reinsertion import Reinserter
>>> reinserter = Reinserter('alignment')
>>> source_segment = 'Hello <g id="1" ctype="x-bold;"> World ! </g>'
# markup removal, then translation...
>>> translated_segment = 'Hallo Welt !'
>>> alignment = {0:[0], 1:[1], 2:[2]}
>>> reinserter._reinsert_markup_alignment(source_segment, translated_segment,
alignment)
'Hallo <g ctype="x-bold;" id="1"> Welt ! </g>'
```

Listing 4.12: Reinsertion based on the original source segment that contains markup, the translated segment and, most importantly, the alignment between the source segment without markup and the translation.

4.4.3 Alignment Reinsertion

If word alignment is available, either reported by the decoder or computed separately by a word alignment tool, then markup can be reinserted using this alignment. Word alignments pinpoint more precisely the location where markup should be reintroduced into the target segment, so compared to segmentation reinsertion, an improvement can be expected. To illustrate this, Listing 4.12 shows an example of alignment reinsertion in `mtrain`, using exactly the same input string as the example for segmentation reinsertion in Listing 4.10. Whereas segmentation reinsertion could not place the markup correctly in the target segment, alignment reinsertion can perfectly reinsert markup in this case.

The algorithm itself is a minor variation of the segmentation reinsertion algorithm[40]. shown in Listing 4.11, so that I will not reproduce it here. The main difference is that it loops through the target *tokens* instead of the target phrases and determines the elements that should be inserted separately for each token. Each unit of markup from the source segment (that is, an element tag) is thought of as being attached to adjacent tokens. For example, in the source segment shown in Listing 4.12, `<g id="1" ctype="x-bold;">` attaches the token on its left, `Hello`. After `Hallo` is written to the output, the algorithm decides that `<g id="1" ctype="x-bold;">` should now be written, since `Hallo` is aligned to `Hello` and `<g id="1" ctype="x-bold;">` is next to `Hello`.

Obviously, the success of alignment reinsertion depends on the quality of word alignments. Word alignment will never be perfect and will deteriorate as the sentence length grows. If word alignments are missing, the method will either ignore the markup or resort to inserting all of the remaining markup at the end of the target segment, depending on the user's preferences.

4.4.4 Hybrid Reinsertion

The third reinsertion strategy is a reimplementation of Joanis et al. (2013, see Section 3.1.5), a more elaborate approach that uses both word alignment and phrase segmentation, that is why I have called it "hybrid". Although it is a reimplementation and I have made an effort to clarify the details of the algorithm with the main author, I do not claim that my interpretation of hybrid insertion is identical to the one described in this paper.

[40] Also, it is a minor variation of the work done by Hudík and Ruopp (2011), who have later adapted their code to work with word alignment as well, see *https://github.com/achimr/m4loc/blob/master/xliff/reinsert_wordalign.pm*.

```
>>> from mtrain.preprocessing.reinsertion import Reinserter
>>> reinserter = Reinserter('hybrid')
>>> source_segment = 'Hello <g id="1" ctype="x-bold;"> World ! </g>'
# markup removal, then translation...
>>> translated_segment = 'Hallo Welt !'
>>> alignment = {0:[0], 1:[1], 2:[2]}
>>> segmentation = {(0,1):(0,1), (2,2):(2,2)}
>>> reinserter._reinsert_markup_full(source_segment, translated_segment,
segmentation, alignment)
'Hallo <g ctype="x-bold;" id="1"> Welt ! </g>'
```

Listing 4.13: Hybrid reinsertion given perfect segmentation and alignment. In this case, the following rule applies: the SPR extends beyond the STR and the tag pair is inserted so that it surrounds all target tokens that are aligned to the STR.

The rules that govern the process of hybrid reinsertion are substantially more complex than the rules for segmentation and alignment reinsertion. They require the formal introduction of a number of concepts: source tag regions, target covering phrases and source phrase regions – all of which I have described in Section 3.1.5, but I will still briefly repeat their meaning:

- **source tag region (STR)**: a sequence of source tokens that is enclosed in a pair of matching opening and closing tags,
- **target covering phrases (TCP)**: all target phrases that a specific source tag region has contributed to,
- **source phrase region (SPR)**: the tokens of all source phrases that correspond to a set of target covering phrases.

Notice that this is the only method so far that understands the notion of a tag *pair*, and this opens up new possibilities. Hybrid reinsertion can model tokens that are inside a particular pair of tags, and trace all of those tokens to their position in the target segment. Therefore, markup can be reinserted in a way that all target tokens that are aligned to source tokens inside a particular tag pair will be inside this particular tag pair in the target segment as well. The actual rules that guide reinsertion are (based on the descriptions in Joanis et al., 2013), for each pair of tags:

- If the STR and SPR are identical and all phrases of the TCP are adjacent to each other, then insert the tag pair so that it surrounds the TCP.
- If the SPR is more extensive than the STR, while the TCP phrases are still adjacent, then insert the tag pair in a way that it surrounds all target tokens that are aligned to any token of the STR.
- If the TCP phrases are not next to each other, insert the tag pair in a way that it surrounds all phrases of the TCP, including unrelated phrases in-between the TCP phrases.

In addition to those rules for tag pairs, selfclosing tags are seen as being attached to the next source token and are placed before the target token this source token is aligned to. So, basically, alignment reinsertion is used for selfclosing tags.

As an example, consider the simple code snippet in Listing 4.13 where the second rule for paired tags applies: the SPR (source token indexes 0, 1 and 2) extends beyond the STR (source token indexes 1 and 2). In that case, the tag pair is inserted so that it surrounds all target tokens that are aligned to a token in the STR (target token indexes 1 and 2) – and the TCP is ignored in this case.

The advantage of hybrid insertion is clearly that it keeps together pairs of element tags, which is unique to this method. On the other hand, the added complexity might lead to more intricate errors. For instance, different STRs can have TCPs that overlap and "there is no good rule to fix it automatically" (Joanis et al., 2013, 78). The complexity of the code might also have an impact on the performance.

I have now described all of the five markup handling strategies I have implemented: identity masking, alignment masking, segmentation reinsertion, alignment reinsertion and hybrid reinsertion. I hope it has become clear that I have implemented a range of markup handling methods representative of all the published work in the field and that what I have provided will be a sound basis for the experiments I will report on in the next section.

5 Experimental Setup

All experiments I have conducted are comparisons between different methods of handling markup in machine translation. Whenever the input to a translation system contains markup, its developer needs to decide on a certain method to deal with the markup, but, as I have argued earlier (see Section 3.1.5, for instance), this is difficult. Even with all the information currently available, it is difficult to make an informed decision because useful empirical comparisons between methods are rare.

In a number of cases, a description of a method or even its source code was published, but without presenting empirical evidence about its performance. Other publications include small evaluations of the method, but rarely is it a comparison of different methods. And finally, the comparisons that do exist are simply too dated to be useful today, because the methods compared are too crude from today's point of view and because the SMT frameworks have improved considerably since then.[41]

Therefore, I think it is instructive to set up experiments that facilitate the comparison between various methods of markup handling. For instance, a juxtaposition of reinsertion and masking strategies is necessary and to the best of my knowledge, a reinsertion method has never been compared to a masking method in earnest. But I will also explore more nuanced questions, for example whether forcing the translation of the mask tokens is useful, which also has never been addressed in the literature. One of the reasons that the answers to such questions are currently lacking is that nobody has ever implemented markup handling methods sufficiently different from each other in the same framework. This is the gap this thesis work is trying to fill, by providing implementations for five different markup handling strategies, each with additional customizable options to allow for meaningful comparisons in controlled experiments.

The data set I have used in my experiments is a collection of real-world XLIFF documents that contain markup and it will be described next, in Section 5.1. Then, Section 5.2 explains in detail the experiments I have conducted – which should enable readers to reproduce exactly the results I have obtained.

5.1 Data Set

For all experiments, I have used a corpus of parallel XLIFF documents in German and English as the data set[42]. Figure 5.1 shows basic descriptive statistics for this data set[43], among other things the number of segments and the distribution of markup in the data. The total number of segments is not impressive, but should be sufficient to train a phrase-based SMT system. Given that the overall goal is not to train the best-performing system but to compare markup handling methods, the number of segments is acceptable.

The number of tokens is counted in two ways: a naive count that splits segments at whitespaces, and a markup-aware count that counts each element tag only once even if it has whitespace in it. If the counting is markup-aware, there are considerably more tokens in the data because text content is often followed directly by markup or vice versa without whitespace between them. Taking an actual example from the data, the naive count will treat `Version<ph>` as a single token, while the markup-aware count will separate the two into `Version` and `<ph>`.

About a quarter of all segments have at least one markup tag in them, which means that markup

[41] For a justification of the claims I am making here and tangible examples, see Section 3.

[42] Thanks to Roberto Nespeca and finnova Bankware AG for the generous permission to use their data in my experiments.

[43] The script I have written to compute those statistics can be used for any line-based corpus in general, and it is available in the code directory submitted with this thesis: `util/corpus-stats.py`.

	English data set	German data set
Number of non-empty segments in total	427180	425650
Number of tokens in total (naive)	2684111	2335924
Number of tokens (markup-aware)	3419637	3038984
Segments that have at least 1 tag	98212	97041
Average number of tags per segment	1.26	1.24
Number of malformed segments	103	98
Average number of tokens per segment (naive)	6	5
Average number of tokens per segment (markup-aware)	8	7
Average number of characters per segment	53	53

Table 5.1: Basic statistics of the data set used in the experiments. Segments that only contain whitespace are considered empty, that is why the counts for English and German are not equal. "Tag" should be interpreted simply as an XML tag, not as an XML element with both an opening and closing tag.

is abundant in the data, more than enough to test markup handling. In each data set, there is a small number of segments (around 100) that are not well-formed XML fragments (see the discussion of XML fragments in Section 4.4.1) where naive counting was used as a fallback. For the experiments, this means that about 100 segments need to be excluded at the outset because malformed content cannot be processed in a markup-aware fashion.

The data set contains markup because it was extracted from XLIFF 1.2 documents with inline markup.[44] The documents are so-called "introductory checklists" used for parameterization of banking software, similar to software manuals, so the texts are from a very technical domain. It is technical in the sense that the segments frequently include strings of programming code, numbers, placeholders, controlled vocabulary and – of course – formatting. Table 5.2 shows a random, unbiased sample of segments from the data set. Statistical translation systems are known to perform better if the training data is from a technical domain and this will likely hold true in my experiments.

5.2 Experiments

I have conducted experiments in a controlled environment, only varying the factors that are the subject of investigation. Section 5.2.1 will elaborate on those controlled conditions and give clear instructions to reproduce the translation systems I have built. The main experiment is a comparison between all methods I have implemented and it will be described in Section 5.2.2. The second and third experiment are not as comprehensive and focus on a certain aspect of markup handling. Section 5.2.3 introduces an experiment to gauge the impact of forcing the translation of mask tokens, while Section 5.2.4 examines whether masking and reinsertion methods should aggressively insert markup or remove mask tokens if the evidence is inconclusive. Finally, Section 5.2.5 will explain how I have measured the outcome of those experiments, both with automatic metrics and manual evaluation.

[44] The extraction code was written in the course of the Finnova MT project at the University of Zurich, mainly by Mark Fishel and Rico Sennrich, with small modifications by myself.

<bpt id="1" ctype="bold">{}</bpt>HDs with parameterisation text<ept id="1">{}</ept>
Description of parameter P_Queue in SRMUTJOUR extended
ch.finis.f2.sr.fios.<bpt id="1" ctype="bold">{}</bpt>Fios2<ept id="1">{}</ept><ph id="2"><mq:ch val=" " /></ph><ph id="3"><mq:ch val=" " /></ph> Reads files via file system gateway
FILE_EXTENSION
Mr
The PP is delivered with the following values:
<bpt id="1" ctype="bold">{}</bpt>Individual interest rate: <ept id="1">{}</ept>In the case where delivered by import, the imported interest rate is assumed in the field.
The four upper text arrays are the global texts stored in VL_STAMM.

Table 5.2: Example segments from the data set. The style of writing is very technical in most cases, punctuation is often omitted, words are frequently all-caps and segments have inline markup and placeholders.

5.2.1 Controlled Conditions of All Experiments

Let me emphasize again that the focus of my experiments is not to create the best-scoring system, but to compare the performance of markup handling methods. If perfect scores were the goal, one would certainly have to do more cleaning of the data, remove known factors of disruption (e.g. lines that consist of a single digit), train with larger general-domain corpora and then adapt to the target domain with the smaller in-domain corpus. But the goal is to build a system with reasonable performance so that it does not get in the way of testing markup handling and then keep it constant for all experiments.

The first constant in the experiments is the data set. I have divided the data described in Section 5.1 into a fixed training, tuning and testing set with `mtrain`. `mtrain` randomly assigns segments to either of those sets with reservoir sampling (Algorithm R; Vitter, 1985). The tuning set has a fixed size of 2000 segments, the testing set has a fixed size of 1000, as is customary for machine translation experiments. Segments that have malformed markup were excluded from all sets, as well as segments that only contained whitespace or were longer than 80 tokens (markup-aware count). After those exclusions, 397714 segments were left in the training set. Those sets are the point of departure for all systems and did not change throughout the experiments.

All experimental systems were trained using the `mtrain` training binary that executes scripts from Moses release 3.0 in the background. The basic parameters for training and use are:

- the translation direction is always from German to English, because the standard Moses detokenizer does not have inbuilt rules for German
- phrase-based systems, with a maximum phrase length of 7 tokens,
- a 5-gram KenLM language model with modified Kneser-Ney smoothing,
- a lexicalized reordering model (`msd-bidirectional-fe`),
- a standard Moses recasing engine will be trained,
- the standard Moses tokenizer and detokenizer are used,
- word alignment and symmetrization is done with `fast_align` and `atools`,
- the phrase and reordering tables will be compressed with the `cmph` library,

- tuning is done with MERT,
- the Moses decoder is invoked with `-xml-input constraint` for forced decoding.

This list is what all systems have in common, while the method of handling markup was varied in the experiments that I will describe next.

5.2.2 Experiment 1: Comparison of Five Different Markup Handling Strategies

The first experiment compares all methods of markup handling that I have implemented. A further system that is used for the comparison treats markup simply as text and it could be seen as the baseline. But a better baseline would perhaps be a system that completely ignores markup both in training and translation, so I also included this. Some of the strategies can be customized further in the `mtrain` settings (in `mtrain.constants`) and I have used the default configurations in each case. This means that in this experiment, seven systems with the following markup strategies were trained (all that was said in Section 5.2.1 applies):

- **naive**: a system that treats XML markup as text and fully tokenizes markup. The fragmented markup is present in training and passed to the decoder after escaping.
- **strip**: markup is stripped entirely from the training, tuning and evaluation corpus.
- **identity masking**: markup is masked with tokens that are unique within sentences. The translation of mask tokens is not forced. If there is not enough evidence, unused entries from the mapping are inserted at the end of the segment. Mask tokens in the target segment that are not in the mapping are removed.
- **alignment masking**: all occurrences of markup are masked with the same token, `__xml__`. The detailed configuration is identical to identity masking.
- **segmentation reinsertion**: markup is removed for training and before translation. After translation, it is reinserted based on phrase segmentation. Markup that cannot be placed correctly is inserted at the end of the segment.
- **alignment reinsertion**: the system is identical to segmentation reinsertion, except that word alignment is used for reinsertion.
- **hybrid reinsertion**: markup is removed in the same way as for the other reinsertion methods. It is reinserted after translation using both phrase segmentation and word alignment and a set of more sophisticated rules (compared to the other methods).

My expectations for the outcome of this experiment are that identity masking will outperform alignment masking because identity unmasking is unambiguous whereas alignment unmasking could be said to be a heuristic process. Alignment reinsertion will perform better than segmentation reinsertion because reinserting markup only at the boundaries of phrases is bound to cause placement errors. Overall, I suspect that reinsertion is more useful than masking because several modern tools (e.g. ModernMT and Lilt, see Section 3.2) use reinsertion and not masking but this might not be due to the inferior performance of masking per se, but because the translation quality may actually be better without mask tokens in the data.

5.2.3 Experiment 2: Impact of Forcing the Translation of Mask Tokens

The second experiment is specific to masking strategies and is designed to test whether forcing the translation of mask tokens is beneficial. The rationale for this experiment is that one of the situations where all unmasking strategies fail is when the mask token is absent from the translation. It is unknown how prevalent this problem is and how likely the decoder is to drop mask tokens during translation.

One could simply compare the source segments with the machine-translated segments to estimate the number of mask tokens that have disappeared, but then the presence of mask tokens in the translation is a condicio sine qua non for masking, but it does not guarantee that masking is successful in all cases. If the mask token is present in the translation and is present in the source segment, identity unmasking is guaranteed to succeed but alignment unmasking might still fail.

So, the question that is answered by this experiment is whether forcing the translation of mask tokens has an influence on the performance of masking strategies, both identity and alignment masking. To this end, two more systems are trained:

- **forced identity masking**: identical to the identity masking system described in Section 5.2.2 except that the translation of mask tokens is forced. For instance, `__xml_0__` is turned into `<mask translation="__xml_0__">__xml_0__</mask>` before translation.
- **forced alignment masking**: identical to the alignment masking system described in Section 5.2.2 except that mask tokens like `__xml__` are turned into `<mask translation="__xml__">__xml__</mask>` before translation.

Under such conditions, identity masking is expected to be perfect and alignment unmasking is expected to draw close to identity masking with regard to performance. An important caveat that I want to anticipate here is that the actual translation quality might suffer from forced decoding and automatic metrics might be too indirect a means to detect this decrease in quality. However, a manual evaluation of the overall translation quality is beyond the scope of my work and therefore, such an effect will go largely unnoticed.

5.2.4 Experiment 3: Impact of Aggressive Unmasking and Reinsertion

The third experiment addresses the question whether it is beneficial to unmask and reinsert "aggressively". Aggressive unmasking means that if there is not enough evidence to place an element tag back in the translation, it is placed at the very end of the segment, until the mapping is exhausted. Also, any mask tokens that remain in the translation are removed because it is assumed that the decoder introduced them for no good reason. Aggressive reinsertion means that unused markup from the source segment is placed at the end of the translation if there is not enough evidence, as a last resort.

Aggressive strategies might be advantageous because they prevent the loss of markup during translation, but might also insert irrelevant markup into the translation. When alignment and segmentation do not support the decision to insert (or replace a mask token with) markup in the translation, this can mean that alignment and segmentation are lacking, but also, it can mean that the markup should in fact not appear in the translation. After all, markup, like any other token, can be dropped from the translation (e.g. if all of its text content is also dropped) and in such a case it would not be accurate to insert it at the end.

The systems I have introduced in Section 5.2.2 already exhibit this aggressive behaviour: they aggressively reinsert unused markup at the end of segments, and the masking systems remove unreplaced mask tokens from the translation. I have decided to make aggressive behaviour the default behaviour because it seems more important to preserve markup than to reinsert it in the right place, and because unreplaced mask tokens are not useful in the translation, but it is always better to validate assumptions with an experiment.

So, to test whether aggressive reinsertion and removal of superfluous mask tokens is useful in the first place, I have trained the following systems and compared them to existing systems from the first experiment:

- **conservative identity masking**: identical to the identity masking system described in Section 5.2.2, except that unused mapping entries are ignored and no mask tokens are removed from the final translation.
- **conservative alignment masking**: identical to conservative identity masking except that mask tokens are different and word alignment is used for unmasking.
- **conservative segmentation reinsertion**: identical to the segmentation reinsertion system described in Section 5.2.2, except that unused markup from the source segment is ignored.
- **conservative alignment reinsertion**: identical to conservative segmentation reinsertion except that reinsertion is based on alignment instead of segmentation.

Hybrid reinsertion does not provide an option to behave aggressively that can be turned off, all markup is inserted into the translation anway. Therefore, I have excluded hybrid reinsertion from this experiment.

5.2.5 Evaluation Protocol for All Experiments

I have measured the outcomes of all experiments automatically and manually. Automatic evaluation is a full-flegded module of `mtrain` that I have written to complement my thesis work. Manual evaluation was performed because there is reason to believe that automatic metrics do not accurately reflect improvements or degradations in markup handling.

The automatic evaluation was done with Multeval (Clark et al., 2011) which is integrated in `mtrain`, currently in the `mtrain.evaluator` module. MultEval reports BLEU, METEOR, TER and length scores and is in principle also capable of estimating the instability of the optimizer (MERT in this case) but I have not used this feature since it would have required to retranslate all test sets with the Moses configuration of different tuning runs. MultEval was run with the standard options, more precisely this means that BLEU is computed according to `mteval-v13.pl` from NIST, without performing tokenization, METEOR 1.3 is used[45] and TER 0.7.25[46].

The automatic evaluation can still be quite elaborate because `mtrain` supports an extended evaluation that runs several rounds of testing and scoring, varying the processing steps that are applied to the machine-translated target side of the test set and the target side of the reference. In each round of evaluation, the reference is matched to the postprocessing steps that are applied to the machine translation:

- **casing**: if the machine translation remains lowercased, the reference is also lowercased. If the translation is recased, the casing of the reference is not modified.
- **tokenization**: if the machine translation is detokenized, then the reference is tokenized. If the translation remains tokenized, then the reference is not tokenized.
- **markup**: if markup is removed from the machine translation, it is also removed from the reference and vice versa.

In the case of evaluating markup handling, not all of those evaluation scenarios are useful. While evaluating cased versus uncased translations is very important in general, casing is not expected to have any influence on markup handling and vice versa and I have therefore *only* evaluated lowercased text. Also, I have only evaluated tokenized text (which means that the machine-translated segments were not detokenized and the human reference segments were tokenized with a markup-aware tokenizer) for two important reasons that are specific to the tools I have used. Firstly, MultEval expects that both

[45] METEOR version 1.3 is outdated by now, but can still be downloaded here: *http://www.cs.cmu.edu/~alavie/METEOR/download/meteor-1.3.tgz*.

[46] Which is the most recent version of TER, available here: *http://www.cs.umd.edu/~snover/tercom/tercom-0.7.25.tgz*.

the hypotheses and references are tokenized[47] and it would be inaccurate to let it compute metrics with detokenized text.

Secondly, the standard Moses detokenizer unfortunately ignores lines that are wrapped in XML markup, meaning that the text inside such an XML element is not detokenized, but also hyphenations like `@-@` introduced by the tokenizer are not reversed. There is no option to influence this behaviour, as there is for the Moses tokenizer. In my test set, over 100 segments are wrapped in markup and therefore, it does not make sense to include an evaluation on detokenized translations. Evaluation both with and without markup in the hypothesis and reference is insightful, however. Evaluating without any markup will reveal if markup handling has any influence on the overall quality of translation.

Moving on to the manual evaluation, while the automatic evaluation translates all of the 1000 segments in the test set, the manual evaluation only looks at the segments where both the source and target reference have tags in them. In my test set, there are 176 pairs of reference segments where both sides contain at least one tag. So, all tags in 176 machine-translated segments will be inspected manually and assigned one of the following categories (inspired by Joanis et al., 2013):

- **good**: correct markup is present, correctly placed,
- **reasonable**: correct markup is present, but needs to be moved,
- **wrong**: markup is broken or not present at all,
- **garbage-in**: the decoder output is unintelligible and there is no proper place for markup.

In the manual evaluation I will focus on evaluating the markup handling, not the performance of the systems in general. This implies that I will not perform a manual evaluation without markup. For each trained system, I will evaluate all tags in all 176 segments that contain markup, which amounts to a total of 658 tags for each system, and a grand total of around 7000 tags for the whole manual evaluation. I will only be looking at the lowercased, tokenized version of the translation output, after processing the reference accordingly.

This will allow a manual evaluation by tags, and I will add an evaluation by segment. The rules for assigning categories to whole segments are:

- segments that have one or more “wrong” tags in them are “wrong”,
- of the remaining segments, those that have at least one “reasonable” tag are “reasonable”,
- of those remaining, the ones that have at least one “good” tag are “good”, because any “garbage-in” tags in those segments are not the fault of the markup handling method,
- and all remaining ones are “garbage-in”, since they only contain “garbage-in” tags.

Segments are categorized strictly in the sense that one “wrong” tag already leads to the label “wrong” for the whole segment. Likewise, if “reasonable” tags co-occur with “good” tags, the whole segment will still be “reasonable”. On the other hand, if a segment has both “garbage-in” and “good” tokens, the whole segment is “good”, because the errors in those segments are not due to the markup handling method, but due to the decoder.

Overall, the manual evaluation procedure errs on the side of caution and will not overestimate the performance of methods. I have performed this manual analysis myself, although I am aware that this introduces a certain bias. As a countermeasure, I have tried to write the evaluation guidelines as clearly as possible.

[47] See *https://github.com/jhclark/multeval* for an extended discussion of tokenization in machine translation evaluation.

	with markup				without markup			
	BLEU	**METEOR**	**TER**	**Length**	**BLEU**	**METEOR**	**TER**	**Length**
naive	84.2	59.4	12.9	98.7	33.5	38.1	82.2	145.3
strip	54.6	42.8	30.6	84.6	60.5	46.3	26.6	93.9
identity masking	72.1	52.8	19.1	98.1	**61.0**	**46.7**	**26.4**	94.3
alignment masking	71.5	52.5	19.5	98.0	60.4	46.3	26.9	94.2
segmentation reinsertion	60.3	47.2	24.2	98.6	60.5	46.4	26.8	94.9
alignment reinsertion	70.1	50.5	21.3	98.6	60.5	46.4	26.9	94.9
hybrid reinsertion	**74.1**	**54.1**	**18.2**	98.5	60.4	46.3	26.8	94.8

Table 6.1: Automatic evaluation of the overall performance of markup handling methods. System 'naive' treats markup as normal text, while system 'strip' removes markup completely from training and translation. The metrics reported are BLEU (higher is better), METEOR (higher is better) and TER (lower is better) and the best scores are set in bold.

6 Results

In general, all markup handling strategies have shown good performance, which is surprising, especially in the case of systems that use segmentation reinsertion. Considering both the automatic and manual evaluation, identity masking has outperformed all other strategies. Section 6.1 will elaborate on the findings of the main experiment. Section 6.2 will report the results of the second experiment and will make it clear that forcing the translation of mask tokens is neither necessary, nor does it improve translations. Finally, Section 6.3 will confirm that aggressive behaviour is indeed a useful default for both masking and reinsertion strategies.

6.1 Experiment 1: Comparison of Five Different Markup Handling Strategies

The purpose of the first experiment is to compare the overall performance of all five methods I have implemented, testing them with their default configurations. The markup handling methods that were tested are identity masking, alignment masking, segmentation reinsertion, alignment reinsertion and hybrid reinsertion. An analysis with automatic metrics has revealed that all methods except segmentation reinsertion were reasonably successful, but hybrid reinsertion and identity masking have outperformed all other methods. Since the latter methods were on par in the automatic evaluation, this has necessitated a manual evaluation as a further arbiter between them and it clearly shows that identity masking outperforms hybrid reinsertion in terms of correct tags and correct segments.

Table 6.1 reports the results of the automatic evaluation of the first experiment. The first two rows are systems that treat markup improperly and that could be regarded as baselines, albeit unreasonable ones. The 'naive' system does not have any markup handling at all, the main effect of which is that the tokenizer will tear apart the markup. In many cases it will not be possible to reassemble the markup after translation, but the standard detokenizer will not join a sequence of characters back to markup anyway. The 'strip' system removes markup completely, it has markup neither in the training data nor in the translations. The remaining rows are the proper methods of markup handling I have discussed at length in this thesis.

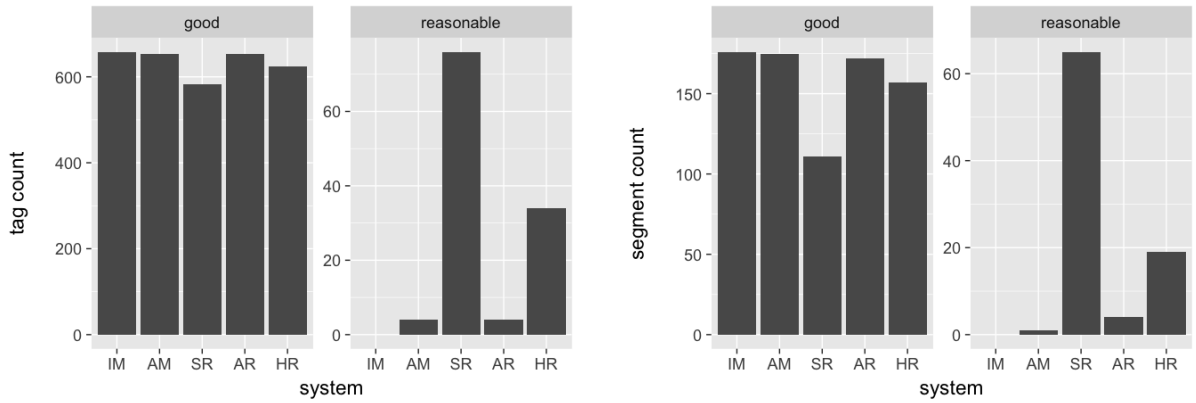

Figure 6.1: Manual evaluation of the overall comparison between markup handling methods. Methods are evaluated by tag (left) and by segment (right). Categories 'wrong' and 'garbage-in' are omitted because they only have zero counts. IM = identity masking, AM = alignment masking, SR = segmentation reinsertion, AR = alignment reinsertion, HR = hybrid reinsertion.

Bear in mind that the hypothesis and reference segments are lowercased and tokenized (or, rather, not detokenized in the case of the hypothesis segments) as I have explained in Section 5.2.5. Turning to the actual scores of the systems, the 'naive' system has by far the highest scores when the hypothesis and reference contain markup (the lefthand side of Table 6.1). This result is expected because naively tokenizing markup drastically increases the number of tokens in segments and it is well-known that automatic metrics are biased towards favoring longer segments[48]. When markup is removed from the evaluated segments (the righthand side of the table) the scores of the naive system drop considerably and all of them are now worse than those of all other methods.

As expected, the 'strip' system performed worse than all proper methods when evaluating with markup (for instance, a BLEU score of only 54), but comparable to the other methods in an evaluation without markup. The main use of this system is to have a baseline for evaluation without markup to see whether markup handling has any effect on the general quality of machine translation.

Moving on to the systems with proper markup handling, all of them except segmentation reinsertion performed well, their BLEU scores ranging roughly from 70 to 74 in the evaluation *with* markup. METEOR and TER scores also vary between methods but not by much: METEOR ranges from 47 to 54, while TER ranges from 24 to 18. When evaluating with markup, hybrid reinsertion is the best-performing method with higher BLEU and METEOR scores and lower TER score than all other methods. On the other hand, when evaluating without markup, identity masking takes the lead, although all other systems are very close and there is not much variation in general now. Again, this is expected because a method for markup handling should not have an impact on the translation of normal text. If anything, it should have a positive effect, as seems to be the case with identity masking. In any case, segmentation reinsertion clearly performed worse than all other methods (e.g. a BLEU score of 60), while identity masking and hybrid reinsertion were the most successful.

In contrast, the manual evaluation has shown that the automatic evaluation overestimates the performance of hybrid reinsertion. As is evident in Figure 6.1, identity masking still performs best: 658 out of 658 tags are 'good' (present and correctly placed in the translation) and, of course, all of the segments were correct in this case. This means that identity masking has perfect performance given the specific

[48]See e.g. Jonathan Clark's discussion of the pitfalls of tokenization and its effect on BLEU scores here: *https://github.com/jhclark/multeval/blob/master/README.md*.

	with markup				without markup			
	BLEU	**METEOR**	**TER**	**Length**	**BLEU**	**METEOR**	**TER**	**Length**
identity masking	**72.1**	**52.8**	**19.1**	98.1	**61.0**	**46.7**	**26.4**	94.3
forced identity masking	71.6	52.7	19.4	98.4	60.4	46.5	26.7	94.6
alignment masking	**71.5**	**52.5**	**19.5**	98.0	**60.4**	**46.3**	**26.9**	94.2
forced alignment masking	65.8	50.0	21.8	98.3	60.3	**46.3**	27.2	94.5

Table 6.2: Automatic evaluation of the impact of forcing the translation of mask tokens. The metrics reported are BLEU (higher is better), METEOR (higher is better) and TER (lower is better) and the best scores are set in bold.

data set I have used. Methods that rely on alignment, alignment masking and alignment reinsertion, follow very closely, both with only a handful of tags that were classified only as 'reasonable' (tag is present but needs to be moved). Both have a very high number of correct segments, alignment masking is slightly ahead with 175 'good' segments out of 176 in total, while alignment reinsertion has 172 'good' segments.

Hybrid reinsertion could only place 624 tags correctly (out of 658) and only 157 segments out of 176 are perfect with regard to markup. The worst method by far is segmentation reinsertion as it only put 582 tags in the right place, which is not a bad result per se, only given the superior performance of the other methods. In the whole manual evaluation, no tag was categorized as 'wrong' (markup is wrong or not present) and the failure of a method was never due to bad decoder output (category 'garbage-in'). Those two observations mean that the methods are robust and preserve markup very well and that the underlying machine translation system did not get in the way of markup handling.

6.2 Experiment 2: Impact of Forcing the Translation of Mask Tokens

The second experiment was geared towards investigating whether forcing the decoder to translate mask tokens would make a difference, or put another way, if not forcing the translation of mask tokens leads to the decoder omitting them in some cases.

Therefore, I have compared the two variants of masking (identity masking and alignment masking) with variants of them that force the translation of mask tokens. Table 6.2 shows the results of the automatic evaluation and its main outcome is that forcing the translation of mask tokens does not improve performance. On the contrary, it seems that it has a detrimental effect on translation quality, for instance reducing the BLEU score of a system with identity masking by 0.5 and that of a system with alignment masking by 6. However, in the case of identity masking, it is only a small difference between the two systems which implies that forcing the mask tokens did not change much.

This finding is confirmed by the manual evaluation that is summarized in Figure 6.2. While looking at the figure, readers should focus on the differences between identity masking and forced identity masking (IM and FIM) and look at alignment masking versus forced alignment masking (AM versus FAM). Any other comparison will not be instructive in this case or was already discussed in Section 6.1. The figure shows that forcing the translation of *identity* mask tokens does not change anything at all (only looking at markup handling, that is): the original identity masking system places correctly 658 out of 658 tags, and so does the forced identity masking system.

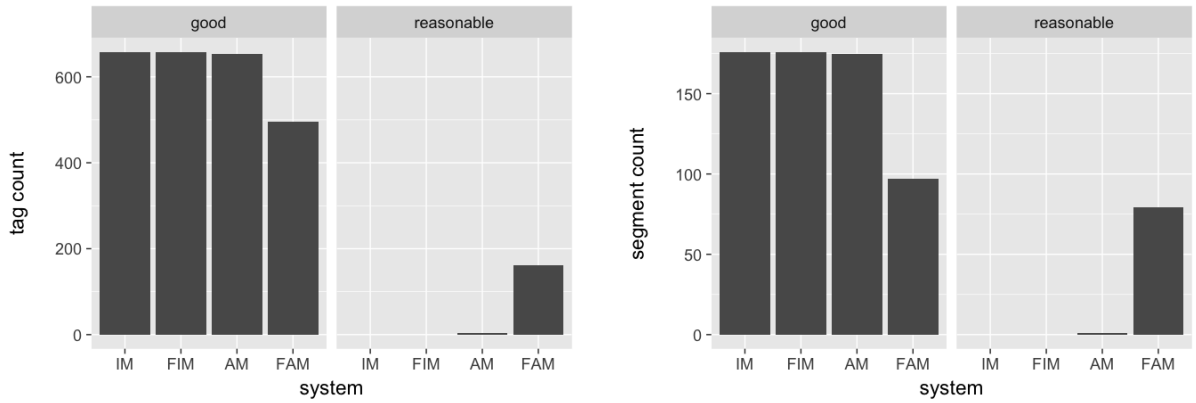

Figure 6.2: Manual evaluation of the impact of forcing the translation of mask tokens. Methods are evaluated by tag (left) and by segment (right). Categories 'wrong' and 'garbage-in' are omitted because they only have zero counts. IM = identity masking, FIM = forced identity masking, AM = alignment masking, FAM = forced alignment masking.

In the case of forced alignment masking, forcing the translation of mask tokens did not lead to an improvement, but to considerable degradation. The original alignment masking method has 654 correct tags (category 'good') and only 4 were in the wrong place (category 'reasonable'), while the forced alignment masking system only has 496 perfect tags and 162 other tags needed to be moved. Since my intention in this section is to simply report results without explaining them yet, I will postpone explanations for this fact to Section 7.1.1.

In any case, both the automatic and manual evaluation clearly show that forcing the translation of mask tokens does not lead to an improvement. By implication, this means that the decoder is not at all prone to discarding mask tokens from a translation and does not need explicit encouragment to keep the masks. In the case of alignment masking, forcing the translation of mask tokens leads to a decrease in performance, both in terms of automatic metrics and manual evaluation of markup handling.

6.3 Experiment 3: Impact of Aggressive Unmasking and Reinsertion

The third experiment investigates whether aggressive behaviour during unmasking or reinsertion is reasonable (which is very likely) and if so, how much behaving in this way contributes to the final result of handled markup.

For identity unmasking, aggressive behaviour means that mask tokens that are not in the mapping (because the decoder has introduced them on a whim) are removed from the target segment. For alignment unmasking, it additionally also means that in case of incomplete or inconclusive word alignment, markup in the mapping is inserted anyway, at the end of the target segment. Aggressive alignment reinsertion is a very similar mechanism and segmentation reinsertion does the same, but uses phrase segmentation instead of word alignment for reinsertion. Thus, aggressive behaviour only comes into play when there is *uncertainty* and therefore, the results I present in this section incidentally also answer the question of how much uncertainty there is in the process of unmasking or reinserting markup.

At this point, it is very important to reiterate the fact that systems trained with `mtrain` are aggressive *by default*, but can be configured to be *conservative* instead. The systems presented in Section 6.1 are aggressive and so are those in Section 6.2.

	with markup			without markup		
	BLEU	**METEOR**	**TER**	**BLEU**	**METEOR**	**TER**
aggressive identity masking	**72.1**	**52.8**	**19.1**	**61.0**	**46.7**	**26.4**
conservative identity masking	71.7	52.6	19.2	60.5	46.5	26.6
aggressive alignment masking	71.5	52.5	19.5	60.4	46.3	26.9
conservative alignment masking	**71.9**	**52.7**	**19.0**	**60.8**	**46.6**	**26.2**
aggressive segmentation reinsertion	60.3	47.2	24.2	**60.5**	**46.4**	**26.8**
conservative segmentation reinsertion	**72.8**	**53.4**	**18.6**	60.1	46.3	26.9
aggressive alignment reinsertion	**70.1**	**50.5**	**21.3**	**60.5**	**46.4**	**26.9**
conservative alignment reinsertion	69.6	50.1	21.9	60.4	46.3	27.0

Table 6.3: Automatic evaluation of the impact of aggressive behaviour during unmasking and reinsertion. The metrics reported are BLEU (higher is better), METEOR (higher is better) and TER (lower is better) and the best scores are set in bold. Length scores are omitted due to lack of page width.

Considering only the automatic evaluation presented in Table 6.3, the scores of aggressive identity masking are higher than for conservative identity masking, but only very slightly. This suggests that in identity masking, there is very little uncertainty involved because if there is perfect evidence, there is no difference between an aggressive and conservative method. In the same vein, the scores for aggressive and conservative alignment masking are very similar, and it is important not to read too much into differences of less than 0.5 BLEU. In the case of alignment masking, there is little difference between the two variants because the word alignment reported by the decoder seems to be stable and reliable enough for markup handling.

Yet, for segmentation reinsertion the picture is different. A conservative version of segmentation reinsertion outperforms an aggressive baseline by more than 12 BLEU points and similar improvements in METEOR and TER are observed. This means that phrase segmentation is often inaccurate or inconclusive, so that the reinsertion algorithm decides not to reinsert the markup (in the case of a conservative system) or insert it at the very end of the segment (in the case of an aggressive system). To sum up, as far as the automatic evaluation is concerned, systems do not need to be cautious given incomplete evidence, except when the markup handling method is segmentation reinsertion.

Looking at the manual evaluation in Figure 6.3, again, the only worthwhile comparisons are between neighbouring cells and when variants of the same markup handling method are compared (e.g. comparing “IM” to “CIM”). Comparing aggressive identity masking to its conservative counterpart reveals that being aggressive or not has no bearing whatsoever on the performance of the method: both systems perfectly handle 658 out of 658 tags. In fact, the aggressive or conservative nature of the identity masking systems never came into effect because there are no uncertain cases.

The same is true for alignment masking, where the number of segments in each category (’good’, ’reasonable’ and ’wrong’) is identical for the aggressive and conservative version of the system. Conservative alignment masking even has one more ’good’ tag than the original, aggressive system had – which is very likely due to inconsistent labelling by the human annotator, myself. Even though the numbers are comparable, conservative alignment masking leads to a single case of a ’wrong’ tag, be-

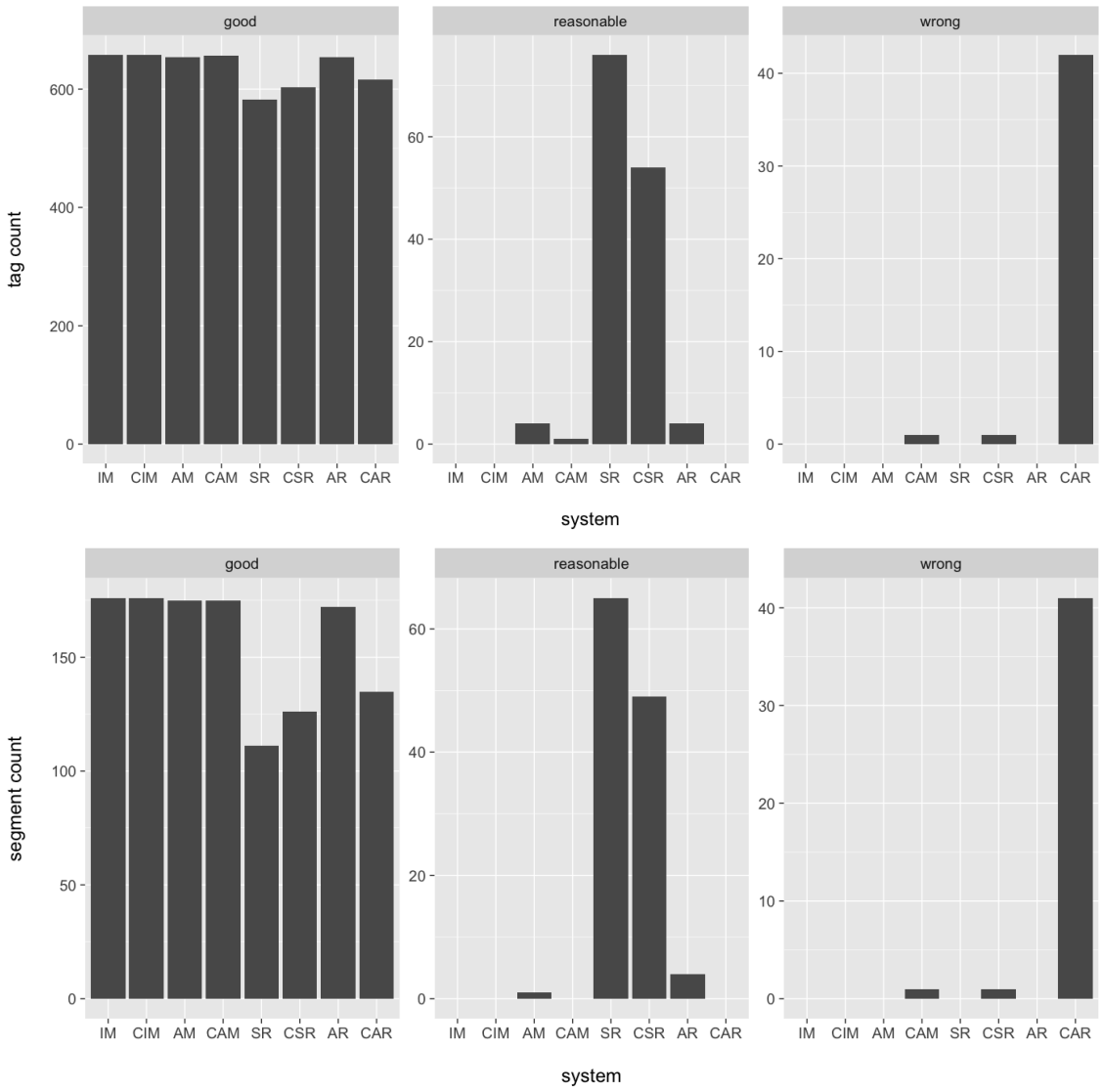

Figure 6.3: Manual evaluation of the impact of aggressive behaviour during unmasking and reinsertion. Methods are evaluated by tag (top) and by segment (bottom). Category 'garbage-in' was omitted because it only has zero counts. IM = identity masking, CIM = conservative identity masking, AM = alignment masking, CAM = conservative alignment masking, SR = segmentation reinsertion, CSR = conservative segmentation reinsertion, AR = alignment reinsertion, CAR = conservative alignment reinsertion.

cause conservative behaviour can lead to markup being "lost in translation". Contrary to the other methods, and consistent with the results of the automatic evaluation above, being conservative is actually beneficial for a system that uses segmentation reinsertion.

Finally, modifying the behaviour of alignment reinsertion so that it inserts conservatively leads to tags being omitted in a number of cases: 42 tags were omitted by conservative alignment reinsertion, but not by aggressive alignment reinsertion. This means that there is uncertainty as to where tags should be reinserted, since in 42 out of 658 cases, an aggressive alignment reinsertion system is at loss and decides to reinsert the tag at the end of the segment. Interestingly, in the majority of those cases, the end of the segment is actually the right place to insert the markup, as the results of the manual evaluation in Section 6.1 have shown.

In summary, I have shown that for masking methods, behaving aggressively or conservatively does not make a difference in most cases, but rarely, conservative alignment masking leads to tags missing from the target segment. On the other hand, reinsertion methods really are influenced by aggressive or conservative behaviour. Segmentation reinsertion actually profits from being conservative, but the overall number of 'good' tags and segments is considerably lower than with other methods because inserting at phrase boundaries is imprecise. If the markup handling method is alignment reinsertion, then being conservative is clearly detrimental.

7 Discussion

After having reported the results themselves, I will offer explanations and comment on them in Section 7.1. Although my experiments have shown that most of the markup handling strategies I have implemented are viable, they have shortcomings of course. I will discuss those thoroughly in Section 7.2, together with a criticism of the design and materials of my experiments.

7.1 Discussion of Experimental Results

First of all, it is worth repeating the fact that all of the five strategies I have experimented with coped well with the task, perhaps with the exception of segmentation reinsertion. But even segmentation reinsertion placed correctly 582 out of 658 tags, a remarkable achievement that I would not want to describe as a failure. Identity masking clearly performed best in all experiments, but I would like to emphasize that the results are not guaranteed to be valid beyond the specific data set I have used and beyond the specifics of my implementations.

Another remarkable fact is that none of the strategies had any effect on the overall translation quality – as far as automatic metrics can tell: after stripping markup from all segments, all systems reached a similar BLEU score between 60 and 61. The scores are not exactly identical because of MERT tuning, which is a non-deterministic process and minor fluctuations are to be expected. This counterbalances the oft-mentioned view that mask tokens in the data might make " the translation itself worse" (Joanis et al., 2013, 78f). But not only do my results suggest that proper markup handling does not hinder the actual translation, they also show that improper markup handling does indeed have a detrimental effect on the overall quality. The automatic scores achieved by the 'naive' system reported in Section 6.1 substantiate this claim, as they are considerably lower in the evaluation without markup (e.g. BLEU score of around 30) than those of all other systems (BLEU score of around 60).

7.1.1 Masking Strategies

In all experiments, identity masking has proven to be the most robust strategy as it solved the problem of markup handling perfectly in all cases, it was ahead of other methods in the automatic evaluation and consistently placed all tags correctly in the manual evaluation (see Section 6.1). Two subsequent experiments have shown that mask tokens are never omitted by the decoder (Section 6.2) and that there is little uncertainty involved in identity unmasking (see Section 6.3). Identity masking works well because there is an unambiguous correspondence between mask tokens and the original content they have replaced, and there is no need for auxiliary information such as word alignment or phrase segmentation. One potential drawback is that too many unique mask tokens might lead to sparse models, but in my experiments, this was not the case. At any rate, in a machine translation project, identity masking should be one of the foremost choices, even though I have come to the conclusion that the results depend heavily on the data set (see Section 7.2.2).

Alignment masking achieved similar results, almost as good as those for identity masking: with the default configuration, it is less than a handful of markup tags that were not placed correctly by alignment masking. This suggests that the word alignment reported by the decoder is sufficient for the task of unmasking. However, it should be borne in mind that the quality of word alignment depends directly on the nature of the training data and that the data set used is an uncontrolled variable. It would be interesting to investigate how alignment masking is affected by bad word alignment.

Forcing the translation of mask tokens in an alignment masking system (see results reported in Section 6.2) has led to an interesting finding that was hard to diagnose: for tokens whose translation is forced, the Moses decoder does not report word alignment points. The alignments between mask tokens are simply ignored and therefore, there will be missing values for some of the target tokens. After noticing this, I have written code to fill in the missing values before alignment unmasking and then retrained the system, simply assuming that a mask token in the target is aligned to a mask token in the source that has the nearest index. This is only an approximation of course, because forcing the translation of mask tokens does not prohibit reordering, and this will lead to errors. And, to the best of my knowledge, this is the explanation for the considerable drop in performance of alignment masking if the translation of mask tokens is forced (see Figure 6.2, in particular).

Fortunately, forcing the translation of mask tokens does not seem to be necessary because the decoder is not at all inclined to discard mask tokens from the translation. Also, neither did it introduce additional mask tokens out of the blue, for that matter (see the results for conservative vs. aggressive systems in Section 6.3).

But one drawback of masking strategies there is no getting away from is that the pre- and postprocessing steps are closely intertwined, because the mapping between mask tokens and original markup must be kept in memory until after the translation. Speaking from my own experience, this also complicates the programming of the methods. In situations where the preprocessing is completely disjoint from postprocessing, a reinsertion strategy should be considered instead, which is what I will discuss next.

7.1.2 Surprisingly Good Performance of Segmentation Reinsertion

Segmentation reinsertion cannot be said to have failed the task of reinserting markup. Quite on the contrary, it is remarkable that segmentation reinsertion could act on the markup in such a precise way, given that phrase segmentation is imprecise to begin with, as I have explained several times already. There are two obvious explanations for the surprisingly good performance of segmentation reinsertion that I would like to discuss: either the phrases in the trained models are short (even only single tokens) or markup is generally placed *between* phrases that could roughly correspond to coherent units of meaning that often occur together.

Looking at the phrase table of the segmentation reinsertion system, phrases are not shorter than usual (between 1 and 7 tokens) and there is no reason to believe that short phrase table entries caused the good results. Also, phrases in machine translation do not always correspond to units of meaning and markup is placed everywhere, not between coherent chunks of text. But a distinct feature of the data set I have used is that markup is frequently present at the very beginning and very end of segments. If there is no reordering, markup at the beginning and end of segments can always be inserted in the right place by segmentation reinsertion, regardless of the length of phrases. For markup tags within segments, segmentation reinsertion is downright paradoxical: it works better if phrases are short, while longer phrases typically lead to better translations, and by extension, segmentation reinsertion works well if the machine translation system is feeble.

In summary, this means that the success of segmentation reinsertion is due to the nature of the data set, which often had markup wrapped around the segment and little reordering. In future projects, there is probably no reason at all to implement or use segmentation reinsertion because it is consistently outperformed by other methods and because there is an inverse relationship between the success of segmentation reinsertion and the quality of machine translation.

7.1.3 Alignment and Hybrid Reinsertion

Turning to the more promising variants of reinsertion, the performance of alignment reinsertion was very similar to that of alignment unmasking. For instance, the number of correctly reinserted tags in the manual evaluation was 654 (see Figure 6.1), exactly the same number as for alignment masking, which is not surprising because both base their decisions on word alignment. On top of that, alignment reinsertion shares with alignment unmasking an important drawback, namely that it depends directly on word alignment, which in turn depends directly on the parallel data. Still, alignment reinsertion is a viable alternative to identity or alignment masking if a masking method is not feasible.

The strategy that was the subject of the most recent publication on markup handling in machine translation (Joanis et al., 2013), hybrid reinsertion, was outperformed by all other methods except segmentation reinsertion. It seems to me that this is partly due to the fact that the rules in the hybrid reinsertion algorithm are too complicated. Adding unnecessary complexity to a system invariably leads to more sources of error and the errors themselves are more difficult to diagnose. But on the other hand, speaking in defense of the method, hybrid reinsertion is clearly meant for scenarios where markup is a controlled vocabulary and known to be *formatting* and where reordering takes place frequently. Under such circumstances, difficult cases like the example shown in Section 3.1.5 arise and hybrid reinsertion is the only method that has a satisfactory solution to them. It is just that the data set I have used in my experiments did not have those cases and little reordering.

In addition, my implementation of hybrid reinsertion does not do justice to the method presented in Joanis et al. (2013) because my algorithm does not guarantee the order of nested tags that are adjacent to each other. Unfortunately, it was only after my experiments that I realized I had overlooked this detail of the algorithm: "some care is taken to preserve the source order when multiple tags end up between the same two target language words" (ibid., 78). As a consequence, I do not claim that my results are representative of the method shown in this paper and this is clearly a limitation of my software, which brings me to the next chapter, limitations of my code and experiments.

7.2 Limitations

My work has several known limitations, either tangible shortcomings of the programming code (that are described in Section 7.2.1) or inherent in the design of my experiments (see Sections 7.2.2 and 7.2.3).

7.2.1 Critical Review of the Implementations

Of all deviations from true software craftsmanship that an attentive programmer will find in my code, I want to briefly comment on the most relevant ones:

- **object-oriented programming**: `mtrain` is generally written in an object-oriented style, with modular classes and objects. But the one thing that is not implemented as a class is a *segment*. It would be useful to model a parallel segment with all its aspects as a class and instantiated objects because of the variety of signatures and return values of the methods that handle segments. Taking an example from markup handling, segmentation reinsertion needs the original source segment including the markup as an input, while alignment unmasking does not. On the other hand, alignment unmasking needs word alignment, while segmentation reinsertion does not. It would be easier to pass segment objects as a single argument to those functions. I have already begun working on this feature, but this branch of `mtrain` is experimental and not stable enough.
- **hybrid reinsertion**: as I have explained in Section 7.1.3, the order of nested markup tags that are

adjacent to each other in the source segment is not guaranteed. Guaranteeing the order requires substantial reorganization of the existing code.

- **forced alignment masking**: the unlikely case of combining a system that forces the translation of mask tokens with conservative behaviour, i.e. not inserting the markup at the end if word alignment is inconsistent, will provoke errors. They will only occur if a markup tag, or more precisely a mask token, is at the end of a segment. The decoder will not report an alignment for this mask token and the fall-back algorithm I have implemented currently does not fill in an alignment for the last token, since aggressive behaviour will cause it to be inserted at the end anyway.
- **spacing errors**: unlike other solutions for markup handling (e.g. Hudík and Ruopp, 2011), `mtrain` does not have a component for handling whitespace in the vicinity or inside markup elements. Spacing is not guaranteed at all and this might well be the most important shortcoming of my implementations. Some spaces are ignored because they are in-between markup elements and any XML library is within its rights to treat such a space as non-significant, but other space characters are lost in masking or markup-aware tokenization for no good reason. The methods do not even respect the `xml:space="preserve"` attribute of XML documents.

In future versions of `mtrain`, I hope to improve the software, most importantly by explicitly modelling whitespace and making segments objects in their own right.

7.2.2 Suitability of the Data Set

There is reason to believe that some aspects of the data set (described in Section 5.1) reduce its suitability for experiments on markup handling. I have selected this data set because it is a real-world example of parallel text that was machine-translated and post-edited by translators, that comes from a technical domain and, importantly, that naturally contains XML markup. All of those aspects make it a very typical data set for machine translation and the translation industry in general.

But for the specific purpose of testing markup handling strategies, the data set is too easy: there is little variation in the texts since the language is very controlled, segments are often short and there is little reordering. Given that the language pair is German and English, it could be expected that reordering would cause problems, but sentences are often unfinished statements where no reordering takes place at all. This finding is also reflected in the automatic scores from all evaluations that are very high: BLEU scores of 60 or more and METEOR scores of 50 and more (see e.g. Section 6.1) suggest that the test set was relatively easy to translate.

I have mentioned that there is little reordering because reordering would have helped to rank markup handling methods in a more meaningful way. In theory, reordering is a major obstacle for some strategies of markup handling, but not so much for others and those differences between the methods went largely unnoticed in my experiments because of the data set. Likewise, as I have explained in Section 7.1.2, markup tags were often present at the beginning and end of segments in a very predictable fashion, which made markup reinsertion particularly easy, especially for segmentation reinsertion.

With hindsight, it would have made sense to find a more suitable data set for my experiments, either a better suited real-word corpus or purpose-built synthetic material. I have already begun working on the latter idea and I have code in place that artificially introduces markup into markup-free parallel corpora. That way, a data set with the desired properties can be created. Among those desired properties are: more markup in general, better distribution of tags within segments without a bias towards the periphery of the segment and having longer sentences with more markup. But any experiments with artificial data should be preceded by a careful analysis of real-world corpora, so that the artificial data would not make the task of markup handling easier or harder than it really is.

7.2.3 Indirectness of Experimental Outcomes

The results of my experiments have revealed that, in regard to markup handling, there is a discrepancy between automatic metrics and manual evaluation. Put another way, automatic metrics cannot really detect with precision whether markup is handled correctly, since they are too coarse and general.

As an example, consider the evaluation of markup tags with several attributes. The order of attributes is irrelevant in XML documents, and XML libraries do not guarantee any order of attributes since changing the order of attributes does not change the semantics. If markup handling is applied to a segment, pure chance will dictate whether the order of attributes will correspond to that of the reference translation or not. But in the former case, BLEU (or any other metric that relies on n-gram statistics) will assign a higher score to the machine translation because longer n-grams are identical to the reference.

This means that an automatic evaluation as the only estimate of the usefulness of a markup handling strategy is too indirect, as a metric like BLEU is too far removed from the actual purpose of markup handling and concepts central to XML markup are unknown to the metric. To alleviate this problem, it seems very reasonable to complement a general metric with a manual evaluation of tag handling, as I have done in my experiments. Another interesting research avenue would be to develop another automatic metric that is tailored to correlate better with human judgements of the quality of markup handling strategies. This idea was articulated by others already, who argue that general metrics do not correlate well with markup placement and the post-editing effort needed (see e.g. Hudík and Ruopp, 2011, 52 and Escartín and Arcedillo, 2015, 140). But then, there already are a plethora of metrics for machine translation and such a metric could still be regarded as an outcome that is too indirect.

In fact, even a manual evaluation of markup handling is an indirect measure of the usefulness of markup handling. Since markup handling is meant mainly to assist translators in post-editing, the only direct and real outcome is the productivity of the translator (see Section 2.3.2). Therefore, a comprehensive user study, for instance similar to the experiment in Green et al. (2013), would provide the most sound evidence of the performance of markup handling methods because it directly measures the variable that is being optimized, namely translator productivity.

8 Conclusion

In this thesis, I have worked on the problem of handling markup in phrase-based, statistical machine translation. Markup handling is problematic not because there are no existing solutions at all, but because there is no generally accepted standard solution to it that could be implemented in standard tools like the Moses framework. That there is no generally accepted solution can be explained by the fact that, at the time of writing, there are no empirical comparisons between the several methods that have been put forward in the last decade.

Therefore, it has been the foremost goal of my work to provide implementations of a diverse array of markup handling strategies in a unified framework, where they can be compared to each other. I have implemented five different markup handling strategies, four of them reimplementations of existing solutions. The fifth strategy, identity masking, is a method that has no published record, but I am sure that other people have had this idea in the past. All of the programming code I have written is available for free within the machine translation framework `mtrain` and therefore, it is the first main contribution of my thesis.

On the basis of my own implementations, I have conducted three experiments to gauge the usefulness of the five markup handling strategies. I have compared all strategies to each other, investigated whether the translation of mask tokens should be forced and whether strategies should exhibit aggressive or conservative behaviour. The main findings are the following. Identity masking performed best and solved the problem of markup handling perfectly. Alignment masking and alignment reinsertion came very close to the performance of identity masking and can be regarded as viable alternatives. Hybrid reinsertion has also shown promise and its slightly inferior performance might be due to the data set and a shortcoming of my implementation of the algorithm. Finally, segmentation reinsertion should be avoided if possible – and can be avoided as soon as word alignment is available or can be computed.

Finally, a discussion of the results has revealed that masking strategies have the general drawback that the pre- and postprocessing must be performed by the same process (or, at least, the postprocessing must be informed by the preprocessing). I have explained the aspects of the data set that make it less suitable for experiments about markup handling, namely that there is little reordering and little variation and I have argued that a different data set would have emphasized better the differences between markup handling methods. The most severe limitation of my work is that the spacing in segments that have markup in them is not taken care of properly.

On a lighter note, I still believe that solving the problem of markup handling in machine translation is a worthy cause and I sincerely hope that the contributions of my thesis will be regarded as useful.

8.1 Recommendations

After working on the problem of markup handling in machine translation for several months, I feel that my findings permit putting forward *recommendations* regarding the use of markup handling strategies in future systems. Since the general applicability of my experimental results is not known, it is advisable to take everything I recommend with a grain of salt, but I am confident that it will be helpful. The recommendations are:

- If there is markup in the training data, or if it can be anticipated that markup could be present in translation, then markup must by all means be handled. Unhandled markup has a detrimental effect on the overall quality of machine translation.
- If possible, use identity masking as the markup handling strategy. In my experiments, it has proven to be the most reliable and robust method. It is also versatile in the sense that masking can be used to mask not only markup, but any set of strings that can be described with a regular expression. Also, mask tokens in the training data did not lead to a decrease in overall translation quality.
- If identity masking leads to sparse models, use alignment masking. If the postprocessing steps after translation are completely disjoint from the preprocessing, use alignment reinsertion.
- If there is ample reordering between the languages, if alignments frequently are 1 to n and if the markup is known to be inline formatting, use hybrid reinsertion.
- Do not use segmentation reinsertion at all, except when word alignment is not reported by the decoder and cannot be computed separately (an unlikely scenario).
- If the system uses masking, do not force the translation of mask tokens as it did not lead to any improvement.
- Let strategies behave aggressively in all cases, although it does not matter for some strategies (e.g. identity masking). From the point of view of post-editing, having the markup inserted at the end of the target segment is much better than markup that is missing from the segment alltogether.

9 Outlook

My own implementations and experiments leave a lot to be desired, but have also helped to unearth interesting and meaningful research opportunities. In this final section, I would like to mention ways in which my work could be continued or complemented by radically different ideas that I have not put into practice:

'Harder' parallel data

The data set I have used in my experiments may have been too linear and redundant to bring out the differences between markup handling strategies. As a consequence, several methods solved the problem almost perfectly, including strategies that rely on word alignment (alignment masking and alignment reinsertion). I hypothesize that more complex parallel data would discriminate better between the methods. The desired properties of such a data set are detailed in Section 7.2.2 and if it does not exist, it could be synthesized instead. Also, the languages that are involved and the direction of translation should be varied.

Better handling of whitespace characters

My work does not include a solution for the handling of whitespace characters within XML elements or next to them (see Section 7.2). This is a serious shortcoming because whitespace is significant in many contexts and spacing errors could slow down post-editors, in the same way improper handling of XML elements slows them down. Proper handling of whitespace would mean that the information about whitespace in the source segment is kept in memory at all times, so that machine-translated segments can be adjusted accordingly.

Better rules for known vocabularies

If markup in a data set is restricted to a certain XML vocabulary with clear semantics, better rules can be defined to handle difficult cases. As an example, consider that the meaning of a `<b>` element is known to be bold text. If content inside a `<b>` element in the source segment is aligned to two, non-contiguous tokens in the machine-translated target segment, then a rule could decide that in this situation, the `<b>` element should be duplicated. In general, the duplication of tag pairs is not warranted. Hybrid reinsertion already took a step in this direction, as it comes with the unstated, but obvious assumption that markup is inline formatting.

Markup handling metric

"Also of interest would be the development of a metric that measures the accuracy of the inline element placement and the required post-editing effort" (Hudík and Ruopp, 2011, 52). Although it was called for several years ago, nobody has designed such a metric yet, even though there would be a wide range of applications. If a new metric could be shown to correlate well with post-editing effort, it could be used for large-scale comparisons between markup handling methods. But also, it could be of use for language service providers that have to negotiate discounts with clients if the machine-translated data contains markup.

Modify core methods instead of pre- and postprocessing

As I have pointed out in Section 2.1.3, all strategies I have described are implemented as pre- and post-processing steps. As such, they are – to a certain extent – independent of the models and decoding algorithm used. But on the other hand, they only have indirect ways of influencing model training and decoding. Markup handling could also be integrated more tightly into the system, for instance by writing additional feature functions for the log-linear model. I can think of feature functions like `malformednessPenalty` and `invalidityPenalty` that could encourage the decoder to select candidates that have intact markup.

Applicability to neural machine translation

It is unclear to what extent the results I and others before me have obtained are limited to phrase-based, statistical machine translation – or even, limited to the Moses framework that all publications including this thesis have used. Other, more advanced paradigms of machine translation might obviate the need for elaborate markup handling. For example, it might be the case that recent neural machine translation systems generalize much better and can actually learn that certain parts of a segment (markup) have a fixed translation. It would be exceptionally interesting to test such a hypothesis.

Applicability to document-level translation

Phrase-based, statistical machine translation typically translates sentence by sentence and there is absolutely no document-wide context. This has advantages (among other things, translation scales well because it can be parallelized), but consistent terminology or anaphora are a problem under certain circumstances. Document-level decoding (Hardmeier et al., 2013) addresses those issues. It is unclear how markup handling strategies would have to be adapted for document-level translation. For instance, identity masking might no longer be feasible because there would be a lot more unique IDs if the unit of translation were a whole document.

User studies

Although I have argued that improper markup handling has an impact on translator productivity, there is little evidence to support this claim (Escartín and Arcedillo, 2015). Although it is extremely laborious, a user study could shed light on the issue. In such an experiment, professional translators could be asked to post-edit machine translations produced by several engines, varying the markup handling strategy that was used in each system. The productivity of translators would be relatively easy to measure, for instance simply by calculating the post-editing time averaged over all segments.

References

Alabau, V., Bonk, R., Buck, C., Carl, M., Casacuberta, F., García-Martínez, M., González, J., Koehn, P., Leiva, L., Mesa-Lao, B., et al. (2013). Casmacat: An open source workbench for advanced computer aided translation. *The Prague Bulletin of Mathematical Linguistics*, 100:101–112.

Banerjee, S. and Lavie, A. (2005). *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, chapter METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments, pages 65–72. Association for Computational Linguistics.

Barlow, G. W. (2002). *The cichlid fishes: nature's grand experiment in evolution*. Basic Books.

Brown, P. F., Cocke, J., Pietra, S. A. D., Pietra, V. J. D., Jelinek, F., Lafferty, J. D., Mercer, R. L., and Roossin, P. S. (1990). A statistical approach to machine translation. *Computational linguistics*, 16(2):79–85.

Brown, P. F., Pietra, V. J. D., Pietra, S. A. D., and Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311.

Ceruzzi, P. E. (2003). *A history of modern computing*. MIT press.

Chen, S. F. and Goodman, J. (1996). An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics.

Cherry, C. and Foster, G. (2012). Batch tuning strategies for statistical machine translation. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 427–436. Association for Computational Linguistics.

Clark, J. H., Dyer, C., Lavie, A., and Smith, N. A. (2011). Better hypothesis testing for statistical machine translation: Controlling for optimizer instability. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 176–181. Association for Computational Linguistics.

Denkowski, M. and Lavie, A. (2011). METEOR 1.3: Automatic metric for reliable optimization and evaluation of machine translation systems. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 85–91. Association for Computational Linguistics.

Devlin, J., Zbib, R., Huang, Z., Lamar, T., Schwartz, R., and Makhoul, J. (2014). Fast and robust neural network joint models for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1370–1380, Baltimore, Maryland. Association for Computational Linguistics.

Doddington, 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*, HLT '02, pages 138–145, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Du, J., Roturier, J., and Way, A. (2010). TMX markup: a challenge when adapting smt to the localisation environment. In *EAMT - 14th Annual Conference of the European Association for Machine Translation*. European Association for Machine Translation.

Dyer, C., Chahuneau, V., and Smith, N. A. (2013). A simple, fast, and effective reparameterization of ibm model 2. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 644–648, Atlanta, Georgia. Association for Computational Linguistics.

Dyer, C., Weese, J., Setiawan, H., Lopez, A., Ture, F., Eidelman, V., Ganitkevitch, J., Blunsom, P., and Resnik, P. (2010). cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *Proceedings of the ACL 2010 System Demonstrations*, pages 7–12. Association for Computational Linguistics.

Escartín, C. P. and Arcedillo, M. (2015). Machine translation evaluation made fuzzier: A study on post-editing productivity and evaluation metrics in commercial settings. In *Proceedings of MT Summit XV*, pages 131–144. Association for Machine Translation in the Americas.

Federico, M., Bertoldi, N., and Cettolo, M. (2008). IRSTLM: an open source toolkit for handling large scale language models. In *Interspeech*, pages 1618–1621.

Federico, M., Bertoldi, N., Cettolo, M., Negri, M., Turchi, M., Trombetti, M., Cattelan, A., Farina, A., Lupinetti, D., Martines, A., et al. (2014). The Matecat Tool. In *COLING (Demos)*, pages 129–132.

Filip, D. and Morado Vázquez, L. (2013). XLIFF Support in CAT Tools. Technical report, OASIS XLIFF Promotion and Liaison Subcommittee.

Forcada, M. L., Ginestí-Rosell, M., Nordfalk, J., O'Regan, J., Ortiz-Rojas, S., Pérez-Ortiz, J. A., Sánchez-Martínez, F., Ramírez-Sánchez, G., and Tyers, F. M. (2011). Apertium: a free/open-source platform for rule-based machine translation. *Machine translation*, 25(2):127–144.

Galley, M. and Manning, C. D. (2008). A simple and effective hierarchical phrase reordering model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 848–856.

Germann, U., Samiotou, A., Ruopp, A., Bertoldi, N., Cettolo, M., Cattoni, R., Federico, M., Madl, D., Caroselli, D., and Mastrostefano, L. (2016). MMT – modern machine translation, first technology assessment report. Technical report, MMT – Modern Machine Translation.

Green, S., Heer, J., and Manning, C. D. (2013). The efficacy of human post-editing for language translation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 439–448. ACM.

Hardmeier, C., Stymne, S., Tiedemann, J., and Nivre, J. (2013). Docent: A document-level decoder for phrase-based statistical machine translation. In *ACL 2013 (51st Annual Meeting of the Association for Computational Linguistics); 4-9 August 2013; Sofia, Bulgaria*, pages 193–198. Association for Computational Linguistics.

Heafield, K. (2011). KenLM: Faster and smaller language model queries. In *Proceedings of the EMNLP 2011 Sixth Workshop on Statistical Machine Translation*, pages 187–197, Edinburgh, Scotland.

Hudík, T. and Ruopp, A. (2011). The integration of Moses into localization industry. In *15th Annual Conference of the EAMT*, pages 47–53.

Hutchins, J. (1994). Machine translation: History and general principles. *The encyclopedia of languages and linguistics*, 5:2322–2332.

Hutchins, J. (1995a). Reflections on the history and present state of machine translation. In *Proc. of Machine Translation Summit V*, pages 89–96.

Hutchins, J. (1997). From first conception to first demonstration: the nascent years of machine translation, 1947–1954. a chronology. *Machine Translation*, 12(3):195–252.

Hutchins, J. (2000a). Warren Weaver and the Launching of MT. *Early Years in Machine Translation, ed. W. John Hutchins. Amsterdam: John Benjamins*, pages 17–20.

Hutchins, J. (2005). The history of machine translation in a nutshell. *Web publication, retrieved December 10, 2016*.

Hutchins, J. (2007). Machine translation: A concise history. *Computer aided translation: Theory and practice*.

Hutchins, W. J. (1986). *Machine translation: past, present, future*. Ellis Horwood Chichester.

Hutchins, W. J. (1995b). Machine translation: A brief history. *Concise history of the language sciences: from the Sumerians to the cognitivists*, pages 431–445.

Hutchins, W. J. (2000b). *Early years in machine translation: memoirs and biographies of pioneers*, volume 97. John Benjamins Publishing.

Hutchins, W. J. (2001). Machine translation over fifty years. *Histoire epistémologie langage*, 23(1):7–31.

Joanis, E., Stewart, D., Larkin, S., and Kuhn, R. (2013). Transferring markup tags in statistical machine translation: A two-stream approach. In O'Brien, S., Simard, M., and Specia, L., editors, *Proceedings of MT Summit XIV Workshop on Post-editing Technology and Practice*, pages 73–81.

Koehn, P. (2009). A process study of computer-aided translation. *Machine Translation*, 23(4):241–263.

Koehn, P. (2010). *Statistical Machine Translation*. Cambridge University Press.

Koehn, P., Axelrod, A., Birch, A., Callison-Burch, C., Osborne, M., Talbot, D., and White, M. (2005). Edinburgh system description for the 2005 IWSLT speech translation evaluation. In *IWSLT*, pages 68–75.

Koehn, P. and Haddow, B. (2009). Edinburgh's submission to all tracks of the WMT2009 shared task with reordering and speed improvements to Moses. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 160–164. Association for Computational Linguistics.

Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., Dyer, C., Bojar, O., Constantin, A., and Herbst, E. (2007). Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, pages 177–180.

Koehn, P., Och, F. J., and Marcu, D. (2003). Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 48–54.

Koehn, P. and Senellart, J. (2010). Convergence of translation memory and statistical machine translation. In *Proceedings of AMTA Workshop on MT Research and the Translation Industry*, pages 21–31.

Krings, H. P. (2001). *Repairing texts: empirical investigations of machine translation post-editing processes*, volume 5. Kent State University Press.

Kuhn, T. S. (2012). *The structure of scientific revolutions*. University of Chicago press.

Läubli, S., Fishel, M., Massey, G., Ehrensberger-Dow, M., and Volk, M. (2013). Assessing post-editing efficiency in a realistic translation environment. In *Proceedings of Workshop on Post-editing Technology and Practice*, pages 83–91.

Mani, I. (2001). *Automatic Summarization*, volume 3 (Natural Language Processing). John Benjamins Publishing.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA.

Martínez-Gómez, P., Sanchis-Trilles, G., and Casacuberta, F. (2012). Online adaptation strategies for statistical machine translation in post-editing scenarios. *Pattern Recognition*, 45(9):3193–3203.

Morado Vázquez, L. and Filip, D. (2014). XLIFF Version 2.0 Support in CAT Tools. Technical report, OASIS XLIFF Promotion and Liaison Subcommittee.

OASIS XLIFF Technical Committee (2008). XLIFF Version 1.2. Standard, Organization for the Advancement of Structured Information (OASIS).

Och, F. J. (2003). Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 160–167.

Och, F. J. and Ney, H. (2003). A systematic comparison of various statistical alignment models. *Computational linguistics*, 29(1):19–51.

Ortiz-Martínez, D. and Casacuberta, F. (2014). The New Thot Toolkit for Fully Automatic and Interactive Statistical Machine Translation. In *Proc. of the European Association for Computational Linguistics (EACL): System Demonstrations*, pages 45–48, Gothenburg, Sweden.

Ortiz-Martínez, D., García-Varea, I., and Casacuberta, F. (2008). Phrase-level alignment generation using a smoothed loglinear phrase-based statistical alignment model. In *Proc. of EAMT*, volume 8.

O'Brien, S. (2011). Towards predicting post-editing productivity. *Machine translation*, 25(3):197–215.

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*, ACL '02, pages 311–318, Stroudsburg, PA, USA. Association for Computational Linguistics.

Ruopp, A. (2010). The Moses for Localization Open Source Project. In *Proceedings of the Ninth Conference of the Association for Machine Translation in the Americas (AMTA)*.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423.

Shannon, C. E. (2001). A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55.

Snover, M., Dorr, B., Schwartz, R., Micciulla, L., and Makhoul, J. (2006). A study of translation edit rate with targeted human annotation. In *In Proceedings of Association for Machine Translation in the Americas*, pages 223–231.

Stolcke, A. et al. (2002). SRILM – an extensible language modeling toolkit. In *Interspeech*, volume 2002, page 2002.

Talbot, D. and Osborne, M. (2007). Randomised language modelling for statistical machine translation. In *ACL*, volume 7, pages 512–519.

Tezcan, A. and Vandeghinste, V. (2011). SMT-CAT integration in a Technical Domain: Handling XML Markup Using Pre & Post-processing Methods. *Proceedings of EAMT 2011*.

Varga, D., Halácsy, P., Kornai, A., Nagy, V., Németh, L., and Trón, V. (2007). Parallel corpora for medium density languages. *Amsterdam Studies in the Theory and History of Linguistic Science Series 4*, 292:247.

Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57.

Weaver, W. (1955). Translation. *Machine translation of languages*, 14:15–23.

Williams, M. R. (1997). *A history of computing technology*. IEEE Computer Society Press.

Zhechev, V. and van Genabith, J. (2010). Seeding statistical machine translation with translation memory output through tree-based structural alignment. In *Proceedings of the 4th Workshop on Syntax and Structure in Statistical Translation*, pages 43–51, Beijing, China. Coling 2010 Organizing Committee.

Appendix

A Installation Requirements of `mtrain`

Installing `mtrain` is straightforward, but there are a number of installation requirements that I would like to list here. Before installation, make sure to have the following:

- **Unix operating system**: `mtrain` is only guaranteed to work on recent distributions of Ubuntu and Debian, but in principle, any Unix system should work. However, we have not tested it on either Mac OS X or Windows (where it could run in Cygwin).
- **Python 3**: `mtrain` is developed and tested with Python version 3.5.2. It presumably works with later versions of Python, but certainly not with Python 2.X. If your system runs Python 2.X by default, run Python 3.X in a virtual environment, for instance with `virtualenv`.
- **Pip 3**: the easiest way to install `mtrain` is with `pip`, a package manager for Python, but only the Python 3 version of it will work.

This will already allow you to install `mtrain` successfully. To be able to run it, you will also need:

- **nose**: if you plan to run the regression tests that ship with the framework, the `nose` package must be installed. If you are not a developer, this might not be of interest to you.
- **lxml**: a very popular XML library for Python that is a requirement if markup handling is enabled.
- **Moses version 3.0**: since `mtrain` is a wrapper around Moses, the latter must be installed on your system. `mtrain` is tested with the 3.0 release of Moses and might also work with later versions, but not with earlier ones. Make sure to compile with `cmph` (for compact phrase and reordering tables).
- **fast_align and atools**: for computing word alignments and symmetrization, `mtrain` uses `fast_align` and `atools`. Compile them with `OpenMP` which will enable multi-threading in those tools.

For more detailed information, please have a look at the docstrings of all modules, classes and functions in the software and at `README.md` if your copy is cloned from the `git` repository.

Curriculum Vitae

Personal details

Full name	Mathias Mueller
Address	Toggenburgerstrasse 68 9500 Wil SG Switzerland
Date of birth	March 15, 1991
Email address	mathias.mueller@uzh.ch

Education

2014 – 2016	Master of Arts UZH in Computational Linguistics and Language Technology, with a focus on Phonetics and Statistical Machine Translation
2009 – 2013	Bachelor of Arts UZH in Computational Linguistics, German Linguistics and English Linguistics

Relevant Professional Activities

2016	Research Assistant at the Institute of Computational Linguistics UZH, contributing to the projects *SPARCLING* and *Text+Berg*
2015 – 2016	Machine Translation Engineer in the project *Machine Translation of Help Desk Tickets* led by the Institute of Computational Linguistics UZH and funded by finnova AG Bankware
2015	XML Consultant for the Swiss Law Sources Foundation
2013 – 2014	Internship as a Computational Linguist at finnova AG Bankware, Lenzburg AG
2010 – 2016	Student Research Assistant in the Phonetics Laboratory UZH

Teaching Experience

2016	Teaching Assistant for the module *XML Technologies and Semantic Web*
2015	Teaching Assistant for the module *Machine Translation and Parallel Corpora*
2014	Teaching Assistant for the module *XML Technologies and Semantic Web*
2012	Teaching Assistant for the module *Sprachtechnologie für Barrierefreiheit*

Universität Zürich[UZH]

Philosophische Fakultät
Studiendekanat

Universität Zürich
Philosophische Fakultät
Studiendekanat
Rämistrasse 69
CH-8001 Zürich
www.phil.uzh.ch

Selbstständigkeitserklärung

Hiermit erkläre ich, dass die Masterarbeit von mir selbst ohne unerlaubte Beihilfe verfasst worden ist und ich die Grundsätze wissenschaftlicher Redlichkeit einhalte (vgl. dazu: http://www.uzh.ch/de/studies/teaching/plagiate.html).

Zürich, 22.12.2016

Ort und Datum

Unterschrift

Seite 1/1

27.7.2016