PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 100 OCTOBER 2013 19–30

QuEst — Design, Implementation and Extensions of a Framework for Machine Translation Quality Estimation

Kashif Shah[a], Eleftherios Avramidis[b], Ergun Biçici[c], Lucia Specia[a]

[a] University of Sheffield
[b] German Research Center for Artificial Intelligence
[c] Centre for Next Generation Localization, Dublin City University

Abstract

In this paper we present QuEst, an open source framework for machine translation quality estimation. The framework includes a feature extraction component and a machine learning component. We describe the architecture of the system and its use, focusing on the feature extraction component and on how to add new feature extractors. We also include experiments with features and learning algorithms available in the framework using the dataset of the WMT13 Quality Estimation shared task.

1. Introduction

Quality Estimation (QE) is aimed at predicting a quality score for a machine translated segment, in our case, a sentence. The general approach is to extract a number of features from source and target sentences, and possibly external resources and information from the Machine Translation (MT) system for a dataset labelled for quality, and use standard machine learning algorithms to build a model that can be applied to any number of unseen translations. Given its independence from reference translations, QE has a number of applications, for example filtering out low quality translations from human post-editing.

Most of current research focuses on designing feature extractors to capture different aspects of quality that are relevant to a given task or application. While simple features such as counts of tokens and language model scores can be easily extracted, feature engineering for more advanced information can be very labour-intensive. Dif-

Corresponding author: Kashif.Shah@sheffield.ac.uk
Cite as: Kashif Shah, Eleftherios Avramidis, Ergun Biçici, Lucia Specia. QuEst — Design, Implementation and Extensions of a Framework for Machine Translation Quality Estimation. The Prague Bulletin of Mathematical Linguistics No. 100, 2013, pp. 19–30. doi: 10.2478/pralin-2013-0008.

ferent language pairs or optimisation against specific quality scores (e.g., post-editing time versus translation adequacy) can benefit from different feature sets.

QuEst is a framework for quality estimation that provides a wide range of feature extractors from source and translation texts and external resources and tools (Section 2). These range from simple, language-independent features, to advanced, linguistically motivated features. They include features that rely on information from the MT system that generated the translations, and features that are oblivious to the way translations were produced, and also features that only consider the source and/or target sides of the dataset (Section 2.1). QuEst also incorporates wrappers for a well-known machine learning toolkit, `scikit-learn`[1] and for additional algorithms (Section 2.2).

This paper is aimed at both users interested in experimenting with existing features and algorithms and developers interested in extending the framework to incorporate new features (Section 3). For the former, QuEst provides a practical platform for quality estimation, freeing researchers from feature engineering, and facilitating work on the learning aspect of the problem, and on ways of using quality predictions in novel extrinsic tasks, such as self-training of statistical machine translation systems. For the latter, QuEst provides the infrastructure and the basis for the creation of new features, which may also reuse resources or pre-processing techniques already available in the framework, such as syntactic parsers, and which can be quickly benchmarked against existing features.

2. Overview of the QuEst Framework

QuEst consists of two main modules: a feature extraction module and a machine learning module. It is a collaborative project, with contributions from a number of researchers.[2] 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 source MT training corpus and language models of source and target languages are necessary for certain 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. It also uses `GPy`, a Python toolkit for Gaussian Processes regression, which showed good performance in previous work (Shah et al., 2013).

[1] `http://scikit-learn.org/`

[2] See `http://www.quest.dcs.shef.ac.uk/` for a list of collaborators.

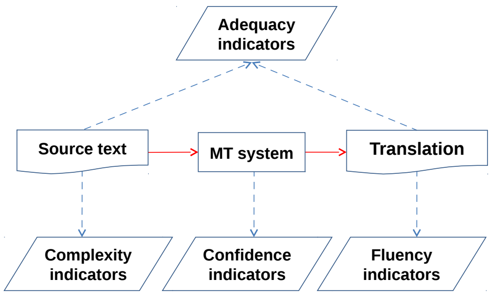


Figure 1: Families of features in QuEst.

2.1. Feature Sets

In Figure 1 we show the families of features that can be extracted in QuEst. Although the text unit for which features are extracted can be of any length, most features are more suitable for sentences. Therefore, a "segment" here denotes a sentence. Most of these features have been designed with Statistical MT (SMT) systems in mind, although many do not explore any internal information from the actual SMT system. Further work needs to be done to test these features for rule-based and other types of MT systems, and to design features that might be more appropriate for those.

From the source segments QuEst can extract features that attempt to quantify the **complexity** or **translatability** of those segments, or how unexpected they are given what is known to the MT system. From the comparison between the source and target segments, QuEst can extract **adequacy** features, which attempt to measure whether the structure and meaning of the source are preserved in the translation. Information from the SMT system used to produce the translations can provide an indication of the **confidence** of the MT system in the translations. They are called "glass-box" features (GB) to distinguish them from MT system-independent, "black-box" features (BB). To extract these features, QuEst assumes the output of Moses-like SMT systems, taking into account word- and phrase-alignment information, a dump of the decoder's standard output (search graph information), global model score and feature values, n-best lists, etc. For other SMT systems, it can also take an XML file with relevant information. From the translated segments QuEst can extract features that attempt to measure the **fluency** of such translations.

The most recent version of the framework includes a number of previously under-explored features that can rely on only the source (or target) side of the segments and on the source (or target) side of the parallel corpus used to train the SMT system. Information retrieval (IR) features measure the closeness of the QE source sentences and their translations to the parallel training data available to predict the difficulty of translating each sentence. These have been shown to work very well in recent work

(Biçici et al., 2013; Biçici, 2013). We use Lucene[3] to index the parallel training corpora and obtain a retrieval similarity score based on tf-idf. For each source sentence and its translation, we retrieve top 5 distinct training instances and calculate the following features:

- IR score for each training instance retrieved for the source sentence or its translation
- BLEU (Papineni et al., 2002) and F_1 (Biçici, 2011) scores over source or target sentences
- LIX readability score[4] for source and target sentences
- The average number of characters in source and target words and their ratios.

In Section 4 we provide experiments with these new features.

The complete list of features available is given as part of QuEst's documentation. At the current stage, the number of BB features varies from 80 to 143 depending on the language pair, while GB features go from 39 to 48 depending on the SMT system.

2.2. Machine Learning

QuEst provides a command-line interface module for the `scikit-learn` library implemented in Python. This module is completely independent from the feature extraction code. It reads the extracted feature sets to build and test QE models. The dependencies are the `scikit-learn` library and all its dependencies (such as NumPy and SciPy). The module can be configured to run different regression and classification algorithms, feature selection methods and grid search for hyper-parameter optimisation.

The pipeline with feature selection and hyper-parameter optimisation can be set using a configuration file. Currently, the module has an interface for Support Vector Regression (SVR), Support Vector Classification, and Lasso learning algorithms. They can be used in conjunction with the feature selection algorithms (Randomised Lasso and Randomised decision trees) and the grid search implementation of `scikit-learn` to fit an optimal model of a given dataset.

Additionally, QuEst includes Gaussian Process (GP) regression (Rasmussen and Williams, 2006) using the `GPy` toolkit.[5] GPs are an advanced machine learning framework incorporating Bayesian non-parametrics and kernel machines, and are widely regarded as state of the art for regression. Empirically we found its performance to be similar or superior to that of SVR for most datasets. In contrast to SVR, inference in GP regression can be expressed analytically and the model hyper-parameters optimised using gradient ascent, thus avoiding the need for costly grid search. This also makes the method very suitable for feature selection.

[3]`lucene.apache.org`

[4]`http://en.wikipedia.org/wiki/LIX`

[5]`https://github.com/SheffieldML/GPy`

3. Design and Implementation

3.1. Source Code

We made available three versions of the code, all available from `http://www.quest.dcs.shef.ac.uk`:

- 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.).
- A stable version of the above source code only (no linguistic processors).
- 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 only extracts up to 50 features.

In addition, the latest development version of the code can be accessed on GitHub.[6]

3.2. Setting Up

Once downloaded, the folder with the code contains all files required for running or building the application. It contains the following folders and resources:

- `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

3.3. 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

[6] `https://github.com/lspecia/quest`

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.

3.4. 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
- `-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, 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 of the decoder (for phrase segmentation, model scores, etc.), and a file with search graph information.

3.5. 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` and 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 sentences and lists of 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.
- `Pipeline`: Abstract class that sets the basis for handling the registration of the existing `ResourceProcessors` and defines their order.
- `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.

3.6. Developer's Guide

A hierarchy of a few of the most important classes is shown in Figure 2. There are two principles that underpin the design choice:

- pre-processing must be separated from the computation of features, and
- 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 application will contain 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 developer's 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`. As 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 it 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 3.7).
- 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`).

A `Pipeline` defines the order in which processors will be initialised and run. They are defined in the `shef.mt.pipelines` package. They allow more flexibility for the execution of pre-processors, when there are dependencies between each other. At the moment QuEst offers a default pipeline which contains the tools required for the "vanilla" version of the code and new `FeatureExtractors` have to register there. A more convenient solution would be a dynamic pipeline which automatically identifies the processors required by the enabled features and then initialises and runs only them. This functionality is currently under development in QuEst.

3.7. 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 should 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 resource_name)`, where `resource_name` has to match the name of the resource registered by the particular tool this feature depends on.

4. Benchmarking

In this section we briefly benchmark QuEst using the dataset of the main WMT13 shared task on QE (subtask 1.1) using all our features, and in particular the new source-based and IR features. The dataset contains English-Spanish sentence translations produced by an SMT system and judged for post-editing effort in [0,1] using TERp,[7] computed against a human post-edited version of the translations (i.e. HTER). $2,254$ sentences were used for training, while 500 were used for testing.

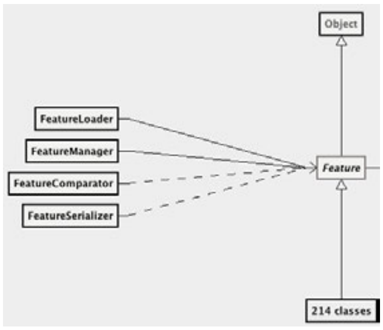
As learning algorithm we use SVR with radial basis function (RBF) kernel, which has been shown to perform very well in this task (Callison-Burch et al., 2012). The optimisation of parameters is done with grid search based on pre-set ranges of values as given in the code distribution.

For feature selection, we use Gaussian Processes. Feature selection with Gaussian Processes is done by fitting per-feature RBF widths. The RBF width denotes the importance of a feature, the narrower the RBF the more important a change in the feature value is to the model prediction. To avoid the need of a development set to optimise the number of selected features, we select the 17 top-ranked features (as in our baseline system) and then train a model with these features.

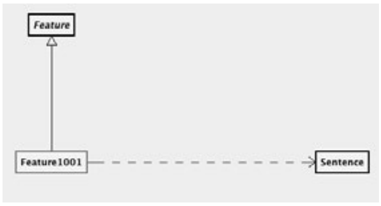
For given dataset we build the following systems with different feature sets:

- **BL**: 17 baseline features that have been shown to perform well across languages in previous work and were used as a baseline in the WMT12 QE task

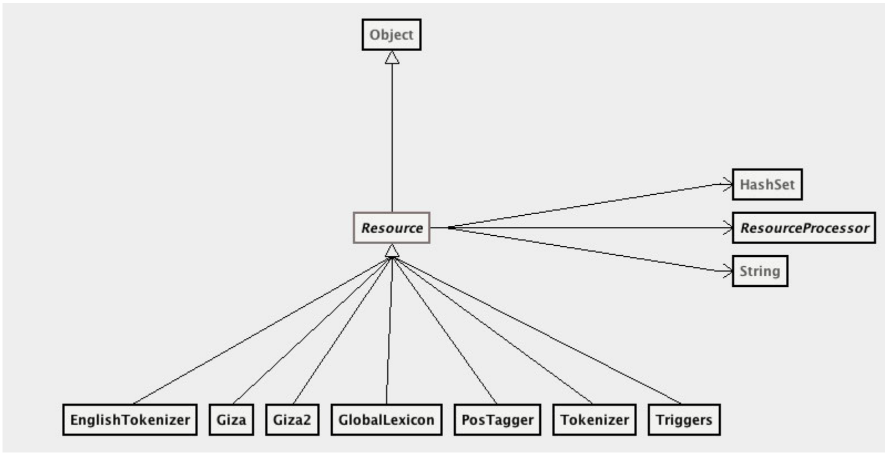
[7] `http://www.umiacs.umd.edu/~snover/terp/`



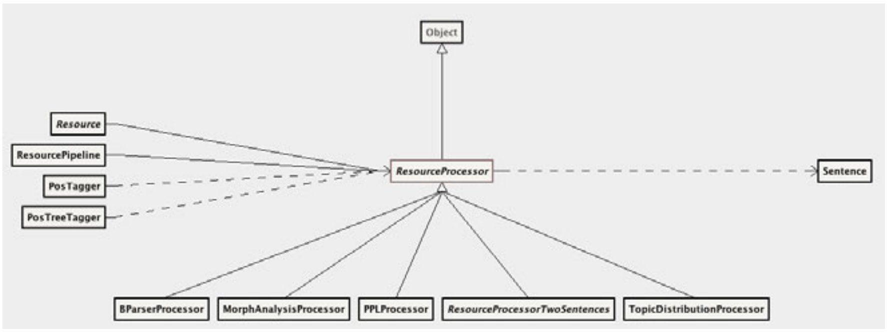
(a) The `Feature` class



(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: Class hierarchy for most important classes.

- **AF**: All features available from the latest stable version of QuEst, either black-box (BB) or glass-box (GB)
- **IR**: IR-related features recently integrated into QuEst (Section 2.1)
- **AF+IR**: All features available as above, plus recently added IR-related features
- **FS**: Feature selection for automatic ranking and selection of top features from all of the above with Gaussian Processes.

Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) are used to evaluate the models. The error scores for all feature sets are reported in Table 1. Bold-faced figures are significantly better than all others (paired t-test with $p \leq 0.05$).

Feature type	System	#feats.	MAE	RMSE
BB	Baseline	17	14.32	18.02
	IR	35	14.57	18.29
	AF	108	14.07	18.13
	AF+IR	143	13.52	17.74
	FS	17	**12.61**	**15.84**
GB	AF	48	17.03	20.13
	FS	17	**16.57**	**19.14**
BB+GB	AF	191	14.03	19.03
	FS	17	**12.51**	**15.64**

Table 1: Results with various feature sets.

Adding more BB features (systems **AF**) improves the results in most cases as compared to the baseline systems **BL**, however, in some cases the improvements are not significant. This behaviour is to be expected as adding more features may bring more relevant information, but at the same time it makes the representation more sparse and the learning prone to overfitting. Feature selection was limited to selecting the top 17 features for comparison with our baseline feature set. It is interesting to note that system **FS** outperformed the other systems in spite of using fewer features.

GB features on their own perform worse than BB features but the combination of GB and BB followed by feature selection resulted in lower errors than BB features only, showing that the two features sets can be complementary, although in most cases BB features suffice. These are in line with the results reported in (Specia et al., 2013; Shah et al., 2013). A system submitted to the WMT13 QE shared task using QuEst with similar settings was the top performing submission for Task 1.1 (Beck et al., 2013).

5. Remarks

The source code for the framework, the datasets and extra resources can be downloaded from `http://www.quest.dcs.shef.ac.uk/`. The project is also set to receive contribution from interested researchers using a GitHub repository. The license for the Java code, Python and shell scripts is BSD, a permissive license with no restrictions on the use or extensions of the software for any purposes, including commercial. For pre-existing code and resources, e.g., `scikit-learn`, `GPy` and Berkeley parser, their licenses apply, but features relying on these resources can be easily discarded if necessary.

Acknowledgements

This work was supported by the QuEst (EU FP7 PASCAL2 NoE, Harvest program) and QTLaunchPad (EU FP7 CSA No. 296347) projects. We would like to thank our many contributors, especially José G. C. Souza for the integration with `scikit-learn`, and Lukas Poustka for his work on the refactoring of some of the code.

Bibliography

Beck, Daniel, Kashif Shah, Trevor Cohn, and Lucia Specia. SHEF-Lite: When less is more for translation quality estimation. In *Proceedings of WMT13*, pages 337–342, Sofia, 2013.

Biçici, E. *The Regression Model of Machine Translation*. PhD thesis, Koç University, 2011.

Biçici, E. Referential translation machines for quality estimation. In *Proceedings of WMT13*, pages 341–349, Sofia, 2013.

Biçici, E., D. Groves, and J. van Genabith. Predicting sentence translation quality using extrinsic and language independent features. *Machine Translation*, 2013.

Bojar, O., C. Buck, C. Callison-Burch, C. Federmann, B. Haddow, P. Koehn, C. Monz, M. Post, R. Soricut, and L. Specia. Findings of the 2013 Workshop on Statistical Machine Translation. In *Proceedings of WMT13*, pages 1–44, Sofia, 2013.

Callison-Burch, C., P. Koehn, C. Monz, M. Post, R. Soricut, and L. Specia. Findings of the 2012 Workshop on Statistical Machine Translation. In *Proceedings of WMT12*, pages 10–51, Montréal, 2012.

Papineni, K., S. Roukos, T. Ward, and W. Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th ACL*, pages 311–318, Philadelphia, 2002.

Rasmussen, C.E. and C.K.I. Williams. *Gaussian processes for machine learning*, volume 1. MIT Press, Cambridge, 2006.

Shah, K., T. Cohn, and L. Specia. An investigation on the effectiveness of features for translation quality estimation. In *Proceedings of MT Summit XIV*, Nice, 2013.

Specia, L., K. Shah, J. G. C. Souza, and T. Cohn. QuEst – a translation quality estimation framework. In *Proceedings of the 51st ACL: System Demonstrations*, pages 79–84, Sofia, 2013.

Address for correspondence:
Kashif Shah
`Kashif.Shah@sheffield.ac.uk`
Department of Computer Science
University of Sheffield
Regent Court, 211 Portobello, Sheffield, S1 4DP UK