Interoperability in MT Quality Estimation or wrapping useful stuff in various ways

Eleftherios Avramidis
German Research Center for Artificial Intelligence (DFKI)
Language Technology Group
Alt Moabit 91c, 10559 Berlin
eleftherios.avramidis@dfki.de

Abstract

The situation on the interoperability of Natural Language Processing software is outlined through a use-case on Quality Estimation of Machine Translation output. The focus is on the development efforts for the QUALITATIVE tool, so that it integrates a multitude of state-of-the-art external tools into one single Python program, through an interoperable framework. The presentation includes 9 approaches taken to connect 25 external components, developed in various programming languages. The conclusion is that the current landscape lacks important interoperability principles and that developers should be encouraged to equip their programs with some of the standard interaction interfaces.

Keywords: Machine Translation, Quality Estimation, interoperability

1. Introduction

Software development in Computational Linguistics and Natural Language Processing (NLP) has been for many years a means for scientific experimentation, primarily confined on an academic environment. This often had as a result that the software written and the data collected for this purpose lack most software engineering principles, as they mainly address the need for performing research experiments that can get results in a timely manner. During the last decade, Language Technology (LT) has jumped from the research labs to the industry and many open-source tools and data that had originally been written for research purposes have ended being used in a wide scale, albeit leading to a very diverse and multilateral landscape.

For both industry and academic research, re-using software and data seems the obvious solution. It means saving effort and time in development to focus on the innovation, but also easily reproducing (and therefore confirming) state-of-the-art methods. Luckily, most of the tools and data are available with open or reusable licenses, but combining many of them into one LT application remains a challenge: every tool may be written in a different programming language, utilizing a different file format or providing a different (or no) external interface.

Our current contribution outlines the situation through a use-case on Quality Estimation of Machine Translation (MT) output. We look on the QUALITATIVE tool (Avramidis et al., 2014), that combines a multitude of state-of-the-art tools into one single application, which can be used both for research and real-time use. We describe the approaches taken to achieve an interoperable framework that bridges the communication between the several components in order to achieve the desired processing of data in a functional way. Although we do not provide a unified wide-scale solution, by sharing the experience of our own development from the perspective of MT Evaluation, we aim to highlight a part of the ecosystem and raise the awareness of how difficult and challenging it is to get everything together without re-writing from scratch.

2. Previous work

The idea of complex interoperable pipelines is not new in NLP or MT specifically. Most of such tasks consist of many tools in order to process the data and extract all sorts of linguistic knowledge and analyses. Among the most popular tools are the pipelines for training and evaluating Statistical Machine Translation systems. Consequently, we are reviewing some of the most prominent relevant frameworks.

EMS (Koehn, 2010) is the pipeline for training models for Statistical MT through Moses. Originally written as a single very long Perl script, it has been extended and adapted through the years to include more than 30 components. EMS wraps each external tool in a `bash` script launched through the shell, whereas data transfer between the components is done through temporary files on the disk or shell pipes. The majority of the launched programs are written in Perl and C++, whereas there are also some in Python and Java. The advantages of EMS are that it is very modular and it can wrap any program that operates on the Linux commandline. Additionally, it can run as an injection script for the Sun Grid in order to distribute and parallelize tasks across many computational servers. EMS could in principle be adapted to function for other type of experiments apart for MT, though not many examples have been reported.

TREEX (Popel and Žabokrtský, 2010) is a similar pipeline offering sentence-level processing, a Perl API and a socket server, whereas its main use is focused on Statistical MT with deep transfer methods. **LOONYBIN** (Clark and Lavie, 2010) follows a similar approach. Coded in Jython, it allows the user to use modules in Python for wrapping tools in hyper-workflows. Similar approaches are followed by tools such as **EMAN** (Bojar and Tamchyna, 2013). One disadvantage of such tools is that `bash` wrappers, temporary files and pipes can mostly operate efficiently for batches of data and not for single sentences, as the case is for many user-oriented applications. Additionally, such a pipeline itself cannot be very efficiently incorporated in another pipeline, unless it is written in the same tool. There are other experiment pipeline tools such as DagMan, Dryad, DuctTape, PCL, Pegasus, SoapLab, Taverna and UIMA, but we won't focus on them, since they have not been widely used in MT research.

An interesting contribution towards interoperability was the **Open Machine Translation Core** (Johnson, 2013) which attempted to define an abstract interface that standardizes functionality common to all MT systems. Despite the fact that a Java prototype was presented, to our knowledge no progress has been shown in the direction of this standardization by now.

QuEst (Specia et al., 2013) was the first software that appeared to wrap many tools for Quality Estimation. Written in Java, it incorporates directly a few other Java applications as libraries, whereas other tools are also wrapped by using the `bash` shell and intermediate temporary batch files. The machine learning part is held by a separate Python script. For this reasons, running a unified pipeline with realtime user requests (e.g. server mode) is non-trivial.

3. Basic architecture

In our approach, the main core of the program is written in Python. Python has been chosen because it offers the flexibility of dynamic programming, which allows for quick and relatively easy experimentation in many NLP tasks. Functionality from several powerful scientific and machine learning toolkits is available through imported libraries. Additionally, a Python script can be connected to real user applications, either through a web server (e.g. Django), or by offering its functionality via a socket service. This choice offers a flexible framework for both experimentation and practice, although it has got its own limitations (e.g. processing cannot be distributed in many computational machines without additional engineering).

As explained in Avramidis et al. (2014), the program is internally organized in several modules:

- **data reading** receives a file and loads the data in the memory via the respective data structures
- **preprocessing** sends a sentence for the required preprocessing task (e.g. tokenization, compound splitting, truecasing etc.)
- **machine translation** sends source sentences to MT engines and receives their translation
- **feature generation** sends the source sentences and their translations to feature generatior classes and tools and receives the respective vectors of numerical features
- **machine learning** serves for the communication with machine learning toolkits for two functions: training and testing. During training, it sends a batch of vectors, each one with a golden label and it receives a model. During testing, the model is loaded and given a vector, the predicted label is returned.

For each module, the commands are organized so that they form a specific interface as a principle of internal *modularity*. This way, the same functionality can be implemented by different classes. For example, every feature generator class has to implement at least one function that receives a source sentence and its translations and returns a vector of numerical features.

4. Connecting external components

We present two main categories of communicating with external software components, based on whether the execution of the external software is controlled by the our Python script, which we will call the "host", or whether it is run as a remote service.

4.1. Inherent integration

In these functions, the execution of the external software is encapsulated into the host. The goal is to keep the external tool running in the background so that it can receive requests from the host. It gets automatically unloaded when the host program is finished. The part of the host program or the code which handles the specifities of the communication is referred to as at "connector".

4.1.1. Native Python libraries

Many pieces of Python open-source software already offer their functionality in openly available libraries. This is the easiest and most efficient type of integration, as all of the public functions of the included software can be directly called from within our host Python code. The software served by this method includes:

- BLEU (Papineni et al., 2001), Levenshtein Distance (Levenshtein, 1966) and RgbF (Popović, 2012) for MT evaluation scores
- HJERSON (Popović, 2011) for automatic detection of MT errors
- KENLM (Heafield, 2011) for language modelling
- MLPYTHON[1], ORANGE (Demšar et al., 2004)and SCIKIT-LEARN (Pedregosa et al., 2011) for machine learning functions.
- NLTK (Loper and Bird, 2002) for several simple NLP tasks
- NUMPY (Van Der Walt et al., 2011) for memory-efficient handling of numerical arrays and SCIPY (Oliphant, 2007) for scientific (e.g. complex mathematical or statistical) functions.

4.1.2. Java programs

Py4j[2] was chosen as a solution to integrate functionality from open-source Java programs into Python. The Java Virtual Machine (JVM) starts in the background including the required Java Packages (jar) in the classpath. Then, a Py4j gateway connects with the JVM via a socket and makes all public classes and functions loadable and callable from within Python. Python types are automatically converted to Java types and vice versa. If the processes are thread-safe

[1] `http://www.dmi.usherb.ca/~larocheh/mlPython/`

[2] `http://www.py4j.org`

on the Java side, they can be also parallelized in several Python threads.
This method is used to connect with:

- BERKELEY PARSER (Petrov et al., 2006) for parsing with Probabilistic Context Free Grammars (PCFG),
- LANGUAGE TOOL (Naber, 2003; Miłkowski, 2012) for rule-based language checking and
- METEOR (Lavie and Agarwal, 2007; Denkowski and Lavie, 2014) for MT evaluation scoring.

This method is efficient and allows wide access and parametrization to the functionality of the external Java program. Nevertheless, it also requires good knowledge to its internal structure, e.g. via a Java API documentation or by reading the Java source code. This is needed because the imported objects, functions and variables have to be treated in Python the same way they would do in Java. Additionally, the host needs to know or maintain a knowledge of the system socket where the JVM operates, which makes it complicated to run many hosts on the same JVM. In a few cases, parts of the source code had to be modified and be re-build, since not all required functions were declared as public, which is a major requirement.

4.1.3. SWIG

Simplified Wrapper and Interface Generator (SWIG) allows wrapping C++ code as a Python library. Creating such a connector allows to parse C/C++ interfaces and generate the 'glue code' for Python to call into the C/C++ code. In our program we have not developed such a connector, but we have experimented with SWIG-SRILM (Madnani, 2009), an existing wrapper around SRILM (Stolcke, 2002).

4.1.4. Pipes

An external commandline-based software is launched by the host as a sub-process in the background. The standard input, the standard output and the error output can be captured within a Python object (a *pipe*). Therefore, a program-specific connector needs to be written. It should be aware of the commandline behaviour of the software and simulate that through the pipe. The sub-process is treated as a black-box, i.e. no access to particular internal functions is possible.
For example, a standard tokenizer from the MOSES scripts would read from the standard input all characters, waiting for and "end of line". Once the "end of line" is received, the tokenization takes place and the tokenized string is returned through the standard output.
This approach is mainly used for Perl scripts and C++ programs but can be adapted for any commandline application. Such software includes:

- MOSES scripts for pre-processing and post-processing, such as punctuation normalizer, tokenizer, compound splitter (Koehn and Knight, 2003), truecaser (Och et al., 2003), de-truecaser, de-tokenizer etc. Although re-implementations for most of these exist in Python and therefore could be directly included in our code, one may still require to stick to the original MOSES Perl scripts, if they want to re-use pre-trained MOSES translation models or acquire results comparable with other scientific works that use these state-of-the-art Perl scripts.
- TREETAGGER for POS tagging (Schmid, 1994) integrated via the TreeTaggerWrapper (Pointal, 2015).

The advantage of this method is that it can be adapted for many programs without requiring knowledge of their internal coding or functioning, while it still allows loading a tool into memory once and sending individual requests when the host program needs it. The disadvantage is that the only way of interaction is through the standard input and output, which offer no flexibility for parametrization or passing more complex types. Additionally, reading standard output often requires excessive use of regular expressions to understand some complex output, which would otherwise be intended for the visual understanding of the user. Unexpected errors and exceptions are hard to capture, too. We should also mention that some tools only work with input and output files (batch mode) and do not support per-request communication with standard input and output. Finally, serious deficiencies have been noted concerning the buffering support of the pipes, which may cause prevent data to be transferred through the standard input/output.

4.1.5. Shell with external files

The data to process is written by the host on a temporary file. The external program is launched once, asked to process the given temporary file as an input and write its output in another temporary file, which consequently gets read by the host. This is the last resort for having the host communicate with external tools, since loading the entire program per request and writing external files is not efficient for single sentences and is useful only for processing batches of requests. We also noticed that some programs of this kind do not allow many instances to be run in parallel (e.g. because they require an exclusive lock on some internal files, whose location is often non-parametrizable).
We used this method for aligning sentences with GIZA++ (Och and Ney, 2003), acquiring baseline features from QUEST and doing PCFG parsing with BITPAR (Schmid, 2004) with the help of a wrapper (van Cranenburgh, 2010). This method was useful only for experiments that did not require parallelization and single requests.

4.2. Integrating functionality as a remote service

An additional possibility of integrating an external tool is by sending requests to it as a remote service. In this case, the external tool must provide a server which initially loads the program and implements a network protocol of requests and responses. It waits until a request is received from the host, in order to run the required functions. The result of the functions is then sent with a corresponding response.
Four such protocols and the respective tools we have used are:

- **JSON**: with MT-MONKEY (Tamchyna et al., 2013), which acts as a hub and a load balancer for fetching translations from several MT engines

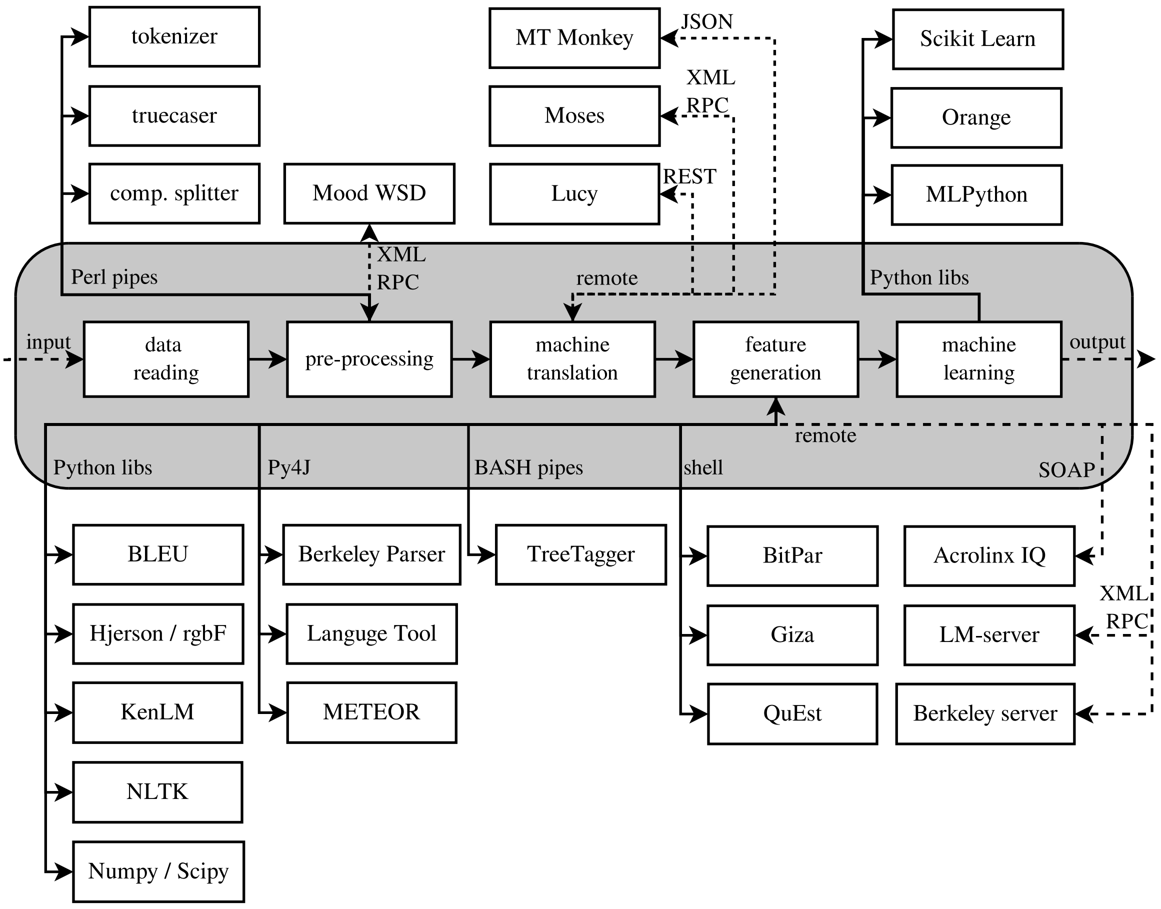

Figure 1: Full diagram of the components that have been integrated into the application

- **SOAP**: with ACROLINX IQ (Siegel, 2011) for language checking
- **REST**: with the LUCY rule-based MT system (Alonso and Thurmair, 2003).
- **XML-RPC**: with MOSES (Koehn et al., 2006) for Statistical MT, with LM-SERVER (Madnani, 2009) for language model scoring, with our own XML-RPC wrapper of BERKELEY PARSER and with MOOD, (Weissenborn et al., 2015) a Word Sense Disambiguation analyzer.

Such an integration is straightforward if the tool already provides such a protocol interface, since the protocols allow for easy mapping of function and variable types across many different programming languages. This solution is based on a network connection, so it is also desirable when one needs to distribute different computationally or memory intensive modules to many computational servers. Nevertheless, such a network communication may be considerably slower due to the network overhead. Additionally, starting and stopping remote services cannot be easily controlled by the host, unlike to the encapsulation described in the previous section.

5. Discussion

The ecosystem is indeed complicated. Integrating existing software saves time from re-implementing it and can confirm replicability of scientific experiments. Nevertheless, as we outlined in our use-case, the different types of software may require different kind of integration. Such an integration often requires low-level or even backwards engineering, which means a lot of non-creative effort.

An obvious conclusion through our experience is that reusability and efficient interoperability mostly depends on the will of the original developer. Adding support for a network service or exporting a Python library is straightforward for the original developers of a software, in contrast to the huge effort required for a third-party developer to understand the functionality and wrap it one way or another. It suffices to mention that out of the 25 external tools and libraries that we integrated, only 5 provided original support (remote service or library) for being integrated with a programming language other than the one they were originally developed in.

In that direction, the specification of a unified way to communicate across different code and platforms would be precious. Whatsoever, even encouraging developers to consider serious solutions for the interoperability of their software would be a major first step. Among the most obvious solutions, we would consider wrapper libraries in the most popular scripting languages (e.g. Python, Perl) and exposing full functionality through a ReSTful service (Richardson and Ruby, 2008), possibly along with "autodiscovery" API functions via the WSDL (Christensen et al., 2001).

Acknowledgment

This work has received support by the EC's FP7 (FP7/2007-2013) under grant agreement number 610516: "QTLeap: Quality Translation by Deep Language Engineering Approaches". Early stages have been developed with the support of the projects TaraXŰ and QT-Launchpad.

References

Alonso, J. A. and Thurmair, G. (2003). The Comprendium Translator system. In *Proceedings of the Ninth Machine Translation Summit*. International Association for Machine Translation (IAMT).

Avramidis, E., Poustka, L., and Schmeier, S. (2014). Qualitative: Open source Python tool for Quality Estimation over multiple Machine Translation outputs. *The Prague Bulletin of Mathematical Linguistics*, 102:5–16.

Bojar, O. and Tamchyna, A. (2013). The Design of Eman, an Experiment Manager. *Prague Bull. Math. Linguistics*, 99:39–58.

Christensen, E., Curbera, F., Meredith, G., and Sanjiva Weerawarana. (2001). Web Services Description Language (WSDL) 1.1. Technical report, World Wide Web Consortium.

Clark, J. H. and Lavie, A. (2010). LoonyBin: Keeping Language Technologists Sane through Automated Management of Experimental (Hyper)Workflows. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation*, pages 1301–1308.

Demšar, J., Zupan, B., Leban, G., and Curk, T. (2004). Orange: From Experimental Machine Learning to Interactive Data Mining. In *Principles of Data Mining and Knowledge Discovery*, pages 537–539.

Denkowski, M. and Lavie, A. (2014). Meteor Universal: Language Specific Translation Evaluation for Any Target Language. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 376–380, Baltimore, Maryland, USA, jun. Association for Computational Linguistics.

Heafield, K. (2011). KenLM : Faster and Smaller Language Model Queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, number 2009, pages 187–197, Edinburgh, Scotland, jul. Association for Computational Linguistics.

Johnson, I. (2013). Open Machine Translation Core: An Open API for Machine Translation Systems. *The Prague Bulletin of Mathematical Linguistics*, 100:91–100.

Koehn, P. and Knight, K. (2003). Empirical Methods for Compound Splitting. In *Proceedings of the 10th Annual Conference of the European Association for Machine Translation*, volume 1, page 8, Budapest, Hungary.

Koehn, P., Shen, W., Federico, M., Bertoldi, N., Callison-Burch, C., Cowan, B., Dyer, C., Hoang, H., Bojar, O., Zens, R., Constantin, A., Herbst, E., and Moran, C. (2006). Open Source Toolkit for Statistical Machine Translation. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 177–180, Prague, Czech Republic, jun.

Koehn, P. (2010). An Experimental Management System Philipp Koehn. *The Prague Bulletin of Mathematical Linguistics*, (94):87–96.

Lavie, A. and Agarwal, A. (2007). METEOR: An Automatic Metric for MT Evaluation with High Levels of Correlation with Human Judgments. In *Proceedings of the Second Workshop on Statistical Machine Translation*, pages 228–231, Prague, Czech Republic, jun. Association for Computational Linguistics.

Levenshtein, V. (1966). Binary Codes Capable of Correcting Deletions and Insertions and Reversals. *Soviet Physics Doklady*, 10(8):707–710.

Loper, E. and Bird, S. (2002). NLTK: The Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02, pages 63–70, Stroudsburg, PA, USA. Association for Computational Linguistics.

Madnani, N. (2009). Source Code: Querying and Serving N-gram Language Models with Python. *The Python Papers Source Codes*.

Miłkowski, M., (2012). *Translation Quality Checking in LanguageTool*, pages 213–223. Corpus Data across Languages and Disciplines. Peter Lang, Frankfurt am Main, Berlin, Bern, Bruxelles, New York, Oxford, Wien.

Naber, D. (2003). A rule-based style and grammar checker. Technical report, Bielefeld University, Bielefeld, Germany.

Och, F. J. and Ney, H. (2003). A Systematic Comparison of Various Statistical Alignment Models. In *Computational Linguistics*.

Och, F., Gildea, D., Khudanpur, S., Sarkar, A., Yamada, K., Fraser, A., Kumar, S., Shen, L., Smith, D., Eng, K., and Others. (2003). Syntax for statistical machine translation. In *Johns Hopkins University 2003 Summer Workshop on Language Engineering, Center for Language and Speech Processing, Baltimore, MD, Tech. Rep.* Citeseer.

Oliphant, T. E. (2007). Python for Scientific Computing. *Computing in Science & Engineering*, 9(3):10–20.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2001). Bleu: a Method for Automatic Evaluation of Machine Translation. IBM Research Report RC22176(W0109-022), IBM.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine Learning in {P}ython. *Journal of Machine Learning Research*, 12:2825–2830.

Petrov, S., Barrett, L., Thibaux, R., and Klein, D. (2006). Learning Accurate, Compact, and Interpretable Tree Annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 433–440, Sydney, Australia, jul. Association for Computational Linguistics.

Pointal, L. (2015). TreeTagger Wrapper.

Popel, M. and Žabokrtský, Z. (2010). TectoMT: Modular NLP Framework. In *Proceedings of the 7th International Conference on Advances in Natural Language Processing*, IceTAL'10, pages 293–304, Berlin, Heidelberg. Springer-Verlag.

Popović, M. (2011). Hjerson: An Open Source Tool for Automatic Error Classification of Machine Translation Output. *The Prague Bulletin of Mathematical Linguistics*, 96(-1):59–68.

Popović, M. (2012). rgbF: An Open Source Tool for n-gram Based Automatic Evaluation of Machine Translation Output. *The Prague Bulletin of Mathematical Linguistics*, (98):99–108.

Richardson, L. and Ruby, S. (2008). *RESTful web services*. " O'Reilly Media, Inc.".

Schmid, H. (1994). Probabilistic part-of-speech tagging using decision trees. In *Proceedings of International Conference on New Methods in Language Processing*, Manchester, UK.

Schmid, H. (2004). Efficient Parsing of Highly Ambiguous Context-free Grammars with Bit Vectors. In *Proceedings of the 20th International Conference on Computational Linguistics*, COLING '04, Stroudsburg, PA, USA. Association for Computational Linguistics.

Siegel, M. (2011). Autorenunterstützung für die Maschinelle Übersetzung. In *Multilingual Resources and Multilingual Applications: Proceedings of the Conference of the German Society for Computational Linguistics and Language Technology (GSCL)*, Hamburg.

Specia, L., Shah, K., de Souza, J. G. C., and Cohn, T. (2013). QuEst - A translation quality estimation framework. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 79–84, Sofia, Bulgaria, aug. Association for Computational Linguistics.

Stolcke, A. (2002). SRILM — an Extensible Language Modeling Toolkit. In *System*, volume 2, pages 901–904. ISCA, sep.

Tamchyna, A., Dušek, O., Rosa, R., and Pecina, P. (2013). MTMonkey: A Scalable Infrastructure for a Machine Translation Web Service. *The Prague Bulletin of Mathematical Linguistics*, 100:31–40, oct.

van Cranenburgh, A. (2010). Enriching Data-Oriented Parsing by blending morphology and syntax. Technical report, University of Amsterdam, Amsterdam.

Van Der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2):22–30.

Weissenborn, D., Hennig, L., Xu, F., and Uszkoreit, H. (2015). Multi-Objective Optimization for the Joint Disambiguation of Nouns and Named Entities. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*, pages 596–605, Beijing, China. Association for Computer Linguistics.