Abstract.
We present an extended version of our open source framework for machine translation quality estimation, QUEST. The framework allows the extraction of a large variety of language- and machine translation system-dependent quality indicators from source segments, their translations, and external resources (i.e. source and target corpora, language models). It also provides machine learning algorithms to build quality estimation models. This new version adds to it more advanced, language-specific, and machine translation system-related indicators. We also present improvements over the latest version with respect to efficiency and user-friendliness. This deliverable describes the architecture of the framework, the list of additional features, documentation on how to further extend it to add new features, and on how to use the framework to build models for a given dataset. Finally, it also presents a web interface for remote access to facilitate the use of the framework by non-expert users.
<table>
<thead>
<tr>
<th>Grant agreement no.</th>
<th>296347</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project acronym</td>
<td>QTLlaunchPad</td>
</tr>
<tr>
<td>Project full title</td>
<td>Preparation and Launch of a Large-scale Action for Quality Translation Technology</td>
</tr>
<tr>
<td>Funding scheme</td>
<td>Coordination and Support Action</td>
</tr>
<tr>
<td>Coordinator</td>
<td>Prof. Hans Uszkoreit (DFKI)</td>
</tr>
<tr>
<td>Start date, duration</td>
<td>1 July 2012, 24 months</td>
</tr>
<tr>
<td>Distribution</td>
<td>Public</td>
</tr>
<tr>
<td>Contractual date of delivery</td>
<td>31 October 2013</td>
</tr>
<tr>
<td>Actual date of delivery</td>
<td>31 October 2013</td>
</tr>
<tr>
<td>Deliverable number</td>
<td>D2.1.2</td>
</tr>
<tr>
<td>Deliverable title</td>
<td>Quality Estimation Software Extensions</td>
</tr>
<tr>
<td>Type</td>
<td>Software</td>
</tr>
<tr>
<td>Status and version</td>
<td>Final</td>
</tr>
<tr>
<td>Number of pages</td>
<td>27</td>
</tr>
<tr>
<td>Contributing partners</td>
<td>USFD, DCU, DFKI</td>
</tr>
<tr>
<td>WP leader</td>
<td>USFD</td>
</tr>
<tr>
<td>Task leader</td>
<td>USFD</td>
</tr>
<tr>
<td>Authors</td>
<td>Lucia Specia, Kashif Shah, Eleftherios Avramidis</td>
</tr>
<tr>
<td>EC project officer</td>
<td>Aleksandra Wesolowska</td>
</tr>
</tbody>
</table>

The partners in QTLlaunchPad are:
- Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Germany
- Dublin City University (DCU), Ireland
- Institute for Language and Speech Processing, R.C. Athena (ILSP/ATHENA RC), Greece
- The University of Sheffield (USFD), United Kingdom

For copies of reports, updates on project activities and other QTLlaunchPad-related information, contact:

DFKI GmbH
QTLlaunchPad
Dr. Aljoscha Burchardt aljoscha.burchardt@dfki.de
Alt-Moabit 91c Phone: +49 (30) 23895-1838
10559 Berlin, Germany Fax: +49 (30) 23895-1810

Copies of reports and other material can also be accessed via http://www.qt21.eu/launchpad

© 2013, The Individual Authors

No part of this document may be reproduced or transmitted in any form, or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission from the copyright owner.
Deliverable D2.1.2 describes work on Task 2.1: **Quality Estimation Baseline and Extensions**. It provides an extended implementation of an open source framework for developing quality estimation models to be used within QTLaunchPad and beyond. This implementation focuses on extending the previous version with additional, more advanced features that can be language- and machine translation system-dependent. The number of features varies from language to language, from 80 to 191. The machine learning components for feature selection, optimisation, model learning and model evaluation are the same as those reported in D2.1.1.

This deliverable describes the architecture of framework with the list of additional features, and documentation on how to further extend it to add new features and on how to use the framework to build models for a given dataset (Chapter 2). It also presents a version for remote access and a web interface to facilitate the use of the framework by non-expert users (Chapter 3).
Contents

1 Introduction 2
  1.1 Relevance to QTLaunchPad ........................................... 3
  1.1.1 WP2 description ......................................................... 3
  1.1.2 Relation to other workpackages ..................................... 3

2 QuEst - features and architecture 6
  2.1 Features ................................................................. 6
  2.2 QuEst features and MQM .................................................. 9
  2.3 Source code ............................................................... 11
  2.4 Setting up ................................................................. 11
  2.5 The feature extractor .................................................... 12
  2.6 Running the feature extractor ......................................... 12
  2.7 Major packages and classes ........................................... 13
  2.8 Architecture ............................................................. 14
  2.9 Adding a new feature .................................................... 17

3 QuEst for lay-users 18
  3.1 Client-server architecture ............................................ 18
  3.2 A Web interface .......................................................... 20
  3.3 Performance considerations ........................................... 21

4 Remarks 24
Chapter 1

Introduction

QUeST is a framework for building and testing models of translation quality prediction. It consists of two main modules: a feature extraction module and a machine learning module. The first module provides a number of feature extractors, including the most commonly used features in the literature and by systems submitted to the WMT12-13 shared tasks on QE [Callison-Burch et al., 2012, Bojar et al., 2013]. It is implemented in Java and provides abstract classes for features, resources and pre-processing steps so that extractors for new features can be easily added.

The basic functioning of the feature extraction module requires a pair of raw text files with the source and translation sentences aligned at the sentence-level. Additional resources such as the machine translation system training corpus and decoder’s log, language models of source and target languages, parsers, etc. are necessary by more advanced features. Configuration files are used to indicate the resources available and a list of features that should be extracted. It produces a CSV file with all feature values.

The machine learning module provides scripts connecting the feature file(s) with the scikit-learn toolkit.\footnote{http://scikit-learn.org/} It also provides wrappers for GPy,\footnote{https://github.com/SheffieldML/GPy} a Python toolkit for Gaussian Processes regression, which showed good performance in previous work [Shah et al., 2013b, Beck et al., 2013, Specia et al., 2013].

QUeST also has a Web interface for remote access, where users can submit files with source and one or more translation sentences (or to get translations from Bing Translator) and get features and predictions for language pairs with models pre-built offline, as well as to perform a ranking of alternative translations of each source segment based on quality predictions when more than one translation is provided.
1.1 Relevance to QTLaunchPad

Deliverable D2.1.2 is intended to extend the basic framework presented in D2.1.1 for use in the QTLaunchPad project and beyond. In particular, the framework will be used to generate predictions within a translation ranking application in D2.1.3, and for the applications addressed in Tasks 2.2 which aim at improving MT systems by: MT system selection and self-training of MT systems. The framework will also be used within the shared task on quality assessment to be organised as part of WP5.

1.1.1 WP2 description

The aim of Work Package 2 is to provide methods, software and reference results for quality estimation of machine translation, as well as for the use of quality predictions in practical applications. More specifically, the objectives of this WP are:

1. Investigate and expand current QE methods for dissemination purposes to distinguish among at least three types of translations: (i) translations that are good enough to be left untouched by human post-editors (but possibly still revised); (ii) translations which require further effort (post-editing) to be published; and (iii) translations that should better be discarded, as they require more effort from human translators to correct them than what is involved in manual translation from scratch.

2. Use QE in a novel way to guide the development of translation systems: in contrast to the usual development cycles, where the goal is to improve the global quality of the MT system as a whole, the aim is to expand the set of translations which are good enough for dissemination.

1.1.2 Relation to other workpackages

An advanced framework for quality estimation such as QUEST can be customised to estimate specific aspects of quality mostly through the use of different features and annotation labels. To a lesser extent, learning algorithms can also support this customisation: standard classification and regression algorithms can be used to predict discrete or continuous absolute quality scores depending on the type of annotation, while ranking algorithms can be used for predictions on relative quality.

This design facilitates the very important connection between the methods and software developed in WP2 and WP1: Quality Barriers and Quality Metrics. On the one hand, the multidimensional quality metrics (MQM) designed in WP1 have helped:

- Identify relevant features for QE, e.g. features focusing on specific types of errors common in translations, such as inconsistent use of terminology. During the development of this new version of QUEST, the core issue types from MQM have been
studied with the aim of identifying gaps in the list of existing features that could be
covered by new features to be added to the framework (Section 2.2).

- Define appropriate quality labels for QE, i.e., from the estimation of a small number
  of standard quality categories (e.g. bad, medium, high quality) to more fine-grained
categories, and whether the global quality metric defined in WP1 can be directly es-
timated. Preliminary experiments investigating this matter are presented in D2.1.3.
These will be further developed as part of the shared task in WP5.

- Prepare corpora that can be used to train and test QE models that represent specific
  quality requirements relevant in the context of QTLaunchPad. Preliminary exper-
iments with existing, small scale corpora are presented in D2.1.3. These will be
further developed as part of the shared task in WP5.

On the other hand, QE will help quality metrics designed in WP1 by providing:

- A method to sample large datasets for manual evaluation/quality assurance such
  that one can focus on instances with specific ranges of quality (e.g. top or bottom
  10%), and

- Methods to partially automate manual error annotation by providing algorithms to
  automatically measure or approximate linguistic characteristics relevant to certain
issue types in MQM, e.g. counting incorrect capitalisation or number agreement
issues, both of which affect the fluency of the translations. These algorithms are
effectively the same as those used for feature extraction: in practice each feature
value can be used on their own as qualitative indicators, or as features combined
through machine learning algorithms to predict a more quantitative global quality
score (possibly such as the main global quality metric in WP1). Section (Section
2.2) presents a list of issue types for which features already exist or are in the
process of being implemented.

Finally, QUeST as a software package will be essential to the shared task to be or-
organised as part of WP5 Test of Infrastructure Setup in Shared Task. This shared task
will involve having quality estimation systems submitted by participants external to QT-
LaunchPad, who will be competing in the task of predicting overall quality scores, and
word-level error annotation. QUeST and its pre-computed feature sets will be released
as the “baseline system” for the shared-task participants, and will serve as a basis upon
which more advanced systems can be developed.

Main dependencies involving this deliverable   As mentioned above, one of the goals
for WP2 is to apply the quality estimation software developed QUeST to build models
from corpora annotated with issue types in MQM (WP1). However, since the corpus
produced so far (TQ Error Corpus, D1.2.1) is not large enough for quality estimation and
while larger corpora are under development in the project (for the shared task in WP5), we have started experimenting with external data annotated with similar issue types as those offered by MQM by one of our industry collaborators, namely WeLocalize, as we describe in D2.1.3.
Chapter 2

QuEst - features and architecture

2.1 Features

The list of MT system- and language independent features used in D2.1.1 – the so-called Black-box features can be found on http://www.QUEST.dcs.shef.ac.uk/quest_files/features_blackbox. The new features added in the second phase can be divided in two groups:

Glass-box features These are features extracted from the internal information of the machine translation system used to produce the translations. The current implementation assumes the output of the Moses phrase-based statistical machine translation system, since it is one of the most widely adopted systems. Several sources of internal information from Moses are used: n-best list with up to n translations for each source sentence, each model component value (translation model scores, language model score, distortion scores, etc.), final model score, phrase and word-alignment information, decoder’s word graph, and verbose information from the decoder regarding the search space (number of nodes in search graph, number of nodes pruned, etc.). From these resources, 45 features are extracted, such as:

- log probability score of the hypothesis (Moses’ global score)
- size of n-best list (number of hypotheses generated)
- using n-best for building a language model: sentence 1/3-gram probabilities and perplexities
- each of the translation and language model component values
- maximum size of the phrases (number of words) in the translation
- proportion of unknown/untranslated words
• average relative frequency of words in the top translation across the \( n \)-best list
• average size of hypotheses in the n-best list
• n-best list density (vocabulary size / average sentence length)
• fertility of the words in the source sentence compared to the n-best list in terms of words (vocabulary size / source sentence length)
• edit distance of the current hypothesis to the centre hypothesis (closest hypothesis to all others in the n-best list)
• total number of hypotheses in search graph
• number/percentage of discarded/pruned/recombined search graph nodes.

The full list of glass-box features can be found on http://www.quest.dcs.shef.ac.uk/quest_files/features_glassbox.

**Advanced features** These are features that require linguistic processors or non-trivial pre-built models from data to be extracted. They are mostly of the black-box type, in the sense that they do not require information from the MT system that generated the translations, with a few exceptions of features that rely on word-alignment information. These are indicated in the list below by the flag (GB). Most of the advanced features are language-independent, in the sense that they can be applied for any language pair, even though they might require specific linguistic processors for those language pairs. For example, in order to extract the content words in a given language, a part-of-speech (POS) tagger for that language is required, as well as a list of content word tags, but these resources can be found for many languages. Truly language-dependent features are those that represent linguistic phenomena that are only relevant for certain language pairs. Features of this type are indicated in the list below by the flag (LD). The list below summarises the advanced features. For a complete list with the 52 features available to date, we refer the reader to the black-box and glass-box lists previously mentioned, as they now include the advanced features.

• percentage of content words in the source sentence/target sentence
• ratio of percentage of content words in the source and target sentences
• language model probability/perplexity of POS tags of target sentence
• percentage of nouns/verbs in the source/target sentence
• ratio of percentage of nouns/verbs/pronouns in the source and target sentences
• number of dependency relations with aligned constituents normalised by the total number of dependencies, with or without taking the order of the constituents into account (GB)

• number of dependency relations with possibly aligned constituents (using Giza’s lexical table with \( p > 0.1 \)) normalised by the total number of dependencies

• absolute difference between the depth of the syntactic trees of the source and target sentences

• number of prepositional phrases in the source/target sentence

• (normalised) absolute difference between the number of prepositional/nominal/verbal/adjectival/adverbial/conjunctive phrases in the source and target sentences

• source/target probabilistic context-free grammar (PCFG) parse log-likelihood [Avramidis, 2012]

• source/target PCFG average confidence of all possible parses in n-best list of parse trees for the sentence [Avramidis, 2012]

• source/target PCFG confidence of best parse [Avramidis, 2012]

• count of possible source/target PCFG parses [Avramidis, 2012]

• Kullback-Leibler/Jensen-Shannon divergence of source and target topic distributions using LDA (Latent Dirichlet Allocation) for topic modelling [Rubino and Specia, 2013]

• source/target sentence intra-lingual triggers [Langlois et al., 2012]

• source-target sentence inter-lingual mutual information [Langlois et al., 2012]

• percentage of incorrectly translated possessive pronouns (for Arabic-English only) (LD) [Specia et al., 2011]

• percentage of incorrectly translated direct object personal pronouns (for Arabic-English only) (LD) [Specia et al., 2011]

• readability features:
  – LIX readability score\(^1\) for source/target sentence
  – Average number of characters in source and target words and their ratios

\(^1\)http://en.wikipedia.org/wiki/LIX
• information retrieval (IR) features that measure the closeness of the source sentences or their translations to the parallel SMT training data, aimed at predicting the difficulty of translating each sentence or finding their translations [Bicici et al., 2013, Bicici, 2013, Shah et al., 2013a]. For every sentence, we retrieve the top 5 instances from the SMT training corpus and compute:
  – IR score over the source sentence or its translation
  – BLEU scores over source sentence or its translation
  – $F_1$ scores over source sentence or its translation [Bicici, 2011]

• Combinatory Categorial Grammar (CCG)-based features [Almaghout and Specia, 2013] (GB):
  – minimum number of CCG constituents which span the target sentence
  – number of maximal phrases in the translation output
  – percentage of supertag/category argument mismatches
  – 5-gram supertag LM probability/perplexity

We include these new features in our benchmarking experiments in D2.1.3.

2.2 QuEST features and MQM

In what follows we list the core issue types in MQM shown in Figure 2.1, with an indication on whether they are already at least partially covered by features in QuEST or their computation cannot be automated. We note that most features provide only an approximation to issue types, in the sense that they would not result in the same figures as those that would be obtained by a human annotator, but they attempt to capture the same types of error.

Accuracy

• Terminology: Normative terminology infringed. This issue is not directly covered by current approaches to quality estimation. However, as a proxy to it, both monolingual (target) and bilingual terminology lists could be used for simple checks, such as whether all content words (or nouns) in the translation belong to the terminology list.

2A detailed description of the issue types can be found on http://www.qt21.eu/launchpad/content/list-mqm-issue-types.
• **Mistranslation**: Incorrect word translation chosen (overly literal, false friend, should not have been translated, entity, date/time/number, unit conversion). This issue cannot be easily automated, apart from some mechanical checks on date/time/number format). Essentially if it could be automated, it would be possible to ensure the generation of the correct translation.

• **Omission**: Translation for source word is missing. Certain existing features approximate this issue type, e.g. simple source versus target segment word counts, counts of words with certain POS tags in both source and target segments, and language models of the target language, which can detect unusual constructions due – among other things - omissions.

• **Addition**: Word that is not in the source segment is added to the translation. Existing features approximate this issue as in the case of “omission”.

• **Untranslated**: A source word is left untranslated in the translation. This issue is currently approximated by out-of-vocabulary features based on language model of the target language.

**Fluency**

• **Register/style**: Incorrect use of words due to variants/slang, company style or style guide. This issue is not directly covered by existing approaches, but it is approximated by the target language model features, as long as this model is trained on documents with the correct register/style.

• **Spelling**: Incorrect word spelling due to capitalisation or diacritics. This issue is also approximated by language model features, which are trained on truecased models.
• **Typography**: Incorrect use of punctuation, unpaired quote marks or brackets. These issues are captured by a number of features, such as those checking for missing closing brackets or quotation symbols in the target segment, and those contrasting the percentage of different punctuation symbols in the source and target languages.

• **Grammar**: The several grammar-related issues (morphology, part of speech, agreement, word order, function words, tense/mood/aspect) are captured partly by target language model features, and partly by advanced syntactic features based on probabilistic context-free grammars, dependency structures and categorical combinatory grammar and others [Felice and Specia, 2012].

• **Unintelligible**: The translation is too bad to be analysed. This issue is approximated by language model features of the target language.

### 2.3 Source code

We made available the following versions of the code on [http://www.quest.dcs.shef.ac.uk](http://www.quest.dcs.shef.ac.uk):

- A **stable**, complete version of the source code.
- A **vanilla** version of the source code which is easier to run (and re-build), as it relies on fewer pre-processing resources/tools. Toy resources for en-es are also included in this version. It extracts up to 50 features.
- A **client-server** version which extracts the baseline features only, but is optimised for sentence-by-sentence processing (see Section 3.1).
- The latest **development** version of the complete code from GitHub: [https://github.com/lspecia/quest](https://github.com/lspecia/quest)

An **installation script** that will download the stable version of the source code, a built up version (jar), and all necessary pre-processing resources/tools (parsers, etc.) is also available for download from [http://www.quest.dcs.shef.ac.uk](http://www.quest.dcs.shef.ac.uk).

### 2.4 Setting up

Once downloaded, the folder with the code contains all files required for running or building the application:

- **src**: java source files
• lib: jar files, including the external jars required by QUEST
• dist: javadoc documentation
• lang-resources: example of language resources required to extract features
• config: configuration files
• input: example of input training files (source and target sentences, plus quality labels)
• output: example of extracted feature values

2.5 The feature extractor

The class that performs feature extraction is shef.mt.FeatureExtractor. It handles the extraction of glass-box and/or black-box features from a pair of source-target input files and a set of additional resources specified as input parameters. Whilst the command line parameters relate to the current set of input files, FeatureExtractor also relies on a set of project-specific parameters, such as the location of resources. These are defined in a configuration file in which resources are listed as pairs of key=value entries. By default, if no configuration file is specified in the input, the application will search for a default config.properties file in the current working folder (i.e., the folder where the application is launched from). This default file is provided with the distribution.

Another input parameter required is the XML feature configuration file, which gives the identifiers of the features that should be extracted by the system. Unless a feature is present in this feature configuration file it will not be extracted by the system. Examples of such files for all features, black-box, glass-box, and a subset of 17 “baseline” features are provided with the distribution.

2.6 Running the feature extractor

The following command triggers the features extractor:

```
FeatureExtractor -input <source file> <target file>
-lang <source language> <target language> -config
<configuration file> -mode [gb|bb|all] -gb [list of GB resources]
```

Where the arguments are:

- -input <source file> <target file> (required): the input source and target text files with sentences to extract features from
CHAPTER 2. QUEST - FEATURES AND ARCHITECTURE

• -lang <source language> <target language>: source and target languages of the files above

• -config <configuration file>: file with the paths to the input/output, XML-feature files, pre-processing tools/scripts and language resources

• -mode <gb|bb|all>: a choice between glass-box, black-box or both types of features

• -gb [list of files]: input files required for computing the glass-box features. The options depend on the MT system used. For Moses, three files are required: a file with the n-best list for each target sentence, a file with a verbose output from the decoder (for phrase segmentation, model scores, etc.), and a file with search graph information in XML (examples are provided with the distribution).

2.7 Major packages and classes

Here we list the important packages and classes. We refer the reader to QUEST documentation for a comprehensive list of modules.

• shef.mt.enes: This package contains the main feature extractor classes

• shef.mt.features.impl.bb: This package contains the implementations of black-box features

• shef.mt.features.impl.gb: This package contains the implementations of glass-box features

• shef.mt.features.util: This package contains various utilities to handle information in a sentence and/or phrase

• shef.mt.tools: This package contains wrappers for various pre-processing tools and Processor classes for interpreting the output of the tools

• shef.mt.tools.stf: This package contains classes that provide access to the Stanford parser output

• shef.mt.util: This package contains a set of utility classes that are used throughout the project, as well as some independent scripts used for various data preparation tasks

• shef.mt.xmlwrap: This package contains XML wrappers to process the output of SMT systems for glass-box features.
The most important classes are as follows:

- **FeatureExtractor**: FeatureExtractor extracts glass-box and/or black-box features from a pair of source-target input files and a set of additional resources specified as input parameters.

- **Feature**: Feature is an abstract class which models a feature. Typically, a Feature consist of a value, a procedure for calculating the value and a set of dependencies, i.e., resources that need to be available in order to be able to compute the feature value.

- **FeatureXXXX**: These classes extend Feature to provide their own method for computing a specific feature.

- **Sentence**: Models a sentence as a span of text containing multiple types of information produced by pre-processing tools, and direct access to the sentence tokens, n-grams, phrases. It also allows any tool to add information related to the sentence via the setValue() method.

- **MTOutputProcessor**: Receives as input an XML file containing source sentences and lists of n-best translation with various attributes and reads it into Sentence objects.

- **ResourceProcessor**: Abstract class that is the basis for all classes that process output files from pre-processing tools.

- **ResourceManager**: This class contains information about resources for a particular feature.

- **LanguageModel**: LanguageModel stores information about the content of a language model file. It provides access to information such as the frequency of n-grams, and the cut-off points for various n-gram frequencies necessary for certain features.

- **Tokenizer**: A wrapper around the Moses tokenizer.

### 2.8 Architecture

A hierarchy of a few of the most important classes is shown in Figure 2.2. Two principles underpin this design choice:

- pre-processing must be separated from the computation of features, and
CHAPTER 2. QUEST - FEATURES AND ARCHITECTURE

(a) The Feature class

(b) A particular feature extends the Feature class and is associated with the Sentence class

(c) An abstract Resource class acts as a wrapper for external processes

(d) ResourceProcessor reads the output of a tool and stores it in a Sentence object

Figure 2.2: Class hierarchy for most important classes.
• feature implementation must be modular in the sense that one is able to add features without having to modify other parts of the code.

A typical feature will need a set of tools or resources (for pre-processing), with associated classes for processing the output of these tools. A Resource is usually a wrapper around an external process (such as a part-of-speech tagger or parser), but it can also be a brand new fully implemented pre-processing tool. The only requirement for a tool is to extend the abstract class shef.mt.tools.Resource. The implementation of a tool/resource wrapper depends on the specific requirements of that particular tool and on the developers preferences. Typically, it will take as input a file and a path to the external process it needs to run, as well as any additional parameters the external process requires. It will call the external process, capture its output and write it to a file.

The interpretation of the tool’s output is delegated to a subclass of shef.mt.tools.ResourceProcessor associated with that particular Resource. A ResourceProcessor typically:

• Contains a function that initialises the associated Resource. Each Resource may require a different set of parameters upon initialisation. ResourceProcessor handles this by passing the necessary parameters from the configuration file to the respective function of the Resource.

• Registers itself with the ResourceManager in order to signal the fact that it has successfully managed to initialise itself and can pass information to be used by features. This registration should be done by calling ResourceManager.registerResource(String resourceName). resourceName is an arbitrary string, unique among all other Resources. If a feature requires this particular Resource for its computation, it needs to specify it as a requirement (see Section 2.9).

• Reads in the output of a Resource sentence by sentence, retrieves some information related to that sentence and stores it in a Sentence object. The processing of a sentence is done in the processNextSentence(Sentence sentence) function which all ResourceProcessor-derived classes must implement. The information it retrieves depends on the requirements of the application. For example, shef.mt.tools.POSProcessor, which analyses the output of the TreeTagger, retrieves the number of nouns, verbs, pronouns and content words, since these are required by certain currently implemented features, but it can be easily extended to retrieve, for example, adjectives, or full lists of nouns instead of counts.

A Sentence is an intermediate object that is, on the one hand, used by ResourceProcessor to store information and, on the other hand, by Feature to access this information. The implementation of the Sentence class already contains access methods to some of the most commonly used sentence features, such as the text it
spans, its tokens, its n-grams, its phrases and its n-best translations (for glass-box features). For a full list of fields and methods, see the associated javadoc. Any other sentence information is stored in a HashMap with keys of type String and values of generic type Object. A pre-processing tool can store any value in the HashMap by calling setValue(String key, Object value) on the currently processed Sentence object. This allows tools to store both simple values (integer, float) as well as more complex ones (for example, the ResourceProcessor).

2.9 Adding a new feature

In order to add a new feature, one has to implement a class that extends shef.mt.features.impl.Feature. A Feature will typically have an index and a description which should be set in the constructor. The description is optional, whilst the index is used in selecting and ordering the features at runtime, therefore it needs to be set. The only function a new Feature class has to implement is run(Sentence source, Sentence target). This will perform some computation over the source and/or target sentence and set the return value of the feature by calling setValue(float value). If the computation of the feature value relies on some pre-processing tools or resources, the constructor can add these resources or tools in order to ensure that the feature will not run if the required files are not present. This is done by a call to addResource(String resourceName), where resourceName has to match the name of the resource registered by the particular tool this feature depends on.
Chapter 3

**QUEst for lay-users**

QUEst may not be directly usable by all types of end-users. The tasks of installing and configuring the toolkit, obtaining the necessary resources, and building new models from data require some technical knowledge of natural language processing and machine learning, in addition to software installation.

Therefore, here we provide a version of QUEst which is meant to make it more accessible to non-expert users, as well as more efficient. In particular, we provide a client-server architecture which allows users to access pre-built models and resources remotely through XML-RPC requests, and which is optimised for speed by keeping resources in memory (Section 3.1); a Web interface where users can upload files with source and translation segments, with the possibility of getting translations from Bing Translator (Section 3.2); and a ranking mechanism that provides a ranked list of multiple the options of translations given for each source segment (Section ??).

### 3.1 Client-server architecture

The adaptation of QUEst to the online scenario has required an upgrade of different components of the previous version. The main goal of such changes has been to: i) allow the processing of one sentence pair at the time, ii) speed up the feature extraction, and iii) make QUEst easily accessible remotely.

**Client-server compatible design**  The previous version of QUEst was designed to process large text files containing the source and target sentences. This required the following steps in a unique sequence at run time, before any feature could start being extracted:

1. Loading in memory of the main resources needed to extract features, such as a list of the n-grams from the MT training corpus, source and target language models, and bilingual dictionaries.
2. Pre-processing of the whole source and target files extracting information such as part-of-speech tagging and language model probabilities.

3. Filtering of the main resources according to the source and target sentences to reduce the computational effort while extracting features.

Only when these steps were completed for the entire input files, the extraction of features for each sentence pair could start. In the client-server version of QUEST, this structure has been refactored for efficiency when processing multiple requests from users, and to better fit the demand of the web interface, processing sentences one at a time. Changes were necessary in the first two steps, with the last step being removed (dealing with on the fly requests for predictions for a given sentence pair does not allow the filtering of resources). This modification has made the resources stored in memory completely independent of the sentences to be processed. The loading in memory of the resources is now part of a global initialisation step. This increased the amount of memory required to store all resources, but it made QUEST more suitable for being embedded in a client-server framework.

**Language Model Servers** Some of the most effective features for QE require the computation of sentence level language model (LM) probabilities or perplexities. In general, effective LMs are obtained from large corpora, and can thus be very large files. This implies that starting the LMs during the QUEST computation and having them running on the same machine can be problematic. To cope with these problems, each LM has been encapsulated in a server which can run independently from QUEST in different machines. Within this version of QUEST, the computation of LM scores is done in a client that, given a sentence, queries the LM server to get the relevant scores. In the initialisation step, the connections with the LM servers are established and a fake query is used to force the initialisation of the LMs.

**Client-Server Framework** This new version of QUEST has been embedded in a server that allows connection to the feature extractor from different clients located in various machines. This wrapper links QUEST to external machines using sockets, while it is linked to QUEST using standard input and output streams. This architecture is outlined in Figure 3.1.

**Offline Pre-processing** Launching external software within QUEST, such as tokenization and true-casing, is time consuming. To mitigate the effect, QUEST now can easily deal with already pre-processed source and target sentences.

These modifications have made QUEST slimmer and easier to be used. In particular, they have speeded up the feature extraction process allowing its use in an online scenario or as a part of a Web interface, as we discuss in the next Section.
3.2 A Web interface

In order to facilitate the use of QUEST by non-experts such as translators or users of online MT systems, we have developed a Web interface that allows users to access the tool remotely from a Web browser, without the need to understand the internal functioning of framework, nor to install/configure the tool or build models. The interface can be accessed at http://www.quest.dcs.shef.ac.uk/QuEstClient_v1/test.php. It offers the following functionalities:

- **Features**: Values for individual features describing the source and translation sentence, e.g. source and target length, LM scores, average translation ambiguity level of source words, etc.

- **Predictions**: An estimated quality score for the translated sentence given the source sentence, produced using Support Vector Regression (SVR) and pre-build models for specific language pairs.

- **Ranking**: The ranking of multiple translations submitted by the user for a given source sentence. This is done based on SVR quality predictions on each source-target sentence pair independently.
The Web interface is developed using PHP and XML-RPC for communication across the main QUEST server and resource servers. For the convenience of users, we have also integrated the free Bing translation API to this Web interface.¹

The pipeline of the framework accessed via its Web interface is the following:

- User inputs a file containing source sentences only or tab separated source sentences and their translation(s) – as many translations as desired.
- User selects the language pair and text type (domain, etc.).
- File is uploaded to the Web server, and read line by line (sentence by sentence).
- If the input file contains only source sentences, a request is sent to Bing’s API with the selected target language.
- Based on the choices (language pair, text type) selected by the user, an instance of QUEST with the appropriate prediction model and resources is triggered.
- QUEST extracts the features by calling the Feature Extractor module. LMs, other resources and prediction models are already loaded into memory by a fake call.
- QUEST generates a prediction by applying the prediction model for that language pair.
- If the input file contains multiple translations for the same source sentence, QUEST ranks these translations.

These functionalities require prediction models previously trained offline for each language pair of interest. Options to build models from examples of translations, quality scores and language resources could be added to the interface in the future, but users will require some advanced knowledge of QUEST to use them.

### 3.3 Performance considerations

In order to assess the runtime performance of QUEST, and in particular, of the online, client-server version presented in this Chapter as compared to the previous, offline version, we perform experiments with two language pair datasets: French-English and German-English, in two different tasks: absolute prediction and ranking of up to five translation options. The ranking experiments are performed by simply taking the candidate translation with the highest score for each source segment. More elaborate ranking strategies are described in the benchmarking experiments in D2.1.3, where we contrast approaches for

¹Please note that the free version only allows 2,000,000 characters to be translated per month.
CHAPTER 3. QUEST FOR LAY-USERS

Table 3.1: Absolute score prediction: datasets and results for Fr-En.

<table>
<thead>
<tr>
<th>Method</th>
<th># Training</th>
<th># Test</th>
<th>MAE</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>1,881</td>
<td>9,000</td>
<td>0.151</td>
<td>0.201</td>
</tr>
<tr>
<td>Prediction</td>
<td>1,881</td>
<td>9,000</td>
<td>0.129</td>
<td>0.171</td>
</tr>
</tbody>
</table>

Table 3.2: Ranking of alternative translations: datasets and results for De-En.

<table>
<thead>
<tr>
<th>Method</th>
<th># Training</th>
<th># Test</th>
<th>Kendall’s $T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>7,098</td>
<td>365</td>
<td>0.04</td>
</tr>
<tr>
<td>Prediction</td>
<td>7,098</td>
<td>365</td>
<td>0.16</td>
</tr>
</tbody>
</table>

ranking and discuss their suitability depending on the training data available to build the ranking models.

The data used and results for these tasks are given in Tables 3.1 and 3.2. For both tasks, a set of 17 well established baseline features was used. These correspond to those used by the baseline system in WMT12-13-14, and also the baseline features we use in our benchmarking experiments in D2.1.1 and D2.1.3. The language models and other resources were built using standard tools provided with QUÆST, such as SRILM and GIZA++. SVR with radial basis function (RBF) kernel was used as learning algorithm, since it has been shown to perform well in previous work. The optimisation of parameters was done using grid search. In both cases, we show a comparison of our models (Prediction) against simple but strong baselines: outputting the average value of training instances for all test instances (Mean) for the absolute scoring task, and randomly picking one of the candidates in the ranking task (Random).

Both datasets are freely available. The French-English dataset is described in [Potet et al., 2012]. It has 10,881 source sentences and their MT output and post-editions. We measure and estimate HTER scores between the MT and its post-edition. The first 1,881 sentences were used for training, and the rest for test. Performance was measured in terms of Mean Absolute Error (MAE) and Root Mean Square Error (RMSE).

The German-English dataset is provided by the WMT13 shared task on QE, with the official training/test splits used. It has up to five alternative machine translations produced by different MT systems for each source sentence, which were ranked for quality by humans as part of the translation task in WMT08-WMT12. A baseline prediction model was trained using the rankings provided as absolute scores. This model was applied to each of the five alternative sentences and the predicted scores were used for ranking them. Performance was measured by comparing QUÆST predictions to rankings performed by

---

2The list of features can be found on [http://www.quest.dcs.shef.ac.uk/quest_files/features_blackbox_baseline_17](http://www.quest.dcs.shef.ac.uk/quest_files/features_blackbox_baseline_17).

### Table 3.3: Sizes of resources and cumulative response time in minutes for QuEst offline vs QuEst online for all sentence pairs (and per sentence pair) on each of the two test sets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Offline QuEst</th>
<th>Online QuEst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fr-En</td>
<td>2,021 (0.224)</td>
<td>343 (0.038)</td>
</tr>
<tr>
<td>De-En</td>
<td>110 (0.300)</td>
<td>22 (0.060)</td>
</tr>
</tbody>
</table>

The resources used to extract the features are source and target 3-gram language models (SRC-LM and TGT-LM), part-of-speech tag source language model (POS-LM), source-target Giza++ lexical table (GIZA), and raw counts of 1-3 grams in a corpus of the source language (NGRAM).

### Table 3.4: Response time in seconds – per sentence – with an online interface of various models (FE = Feature Extractor, PR = Prediction).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>BING</th>
<th>FE</th>
<th>PR</th>
<th>FE + PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fr-En</td>
<td>1.1</td>
<td>1.12</td>
<td>1.17</td>
<td>2.29</td>
</tr>
<tr>
<td>De-En</td>
<td>1.1</td>
<td>2.10</td>
<td>1.51</td>
<td>3.61</td>
</tr>
</tbody>
</table>

humans in terms of Kendall’s $\tau$ correlation.

Table 3.3 gives a comparison between online QuEst and its previous, offline version in term of cumulative response time for all sentence pairs in each our two test sets along with the sizes of the language resources used to extract features for each of them. These figures include the time for loading models and sentences and input/output processing. Offline QuEst needs to do this for each sentence pair, while the new version loads all models only once, clearly showing better performance.

We have also tested the response time of these pre-built models for each module in online QuEst, as shown in Table 3.4. These figures refer to running QuEst at a local host on a single core of machine Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz with 190GB of RAM. The response time for remote requests will depend upon the network speed. It is important to note the difference between response time for each of the dataset: The use of larger resources to extract features yields overall slower response time.
Chapter 4

Remarks

The source code for the framework, as well as datasets used for the benchmark, and pre-processing resources can be downloaded from http://www.quest.dcs.shef.ac.uk/. Various versions are available from this website:

- A full version, currently with 191 features, but which gets constantly updated when new features or developments are added after these are tested by USFD.

- A vanilla version, which consists of a simpler version of the above where features that require any complex pre-processing are removed so that the code can be quickly built locally. The vanilla version has 50 features.

- A client-server version with 17 baseline features.

- A Web interface for remote access to obtain features and predictions for pre-built models.

In addition, the code is also open for contributions by any collaborators from a github repository: https://github.com/lspecia/quest.

The license for the Java code, Python and shell scripts is BSD,\(^1\) a permissive license with no restrictions on the use or extensions of the software for any purposes, including commercial applications. For pre-existing code and resources, e.g. scikit-learn, GPy, GIZA++, SRILM and IRSTLM language modeling toolkits, their licenses apply, but features relying on these resources can be discarded if necessary.

Acknowledgements The client-server architecture design was developed in collaboration with Marco Turchi from the MATECAT project (EU ICT-2011.4.2-287688). MATECAT uses Q\textsc{u}E\textsc{st} as part of their CAT tools and has contributed with improvements to the client-server version of the code. We also acknowledge contributions

\(^{1}\)http://en.wikipedia.org/wiki/BSD_licenses
from participants in the QuEst Project (http://www.dcs.shef.ac.uk/~lucia/projects/quest.html), funded by the Pascal2 Network of Excellence as a Harvest project. These participants visited Sheffield University for 5-10 days between October and December 2012 to implement some of the quality estimation features now available in this version of the QUEST framework. After extensive tests, these features have now been merged into QUET by USFD and included in the benchmarking presented in deliverable D2.1.3.
Bibliography


