Automating high-quality translations for Mobile Apps

Master Thesis

Stefan Wanzenried
from
Bern BE, Switzerland

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

May 2016

Prof. Dr. Philippe Cudré-Mauroux, eXascale Infolab, University of Fribourg, Switzerland, Supervisor
Roman Prokofyev, eXascale Infolab, University of Fribourg, Switzerland, Assistant

Abstract

Every day, hundreds of mobile applications are added to stores such as Google Play or AppStore. Many of them are intended to be used internationally, and thus require translation of the interface. At the same time, many more mobile apps are already available for download in these stores. Leveraging the translation bases of existing applications, we could immediately provide high-quality translations for new apps without need to go through a human-translation process.

This project aims to extract and parse translations of existing applications to see if they can be used to translate new ones. A prototype web application allows to translate existing apps from a given source to a target language. Translations are generated with different models in Statistical Machine Translation. Finally, the quality of the produced translations is evaluated and compared between the different models.

Contents

1 Introduction

Android, iPhone and Windows smart phones are available in countries all over the world. If an app is intended to be used internationally, translating it into multiple languages can be important to attract users. Users may only understand and use the app if its text is available in their native language. Offering multiple languages is therefore crucial for paid apps to increase sales. However, translating an app requires additional effort. From a developers perspective, all strings must be isolated in order to replace them with different values for each supported language. Furthermore, translating text itself is a difficult task which usually requires the work of a professional translator.

The main idea of this thesis is to collect translation data of existing apps and use them to translate new ones. Apps belonging to the same category share similar words or sentences. Also, most sentences in a mobile app are short and thus easier to translate than longer, more complicated sentences. Based on these facts, we claim that a Statistical Machine Translation (SMT) model produces good translations.

To reduce the scope of different mobile operating systems and languages, we focus on automatic translations of Android apps from English to French and German. From now on, the term *app* in this thesis refers to an Android application. French and German are chosen as target languages for the translations because they are natively spoken in Switzerland, which makes understanding the produced results easier.

SMT is the dominant approach in automating translations and used by services like *Google Translate*[1] and Microsoft's *Bing Translator*[2]. In this thesis, we use different SMT models and compare their results for translating mobile apps, both manually and automatically. A prototype web application is built, offering a common interface to translate strings or apps with the implemented translation systems.

[1] `https://translate.google.com` (accessed: 16 May 2016)
[2] `https://www.bing.com/translator` (accessed: 16 May 2016)

2 Problem Analysis

This chapter first describes why app developers should localize their apps. Next, we analyse what kind of translation services already exist for this task. Lastly, we briefly introduce SMT and explain why to use it for the purpose of translating apps.

2.1 Importance of Translating Mobile Apps

Google encourages developers to localize their apps on a localization checklist.[1]

> *Android and Google Play offer you a worldwide audience for your apps, with an addressable user base that's growing very rapidly in countries such as Japan, Korea, India, Brazil, and Russia. We strongly encourage you to localize as it can maximize your apps distribution potential resulting in ratings from users around the world.*

The developer should first identify the countries where the app will be distributed based on the overall market size and opportunity, app category, local pricing etc. According to the identified countries, the supported languages are determined. Ideally an app supports multiple languages already before the first version is available in the app store. Users may install the app only once and forget about it, if they are not able to understand it.

Since 2013, Google offers an app translation service [1]. Several developers who participated at the pilot program shared their results after translating their apps:

- The developers of the game *Zombie Ragdoll*[2] used this tool to launch their new game simultaneously in 20 languages in August 2013. When they combined app translation with local marketing campaigns, they found that 80% of their installs came from non-English-language users [1].

[1] `http://developer.android.com/distribute/tools/localization-checklist.html` (accessed: 16 May 2016)

[2] `https://play.google.com/store/apps/details?id=com.rvappstudios.zombieragdoll` (accessed 12 May 2016)

- Dating app *SayHi*[3] expanded into 13 additional languages using the app translation service. They saw 120% install growth in localized markets and improved user reviews of the professionally translated UI [1].
- The developer of card game *G4A Indian Rummy*[4] found that the app translation service was easier to use than their previous translation methods, and saw a 300% increase with user engagement in localized apps [1].

In 2012, company *App Annie*[5] (former called *Distimo*), published a study where they analysed the impact of translations to iPhone apps [13]. They found out that the use of native language is still limited in several countries, for example Italy, Russia or Brazil. Native-only apps are mostly used in Asian countries (see Figure 2.1). The study followed 200 iPhone apps that introduced native language support in a market and observed their growth. Only one week after the local language support was introduced with an app update, downloading increased by 128% and revenue by 26%.

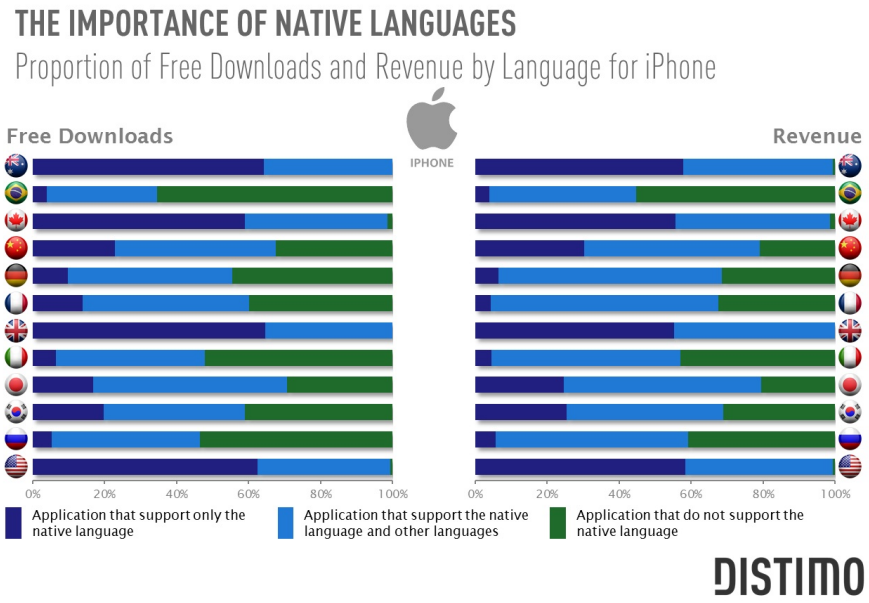


Figure 2.1: Graphic from [13] showing the proportion of free downloads and revenue of iPhone apps by language.

Beside offering a better user experience by translating apps, both examples show that proper localization also directly impacts the number of downloads and revenue.

2.2 Translation Services

Most of the existing services are based on human translations, including Google's app translation service [1]. The develper can choose from third-party vendors who are qualified by Google to offer high-quality

[3] `https://play.google.com/store/apps/details?id=com.unearby.sayhi` (accessed 12 May 2016)
[4] `https://play.google.com/store/apps/details?id=org.games4all.android.games.indianrummy.prod` (accessed 12 May 2016)
[5] `https://www.appannie.com` (accessed 12 May 2016)

translations. According to Google, the costs range from $75 to $150 depending on the app size.

Another possibility to translate apps quickly and convenient is to use Crowdsourcing. Here, we ask people in a crowdsourcing platform to do the translations for us. This approach is often used in combination with machine translation: Initial translations are produced with machine translation, which are then improved or corrected by human translators. Facebook used crowdsourcing to translate their website: "In March 2008, through crowdsourcing, the entire site was translated into French within 24 hours by allegedly over 4,000 native French speakers." [6] One example of a crowdsourcing translation platform is *Translation Cloud*[7]. Everyone who is fluent in at least two languages can login via Facebook account and earn money by translating or correcting strings. To avoid cheaters, an initial language proficiency test must be passed.

Translation services solely based on machine translation do not seem to exist. However, developers *could* use existing translation APIs to produce translations. Popular providers are Microsoft[8] and Google[9]. Microsoft's translation service offers a free plan which is limited to translate 2 million characters per month. Paid plans exist, where the amount of characters is increased, 32 million characters cost $300 per month. The translation API from Google does not offer any free plan but the pricing model is different: One pays $20 per 1 million characters of text.

2.3 Statistical Machine Translation

"Machine translation has a long history, but over the last decade or two, its evolution has taken on a new direction - a direction that is mirrored in other subfields of natural language processing. This new direction is grounded in the premise that language is so rich and complex that it could never be fully analyzed and distilled into a set of rules, which are then encoded into a computer program. Instead, the new direction is to develop a machine that discovers the rules of translation automatically from a large corpus of translated text, by pairing the input and output of the translation process, and learning from the statistics over the data." [8]

As SMT is the dominant approach in machine translation, it is consequential to use it for our purpose of translating mobile apps. In machine translation literature, the SMT models are usually trained on a large parallel corpus, for example the *European Parliament Proceedings Parallel Corpus* [6]. This corpus consists of many sentences for a given language pair, e.g. EN-FR holds over 2 million sentences. Our corpus will hold the extracted sentences from existing apps. We expect to collect less but shorter sentences.

To automatically evaluate the quality of machine produced text, a metric called *BLEU* [11] is mostly used. We adapt this procedure and also use BLEU to evaluate our translations in Chapter 8.

[6] `https://en.wikipedia.org/wiki/Crowdsourcing_as_Human-Machine_Translation` (accessed 12 May 2016)

[7] `http://www.translationcloud.net/` (accessed 13 May 2016)

[8] `http://datamarket.azure.com/dataset/bing/microsofttranslator` (accessed 12 May 2016)

[9] `https://cloud.google.com/translate/docs/` (accessed 12 May 2016)

3 Related Work

This chapter explains the theory of two popular models in SMT . The first one is called *Statistical Phrase-Based Translation* and was introduced in 2003 [5]. The second, more recent model, uses *Deep Learning Neural Networks* to perform translations [4, 12].

3.1 Phrase-Based Machine Translation

3.1.1 Model

Given the task to translate a sentence from a foreign language $f = f_1, f_2, f_3..., f_m$ into an english sentence $e = e_1, e_2, e_3..., e_n$, the translation model is based on conditional probability:

$$P(e|f)$$

To find the best english translation e_{best}, the following equation is used (Bayes rule).

$$\begin{aligned} e_{best} &= argmax_e P(e|f) \\ &= argmax_e \frac{P(f|e)P(e)}{P(f)} \\ &= argmax_e P(e|f)P(e) \end{aligned}$$

where $P(e)$ is called the *language model*[1] and $P(f|e)$ is the *translation model*.

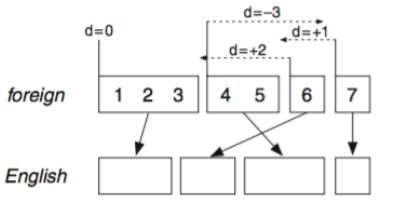
The translation model is further decomposed:

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

[1] `https://en.wikipedia.org/wiki/Language_model` (accessed: 17 May 2016)

where ϕ is called the *phrase translation probability* and d is the *reordering probability*. The foreign sentence *f* is broken up into *I* phrases $\bar{f}_i$. Each foreign phrase is translated into an English phrase $\bar{e}_i$.

Reordering is handled by a *distance based reordering model*. We define $start_i$ as the position of the first word of the foreign input phrase that translates to the *i*th English phrase, and end_i as the position of the last word of that foreign phrase. The reordering distance is the number of words skipped when taking foreign words out of sequence (see figure 3.1) [8].



Phrase	Translates	Movement	Distance
1	1–3	start at beginning	0
2	6	skip over 4–5	+2
3	4–5	move back over 4–6	−3
4	7	skip over 6	+1

Figure 3.1: Distance based reordering [8]

d is turned into a proper probability distribution by applying an exponentially decaying cost function $d(x) = \alpha^{|x|}$ with $\alpha\epsilon[0, 1]$. This means that movements of phrases over large distances are more expensive than shorter movements or no movement at all [8].

3.1.2 Learning a Phrase Translation Table

Building a phrase translation table involves three stages:

1. **Word alignment** The words of each sentence from the parallel corpus of the foreign and English language must be aligned in order to extract phrase pairs. The alignment is typically done by using one of the IBM models[2]. Figure 3.2 shows an example of word alignment for a top translation from our corpus.

2. **Extraciton of phrase pairs** Extract all phrase pairs of parallel sentences that are consistent with word alignment. A phrase pair ($\bar{f}$,$\bar{e}$) is consistent with an alignment *A*, if all words $f_1, ..., f_n$ in $\bar{f}$ that have alignment points in *A* have these with words $e_1, ..., e_n$ in $\bar{e}$ and vice versa [8].

3. **Scoring phrase pairs** Assign probabilities to phrase translations.

[2] `https://en.wikipedia.org/wiki/IBM_alignment_models` (accessed 13 May 2016)

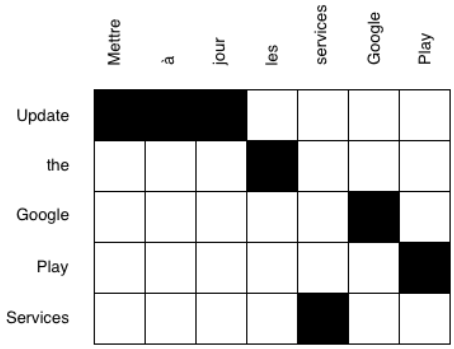


Figure 3.2: Word alignment between a top sentence of our parallel Enlgish/French corpus.

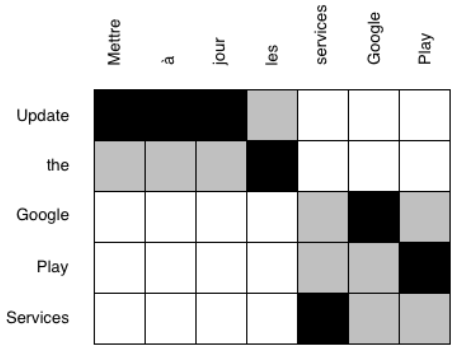


Figure 3.3: Extract phrase pairs consistent with word alignment.

We are able to extract the following phrase pairs from the example sentence (phrases marked with an * match the extractions from Figure 3.3):

- Update the Google Play Services / Mettre à jour les services Google Play
- Update / Mettre à jour
- Update the / Mettre à jour les *
- Google Play Services / services Google Play *
- The Google Play Services / les services Google Play
- Google Play / Google Play
- Google / Google
- Play / Play
- Services / services
- the / les

The last step is to estimate the phrase translation probabilities by relative frequency:

$$\phi(\bar{f}|\bar{e}) = \frac{count(\bar{e}, \bar{f})}{\sum_{\bar{f}_i} count(\bar{e}, \bar{f}_i)}$$

These probabilities are put in the phrase translation table together with the foreign phrases (see Figure 3.1).

Foreign Phrase	$\phi(\bar{f}\|\bar{e})$
paramètres	0.9
réglages	0.36
configuration	0.15

Table 3.1: Example of a phrase translation table containing the estimated probabilities for translating the English word "Settings" to French.

3.1.3 Decoding

Decoding is the process of finding a translation for a given input sentence. Recall the model to compute the translation probability:

$$e_{best} = argmax_e \prod_{i=1}^{I} \phi(\bar{f}_i|\bar{e}_i) d(start_i - end_{i-1} - 1) p_{LM}(e)$$

- **Phrase translation** Pick phrase $\bar{f}_i$ to be translated as phrase $\bar{e}_i$ by looking up the score from the phrase translation table.
- **Reordering** The previous phrase ended in end_{i-1}, the current phrase starts at $start_i$. Compute $d(start_i - end_{i-1} - 1)$
- **Language model** For a *n*-gram model, we need to keep track of the last $n - 1$ words. The score is computed for every added word w_i: $p_{LM}(w_i|w_i - (n-1), ..., w_{i-1})$

The algorithm reads the input sentence from left to right and creates a new hypothesis by picking any phrase translation at a time. The score is computed incrementally for each partial hypothesis. This is repeated recursively until all hypotheses have been expanded. A hypothesis covering all input words cannot be expanded further and forms an end point in the search graph (Figure 3.4). To get the best translation, we pick the completed hypothesis with the highest probability score and backtrack through the search graph [8].

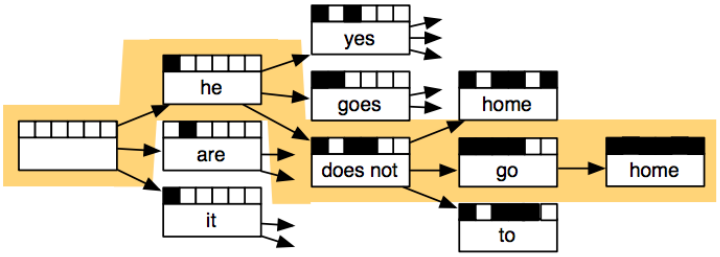


Figure 3.4: Hypothesis expanded by translating the German sentence “er geht ja nicht nach hause” to English. The squares on top indicate a coverage vector of translated German words (filled black if covered) [8].

This process creates an exponential number of hypothesis. To reduce the search space, the following methods are applied:

- **Recombination** Two hypothesis paths lead to two matching hypotheses, e.g. have the same English words in the output. The worse hypothesis is dropped.
- **Stack Pruning** Bad hypotheses are removed early. Hypothesis that have translated the same number of input words are put into a stack, so that they are comparable. The number of hypotheses in a stack is limited, for example by keeping only *k* hypotheses according to a score. This score is typically a combination of the partial probability score and a future cost estimation for translating the remaining sentence.

3.2 Deep Learning Neural Networks

Translating with neural networks is based on a encoder-decoder architecture [4, 12]. Two *recurrent neural networks* (RNN)[3] are involved: The first RNN encodes a variable-length sequence into a fixed-length vector representation. The second RNN decodes a given fixed-length vector back into a variable-length sequence. We first describe shortly the working of a RNN before explaining the encoder-decoder architecture in more detail.

3.2.1 Recurrent Neural Network

A recurrent neural network is a class of artificial neural network where connections between units form a directed cycle. It consists of a hidden state h and an optional output y which operates on a variable-length sequence $x = (x_1, ..., x_n)$. At each time step t, the hidden state h_t is updated by:

$$h_t = f(h_{t-1}, x_t)$$

where f is a non-linear activation function. The function described in [4] is motivated by a *long short-term memory (LSTM)* unit[4], but much simpler to compute and implement.

A RNN can learn a probability distribution over a sequence by being trained to predict the next symbol in a sequence. In this case, the output at each timestep t is the conditional distribution $p(x_t|x_{t-1}, ..., x_1)$.

[3] `https://en.wikipedia.org/wiki/Recurrent_neural_network` (accessed: 13 May 2016)
[4] `https://en.wikipedia.org/wiki/Long_short-term_memory` (accessed 13 May 2016)

3.2.2 Encoder

The encoder RNN reads each word of the source sentence sequentially in the form of a *One-hot vector*. This binary vector has the size of the vocabulary and zeros everywhere except a 1 at the position corresponding to the index of the word in the source vocabulary.

The internal state h_t changes each time a new word is read:

$$h_t = f(h_{t-1}, x_t)$$

At the end of processing each word, the hidden state of the RNN is a summary c of the source sentence (see Figure 3.5).

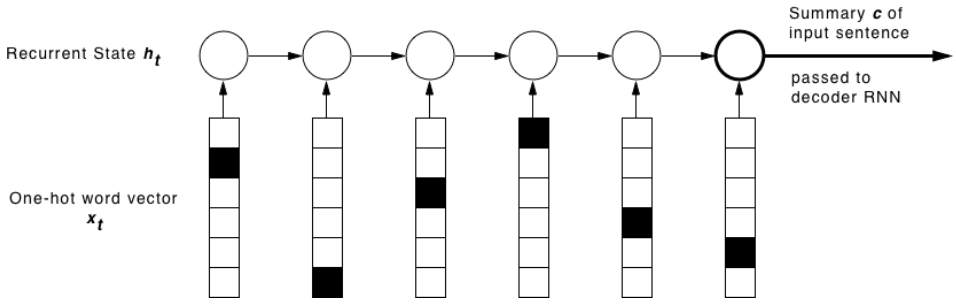


Figure 3.5: Encoder RNN: The final hidden state (marked bold) contains the summary vector.

It is very interesting how the summary vector c looks like. In [12] they projected multiple summary vectors to the two-dimensional space (see Figure 3.6). The plot shows that similar sentences are close together in the summary vector space.

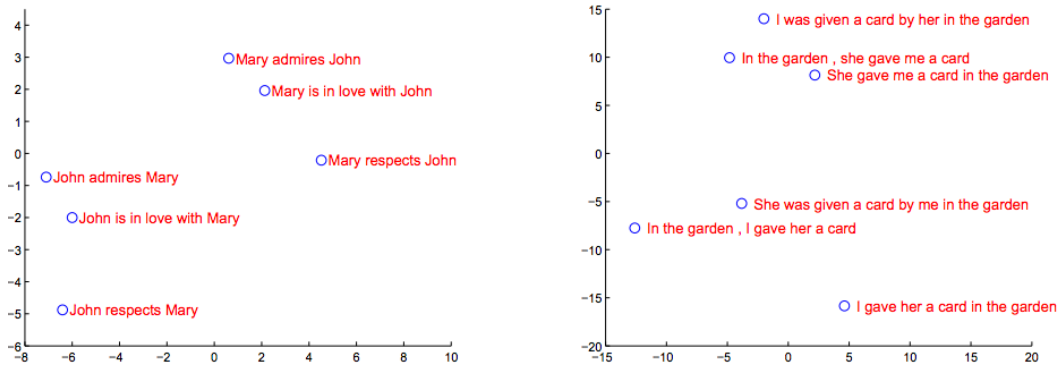


Figure 3.6: Summary vectors from multiple sentences. Similar sentences in terms of syntax and semantics are grouped together [12].

3.2.3 Decoder

The decoder is another RNN which is trained to generate the output sentence by predicting the next word y_t given the hidden state h_t. Both y_t and h_t are conditioned on y_{t-1} and the summary vector c given from the encoder RNN. The hidden state of the decoder is computed by

$$h_t = f(h_{t-1}, y_{t-1}, c)$$

The conditional distribution of the next target word is

$$P(y_t|y_{t-1}, y_{t-2}, ..., y_1, c) = g(h_t, y_{t-1}, c)$$

for given activation functions f and g. The latter must produce valid probabilities.

3.2.4 Training

The proposed encoder-decoder system is trained by maximizing the *log-likelihood*[5] of the training corpus. Training requires a parallel corpus C of source and target sentences (X^n, Y^n). The sentences are represented as sequence of one-hot vectors. We can compute the conditional log-probability of Y^n given X^n by $logP(Y^n|X^n, \theta)$. The log-likelihood of the training corpus is written as:

$$L(C, \theta) = \frac{1}{N} \sum_{n=1}^{N} logP(Y^n|X^n, \theta)$$

where N is the number of training samples and θ is the set of model parameters. Maximizing the log-likelihood function can be done by using *stochastic gradient descent*[6].

[5] https://en.wikipedia.org/wiki/Likelihood_function (accessed: 13 May 2016)
[6] https://en.wikipedia.org/wiki/Stochastic_gradient_descent (accessed: 13 May 2016)

4 Data Analysis

4.1 Apps

A total of 698 free Android apps were collected from various categories[1], ensuring a good mix of different translations. Figure 4.1 shows the top ten apps according to the number of available English translations. Note that the counts indicate the number of effective translations after eliminating uninteresting data during pre-processing the corpus (see Chapter 5). Most of the apps have similar counts for all the three languages. Surprisingly, some apps provide more translations for German and French than English. Different counts can happen because of:

1. Not all strings need to be translated, some English words remain the same in the target language e.g. “E-Mail”.
2. Android stores the translations key-value based in one XML file per language. These files could be out of sync, the developer either forgot to add or remove a French/German translation.
3. Some translations were abandoned during pre-processing.

34 apps did not isolate their strings in XML files, we assume that they are hardcoded in the programming code. This leaves an effective number of 664 apps where we could extract translations. However, 241 apps among them were not multilingual and therefore not useful for building translation systems. Unfortunately, apps are not marked as multi-language in the app store, so we only know if there are isolated translations available after downloading and extracting the data.

[1] Click on Categories to see a list of available categories at: `https://play.google.com/store/apps?hl=en` (accessed: 13 May 2016)

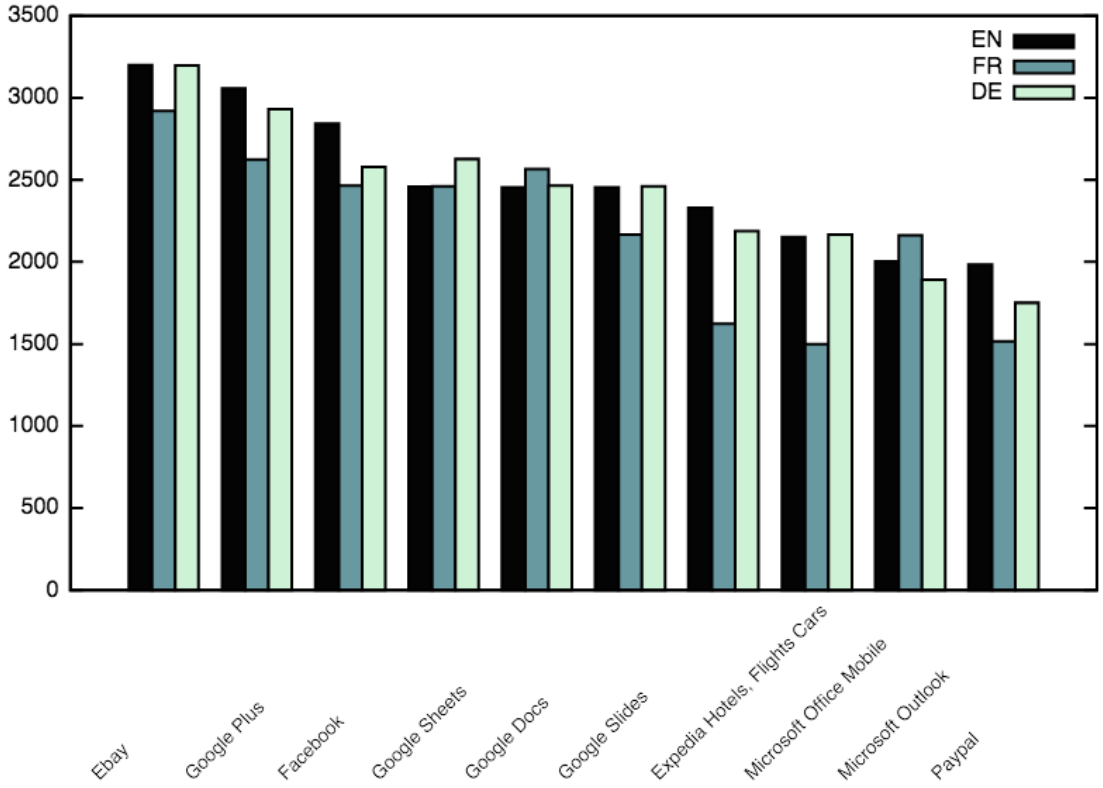


Figure 4.1: Top apps by the number of English translations

4.2 Languages

We collected a total of 423 multilingual apps, offering its content for at least two languages. Figure 4.2 shows the number of apps for the ten most used languages. French and German, where we focus on this thesis, are both in the top four. The top apps are translated in many more languages. Overall, 117 different languages were available.

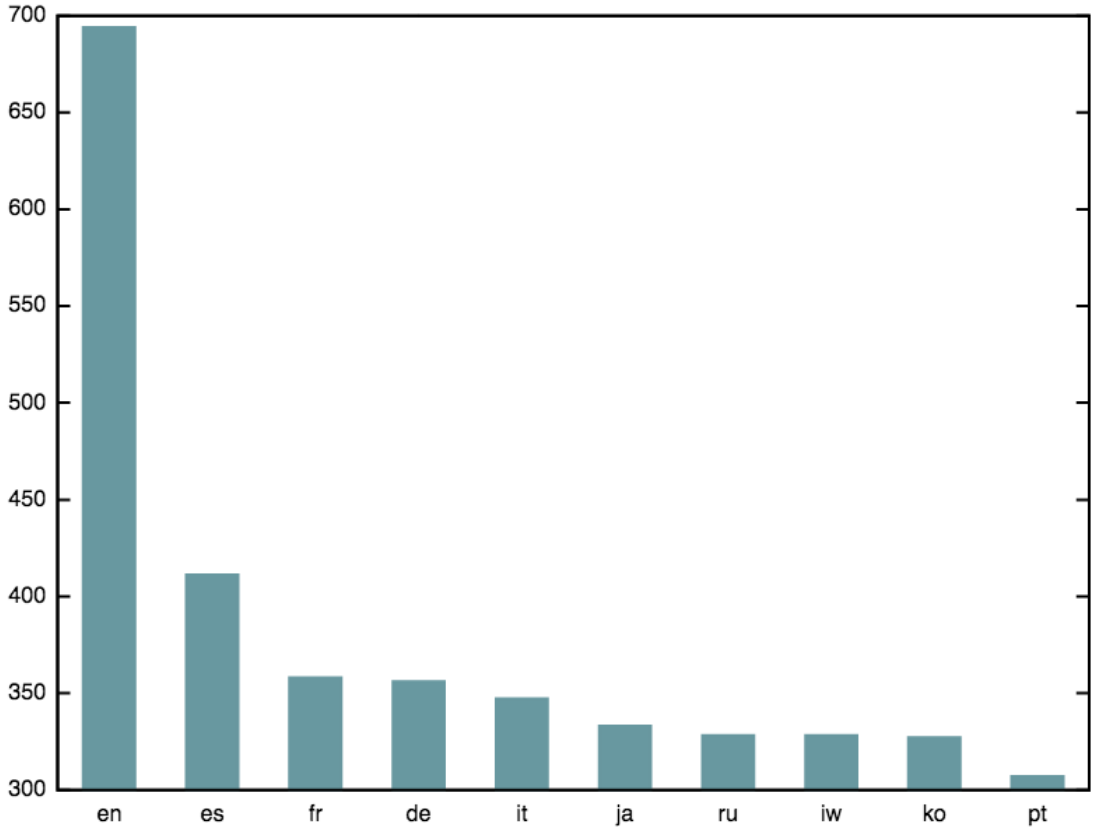


Figure 4.2: Number of apps available in the top ten languages.

4.3 Extracted Sentences

This sections covers statistics regarding the extracted sentences per language, including the top sentences, top n-grams and sentence lengths. Note that the term *sentence* in this section describes single words, sentences or multiple sentences, separated by punctuation.

In total, we collected 114'947 parallel sentences for the English-French language pair and 114'773 for English-German, respectively. The monolingual data holds 18'7642 English sentences, 126'883 French sentences and 126'975 German sentences. Again, the counts are measured after pre-processing the extracted data (see Section 5.2).

4.3.1 Top Sentences

EN	Count	FR	Count	DE	Count
Cancel	1894	Annuler	1067	Abbrechen	886
Done	1230	Supprimer	745	Google Play-Dienste aktivieren	730
Settings	1102	Connexion	745	Google Play-Dienste installieren	728
Search	884	Mettre à jour	486	Anmelden	664
Delete	704	Rechercher	485	Löschen	664
Enable Google Play services	692	Paramètres	450	Aktualisieren	652
Get Google Play services	690	Mettre à jour les services Google Play	370	Einstellungen	569
Log Out	556	Activer services google Play	368	Fertig	548
Update	545	Activer les services Google Play	366	Weiter	484
Share	541	Installer les services Google Play	366	Schliessen	428

Table 4.1: Top ten sentences for English, French and German

One can see that all three languages share similar top sentences, although they do not perfectly align.

4.3.2 Sentence Lengths

As expected, most sentences are very short and consist of less than 5 words for all languages. The lengths were measured on the monolingual corpus after tokenization, which means that punctuation is also included in the counts.

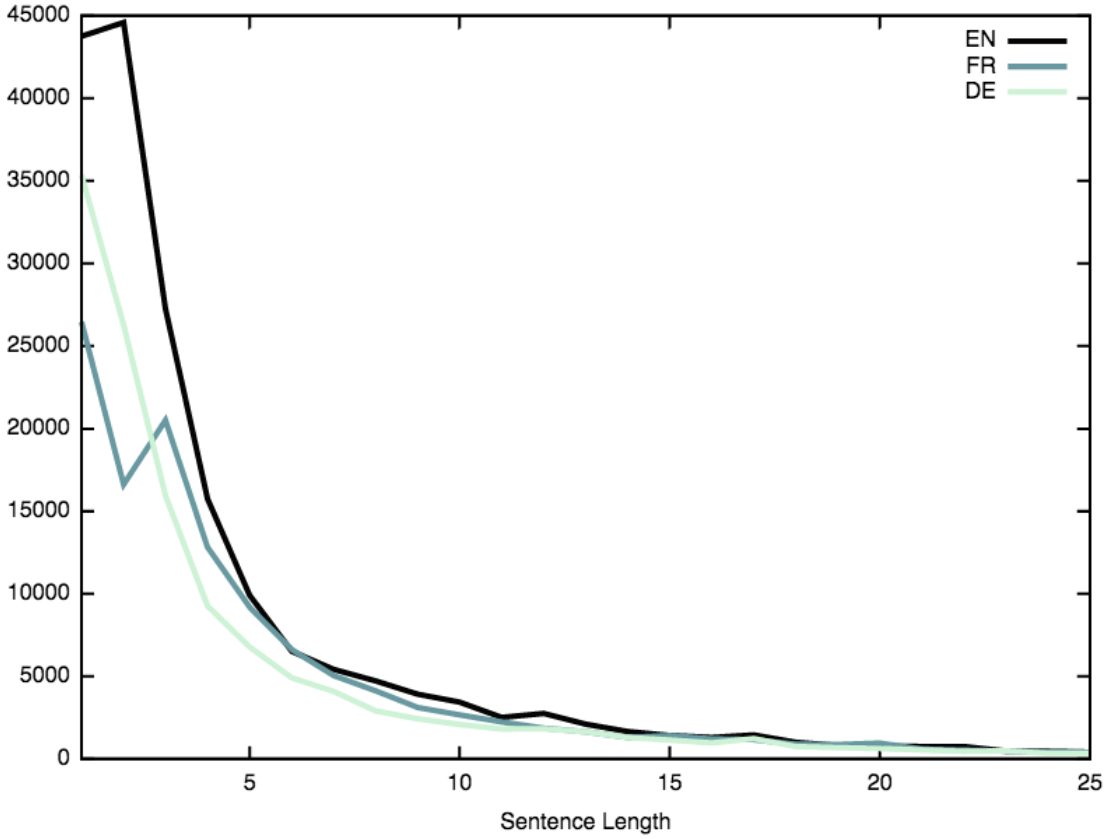


Figure 4.3: Length of sentences for English, French and German

4.3.3 N-Grams

EN	Count	FR	Count	DE	Count
to	28160	de	32818	Sie	14383
the	19306	à	13094	nicht	8216
your	14238	la	10956	die	7853
you	11144	le	10701	und	7366
and	9962	les	10619	zu	5791
a	9657	pour	8057	der	5783
in	8338	votre	7674	auf	5575
is	8087	des	7090	ist	5447
this	7522	et	7025	werden	5337
of	7340	pas	6976	von	5329

Table 4.2: Top unigrams from the monolingual corpus

EN	Count	FR	Count	DE	Count
Google Play	3348	Google Play	3366	Google PlayDienste	2639
Play services	2807	de la	2915	Sie die	1458
try again	2270	services Google	2905	Sie es	1183
to the	1705	à jour	2353	Sie sich	975
want to	1647	les services	1762	versuchen Sie	835
you want	1484	que vous	1593	kann nicht	806
will be	1359	de votre	1373	konnte nicht	795
Please try	1328	de passe	1303	es erneut	714
of the	1325	Impossible de	1284	auf Ihrem	667
is not	1258	a été	1027	in der	667

Table 4.3: Top bigrams from the monolingual corpus

EN	Count	FR	Count	DE	Count
Google Play services	2805	services Google Play	2889	versuchen Sie es	804
you want to	1407	les services Google	1714	Google PlayDienste aktivieren	565
Please try again	1187	mot de passe	920	Sie es erneut	465
try again later	794	ne fonctionnera pas	877	sind Google PlayDienste	452
Are you sure	740	ne sont pas	832	Google PlayDienste erforderlich	445
This app wont	686	Cette application ne	713	Bitte versuchen Sie	427
you sure you	630	application ne fonctionnera	712	PlayDienste erforderlich die	417
sure you want	596	Mettre à jour	620	Google PlayDienste installieren	396
and try again	579	mise à jour	571	Google PlayDienste aktualisieren	396
can not be	555	Google Play qui	480	Sie es später	394

Table 4.4: Top trigrams from the monolingual corpus

5 Data Extraction, Preparation and Storage

5.1 Extract Translations

An Android app has the form of a single binary file (suffix .apk). The XML files containing the translations are packed inside the .apk file. We used a software called *Apktool*[1] to reverse engineer existing apps and extract the needed XML files.

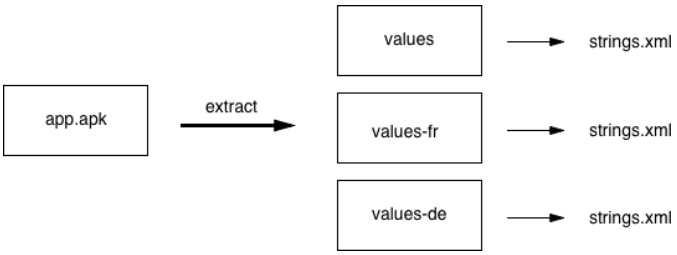


Figure 5.1: For each language, the translations are stored in a XML file

Inside the `strings.xml` files, translations are stored key-value based. Each string has a unique key which holds the translation value for each language. Here is an example how an English XML file looks like, followed by its German counterpart:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="close_app">Close</string>
    <string name="back">Back</string>
    ...
</resources>
```

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
```

[1] `http://ibotpeaches.github.io/Apktool/` (accessed: 13 May 2016)

```
    <string name="close_app">Schliessen</string>
    <string name="back">Zurck</string>
    ...
</resources>
```

5.2 Preprocess Translations

This task involves two parts. First of all, the translations are sanitized from unwanted information such as HTML tags. Some strings are omitted at this stage because they do not provide any meaningful value to the translation process, for example URLs. Secondly, the sanitized strings are tokenized so that they can be used to train a SMT system.

5.2.1 Sanitizing

The following rules are used to get rid of unwanted translations or parts of it:

- Trim the string from newlines (\n), tabs (\t) or carriage returns (\r)
- Strip any HTML tags
- Omit strings that are URLs (starting with http or www)
- Remove strings placeholders like `%s` or `%1$s`
- Ignore words with less than 3 characters
- Finally, the resulting string is trimmed again from spaces and must contain at least one alphanumerical character

5.2.2 Tokenization and Truecasing

Tokenization is required to separate words from punctuation. Each token is separated by a space character: "`Hi, my name is John.`" becomes "`Hi , my name is John .`"

After tokenizsation, *truecasing*[2] was applied to the text corpus. In contrast to lowercasing, truecasing determines the proper capitalization of words. This is done by first analyzing how words are mostly capitalized. The proper case is then applied to each word. It is especially useful for words starting a sentence, that are written uppercase in many languages. For example the word "This" starting a sentence will become "this", because it is usually written lowercase.

5.3 Data storage

5.3.1 Parallel and Monolingual Corpus

After pre-processing the translations, they are written to text files, one sentence per line. The corpus is split into parallel and monolingual data. A parallel corpus (often also called bitext) describes sentence aligned text-files of one language pair. All sentences must be aligned so that line x of the first language corresponds to line x of the second language. In order to get a random mix of translations from different apps, the sentences are shuffled before they are written to the files. The parallel corpus can be used by any

[2] `https://en.wikipedia.org/wiki/Truecasing` (accessed: 13 May 2016)

SMT system to train the models. If the model requires a separate development set of parallel sentences, we can split the existing corpus at any percentage, e.g. use 80% as training data and the remaining 20% as development set. In addition, data of each language is also stored monolingual. All sentences of one language are written into a single text file. This is for example used to build language models, which are an important part of a phrase-based translation model (see Chapter 3.1).

The data analysis (see Chapter 4) showed that the monolingual corpus holds more sentences than the parallel corpus for all three languages. The reason is that the XML files are not synchronized perfectly: If an English sentence, identified by the translation key, is missing in French or German, it cannot be part of the parallel corpus.

5.3.2 Apache Solr

We needed a simple way to store all translations, preferably with additional meta data, to perform data analysis and calculate statistics. *Apache Solr*[3] is used for this purpose. Solr is an open source text search platform built on top of *Apache Lucene*. It allows to store the translations with additional meta data, such as the translation key and a unique app ID. Due to its way of indexing text data, Solr is very fast when querying for translation strings. Furthermore, built in tools allow us to get nice statistics, for example the top translations grouped by language.

Solr is also used as *Baseline Translation System*, where we translate strings with a simple algorithm directly from the collected data in Solr (see Chapter 6.1).

5.3.2.1 Setup and Schema

Solr offers so called *cores* where each core manages a separate index, schema and configuration[4]. In our setup we created one core per language and one document per translation string. The schema is identical for each core (language) and consists of the following fields:

- **id** Unique ID used by Solr to identify a document.
- **app_id** Each app must have a unique app-ID.
- **key** The translation key from the XML file.
- **value** The translation value (word or sentence) corresponding to the above key.
- **value_lc** Again the translation value, additionally indexed with a special start- and ending delimiter. This allows us to search for exact values of any translation string.

[3] `http://lucene.apache.org/solr/` (accessed: 13 May 2016)

[4] `https://cwiki.apache.org/confluence/display/solr/Solr+Cores+and+solr.xml` (accessed: 13 May 2016)

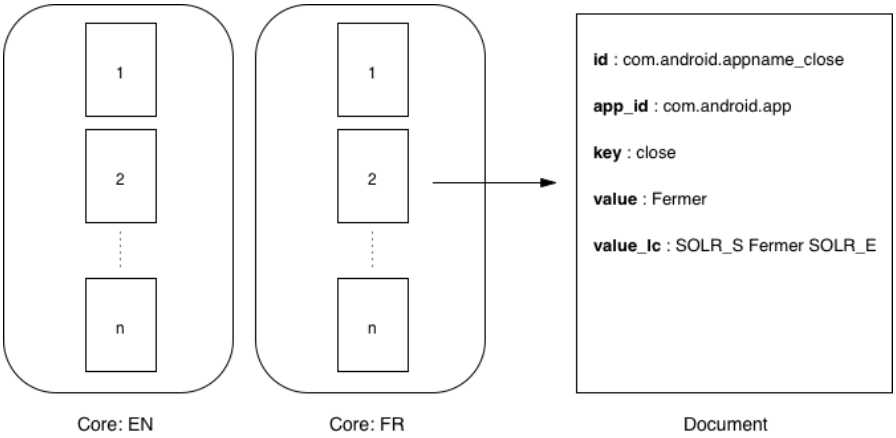


Figure 5.2: Translations are stored in a document inside a separate core per language

The data in the `value` and `value_lc` fields is stored tokenized and lowercased. The Solr *Standard Tokenizer*[5] splits text into tokens, treating whitespace and punctuation as delimiters. Delimiter characters are discarded.

5.3.2.2 Queries

Solr offers a REST based interface to query the database. There is also a built in web application which allows to inspect results. Here is an example query where we search for the string "Log Out" in the English core[6]. The maximum amount of results is set to 500 rows and the return type is JSON:

```
http://diufpc114.unifr.ch:8983/solr/en/select?q=value:"Log%20Out"&rows=
500&wt=json&indent=true
```

This query finds a total of 458 results (see Listing 1). There are not only exact matches returned, the term "Log Out" can occur as substring in a longer sentence. By default, Solr does not offer the possibility to return only exact matches. However, we can simulate this behaviour with the help of the `value_lc` field, which indexed the translation value together with a special start- and end delimiter (`SOLR_S` and `SOLR_E`):

```
http://diufpc114.unifr.ch:8983/solr/en/select?q=value_lc:"SOLR_S%20LogOut%
20SOLR_E"&rows=500&wt=json&indent=true
```

This query returns only 396 results, representing exact matches. Note that the case of the search term doesn't matter and whitespace and punctuation are ignored. Executing the above query with the term "log Out ." returns the same results. This behaviour is especially useful when looking up translations, as we don't need to worry about proper case or punctuation.

[5] `https://cwiki.apache.org/confluence/display/solr/Tokenizers#Tokenizers-StandardTokenizer` (accessed: 13 May 2016)

[6] You must be connected to the LAN of the University of Fribourg in order to access the referenced domain. Also note that the parameters should be fully url-encoded, this was omitted in this report for better readability.

```
"response": {
    "numFound": 458,
    "start": 0,
    "docs": [
      {
        "id": "com.flaregames.gargoyles_com_facebook_loginview_log_out_action",
        "app_id": "com.flaregames.gargoyles",
        "key": "com_facebook_loginview_log_out_action",
        "value": "Log Out",
        "value_lc": "SOLR_S Log Out SOLR_E"
      },
      {
        "id": "com.instagram.android_must_log_out_one_click_login",
        "app_id": "com.instagram.android",
        "key": "must_log_out_one_click_login",
        "value": "You must log out in order to login using this link.",
        "value_lc": "SOLR_S You must log out in order to login using this link. SOLR_E"
      },      ...
      ]
}
```


Listing 1: JSON response from Solr

6 Translation Process

This chapter describes different translation systems that were used to translate apps. We first introduce a simple *Baseline System*, that translates sentences based on the available data in Solr. The next section focuses on *Moses*, a phrase-based SMT system. Lastly, *Tensorflow* was used to provide translations with deep neural networks. The latter two implement closely the models introduced in Chapter 3.

6.1 Baseline System

The *Baseline System* makes direct use of the data stored in Solr and produces translations with a simple algorithm. Remember how the data is organized in Solr (see section 5.3.2): Each translation is stored in a document with a fixed schema and each document belongs to one core. To lookup a translation, the following procedure is used:

1. Lookup the string to translate in the core of the source language.
2. Parse the result documents and extract the `app_id` and `key` fields.
3. Lookup translations in the target language by querying the target core with the collected `app_ids` and `keys`.
4. Count the number of unique translations found. The best translation is the value that was mostly used.

Figure 6.1 shows an example for the available translations for the English word “Settings”. The best translation according to the highest count among all apps is “paramètres”.

Translation	Count
paramètres	48
réglages	3
settings	3
configuration	1

Table 6.1: Available translations for “Settings” in French.

6.1.1 Algorithm

Given a string $S = [\ s_1\ s_2\ ...\ s_n\ ]$ with words s_i in the source language, we want to produce a string $T = [\ t_1\ t_2\ ...\ t_n\ ]$ in the target language. The algorithm recursively tries to translate the longest substrings of S by looking up translations for the substrings in Solr. This process is repeated until there are left only single words s_i to translate.

1. Check if we find any translation for S directly. If so, return the translation and exit.

2. Start translating the longest substrings of S. To build the substrings, we move a “window” of size $length(S) - i$ from left to right over S, where i is the iteration level. Let $length(S) = 7$; the following two substrings of length 6 are possible at the first iteration ($i = 1$):

$$V_1 = [\ s_1\ s_2\ s_3\ s_4\ s_5\ s_6\ ]\ [\ s_7\ ]$$
$$V_2 = [\ s_1\ ]\ [\ s_2\ s_3\ s_4\ s_5\ s_6\ s_7\ ]$$

   Now, we search for translations for all substrings in both variations V_1 and V_2. If there are no translations available, the substring (window) size is reduced by one for the second iteration ($i = 2$):

$$V_1 = [\ s_1\ s_2\ s_3\ s_4\ s_5\ ]\ [\ s_6\ s_7\ ]$$
$$V_2 = [\ s_1\ ]\ [\ s_2\ s_3\ s_4\ s_5\ s_6\ ]\ [\ s_7\ ]$$
$$V_3 = [\ s_1\ s_2\ ]\ [\ s_3\ s_4\ s_5\ s_6\ s_7\ ]$$

   This process of reducing the substring length is repeated until we can translate a substring of at least one variation V_j.

3. Let’s consider what happens if there are translations available. For each variation V_j, we count the number of available translations for their substrings $\bar{S}_i$. We continue with the variation $\tilde{V}$ having the highest total count of translations among their substrings:

$$\tilde{V} = argmax \sum_i count\_translations(\bar{S}_i)$$

   If there is a translation available for each substring $\bar{S}_i$ of $\tilde{V}$, we can build the output string T directly by concatenating the substring translations:

$$T = \bar{T}_1\ \bar{T}_2\ \bar{T}_3\ ...\ \bar{T}_n$$

   Otherwise, we build the output string T incremental: If there is no translation found for substring $\bar{S}_i$, we set $S = \bar{S}_i$ and recursively call the algorithm again. If there are left single words to translate and no translation exists, we insert the source word into T by setting $t_i = s_i$.

This simple algorithm works well for translating short words or sentences where a direct translation exists in the target language. On the other hand, it suffers from the following problems if translating substrings is necessary:

- The sentences or words for the translated substrings are inserted at the same position in T where they occurred in the source string. This won't produce good results since words often need to be reordered depending on the target language.
- Bad translations can have a direct impact on the result. For example, due to an error or uncompleted translation in the XML files, the English word "the" could be mapped to a sentence with many words.

6.1.2 Example

Let us translate the source string "Open the settings to change your username" from English to French.

1. Check for a direct translation: No French translation available.
2. Start translating the substrings. The upper index indicates the number of translations found:

   $$V_1 = [\text{ Open the settings to change your }]^{0}\ [\text{ username }]^{28}$$
   $$V_2 = [\text{ Open }]^{63}\ [\text{ the settings to change your username }]^{0}$$

   Continue with Variation 2 since the count (63 + 0) is higher than the count from Variation 1 (0 + 28).
3. We set T = [ouvrir] and S = [the settings to change your username] and go recursive. Again, there does not exist a translation for S directly so the substrings are built:

   $$V_1 = [\text{ the settings to change your }]^{0}\ [\text{ username }]^{28}$$
   $$V_2 = [\text{ the }]^{1}\ [\text{ settings to change your username }]^{0}$$

4. T = [ouvrir ... nom d'utilisateur] ; S = [the settings to change your]

   $$V_1 = [\text{ the settings to change }]^{0}\ [\text{ your }]^{2}$$
   $$V_2 = [\text{ the }]^{1}\ [\text{ settings to change your }]^{0}$$

5. T = [ouvrir ... votre nom d'utilisateur] ; S = [the settings to change]

   $$V_1 = [\text{ the settings to }]^{0}\ [\text{ change }]^{26}$$
   $$V_2 = [\text{ the }]^{1}\ [\text{ settings to change }]^{0}$$

6. T = [ouvrir ... modifier votre nom d'utilisateur] ; S = [the settings to]

   $$V_1 = [\text{ the settings }]^{5}\ [\text{ to }]^{2}$$
   $$V_2 = [\text{ the }]^{1}\ [\text{ settings to }]^{0}$$

7. T = [ouvrir les paramètres vers modifier votre nom d'utilisateur] ; S = [ ]

At the last step, there are translations available for both substrings of V_1 and the algorithm derived the following string: "ouvrir les paramètres vers modifier votre nom d'utilisateur"

6.2 Moses

Moses[1] [9] is a open source SMT toolkit. The training process follows closely the phrase-based model described in section 3.1. The source code is available on *GitHub*[2]. This project is very active, there are frequent commits to the master branch by 72 contributors and documentation on their website is excellent.

6.2.1 Training

The training process requires sentence aligned parallel data to build the translation models, and monolingual data for the language models. We can make use of the sanitized and tokenized data as described in section 5.2. The training process involves the following steps.

1. Build a language model from the monolingual data of the target language. Moses includes the *KenLM toolkit*[3] for this task.

2. Align words of the parallel sentences. We used an external library *MGIZA*[4], an extension of the popular *GIZA++* word alignment toolkit [10]. MGIZA is multi-threaded and runs faster on a multi-core machine.

3. Extract phrases: All phrases are dumped into one big file. The following listing shows some example content of this file. Each line contains: English phrase, French phrase and the alignment points of matching words (English-French).

```
as you can see from |||  comme vous pouvez le voir ||| 0-0 1-1 2-2 3-3 4-5
as you can see |||  comme vous pouvez le voir ||| 0-0 1-1 2-2 3-3 4-5
as you can |||  comme vous pouvez le ||| 0-0 1-1 2-2 3-3
as you can |||  comme vous pouvez ||| 0-0 1-1 2-2 3-3
as you |||  comme vous ||| 0-0 1-1 2-2
as |||  comme ||| 0-0 1-1
```

4. Score phrases by computing the phrase translation probability $\phi(\bar{f}|\bar{e})$ and build the phrase table. Here is an example how the content of the phrase table is organized in Moses for English-French:

```
your version of Google Play |||  votre version de Google Play ||| 1 0.04 1 0.22
your version of Google |||  votre version de Google ||| 1 0.04 1 0.24
your version of |||  votre version de ||| 1 0.0427528 1 0.245609
your version |||  votre version ||| 1 0.635219 1 0.508584
your |||  ton ||| 1 0.591003 0.0434783 0.00469507
your |||  votre ||| 1 0.794566 0.73913 0.542431
your |||  your ||| 1 0.896708 0.130435 0.0173818
your ||| .  vos ||| 0.0416667 0.00483387 0.0434783 0.0112108
```

   Each line contains the phrase pairs together with the computed phrase translations scores, in the following order:

   (a) Inverse phrase translation probability $\phi(\bar{e}|\bar{f})$

   (b) Inverse lexical weighting $lex(\bar{e}|\bar{f})$

   (c) Direct phrase translation probability $\phi(\bar{f}|\bar{e})$

   (d) Direct lexical weighting $lex(\bar{f}|\bar{e})$

[1] `http://www.statmt.org/moses/` (accessed: 13 May 2016)
[2] `https://github.com/moses-smt/mosesdecoder` (accessed: 13 May 2016)
[3] `https://kheafield.com/code/kenlm/` (accessed 13 May 2016)
[4] `https://github.com/moses-smt/mgiza.git` (accessed: 13 May 2016)

The *lexical weighting* is an additional score to check the reliability of the phrase translation probability. Rare phrase pairs may cause problems if they are collected from noisy data. If both of the phrases $\bar{f}$, $\bar{e}$ only occur once in the training data, then $\phi(\bar{f}|\bar{e}) = \phi(\bar{e}|\bar{f}) = 1$. This often overestimates how reliable rare phrase pairs are. Lexical weighting decomposes phrase pairs into its word translations, so that we can check how well they match up [8].

The output of the training process is the phrase translation table and a configuration file `moses.ini` containing all necessary configuration for the decoder to translate sentences.

Training one model, e.g. English-French with all parallel data only took about one hour. A more time consuming task is *Tuning*, which is explained in the next section.

6.2.2 Tuning

Tuning refers to the process of finding the optimal weights for the linear translation model of Moses:

$$p(e|f) = \phi(f|e)^{weight\_tm} LM(e)^{weight\_lm} D(e,f)^{weight\_dm} W(e)^{weight\_wp}$$

The probability cost that is assigned to a translation is a product of probability costs of four models:

- $\phi(f|e)$ The **phrase translation table** ensures that the source and target phrases are good translations of each other.
- $LM(e)$ The **language model** ensures that the output of the target language is fluently.
- $D(e,f)$ The **distortion model** allows for reordering of the input sentence, but at a cost. The more reordering, the more expensive is the translation.
- $W(e)$ The **word penalty** ensures that the translations do not get too long or too short.

There is a weight assigned to each of these components that influences their importance. These weights can be passed to the decoder when translating sentences. The optimal weights depend on the languages and training corpus. However, Moses offers to optimize the weights automatically by maximizing the BLEU score [11] on a small, separate set of parallel sentences (tuning set).

We used 80% of the available parallel data for training and the remaining 20% for tuning. Tuning takes much more time than training. Moses executes different tuning runs and measures the BLEU score achieved on the tuning set. The optimal weights were usually determined after 6-11 runs, which took about 3-6 hours.

6.2.3 Decoding

The decoder of Moses is controlled by the `moses.ini` file. This file stores the path to the phrase translation table and the language model. Additionally it also holds the tuned weights for the translation model. Many more configuration options[5] can be set in this file or passed as parameters when executing the decoder. Here is a summary from [7] how the decoder of Moses works:

1. The decoder collects all phrase translations from the phrase table that could be applied on the input strings of words (translation options). The translation options are stored with the following information:
    - First source word covered

[5] `http://www.statmt.org/moses/?n=Moses.DecoderParameters` (accessed 13. May 2016)

- Last source word covered
- Phrase translation in the target language
- Phrase translation probability

2. The output sentence is generated left to right in form of hypotheses. Each hypothesis is represented by:

   - A link to the best previous state, so that we later can back track through the graph to find the best translation.
   - The cost (probability) and covered source words so far. A low cost means high probability.
   - An estimate of the future cost, how expensive it is to translate the remaining words of the source sentence.

   The decoder uses *Recombination* and a *Beam Search* (see Section 3.1.3) to reduce the size of the search space. The beam search compares hypothesis covering the same number of source words in stacks. Given the cost so far and the future cost estimation, hypotheses that fall outside of the beam are pruned. The beam size can be defined by threshold and histogram pruning. A relative threshold cuts out a hypothesis with a probability less than a factor α of the best hypotheses. Histogram pruning keeps a certain number n of hypotheses in each stack.

Example We translate the same example string as used in the previous chapter "Open the settings to change your username" to French. Here is the command to invoke the decoder:

```
echo 'Open the settings to change your username' | /home/stefan/mosesdecoder/bin/moses
    -f /home/stefan/AppTranslator/data/moses/en-fr/mert-work/moses.ini -verbose 3 &>
    moses_decoding.out
```

We used histogram pruning with a stack size of 100 and the optimal model weights determined after tuning.

The decoder produces the following output string: "ouvrir les paramètres pour modifier votre nom d' utilisateur". Here are some statistics on how the decoder derived the target sentence:

- Total translation options collected: 197
- Total hypotheses considered: 42614
- Total hypotheses discarded: 31220
- Total hypotheses recombined: 8292
- Total hypotheses pruned: 2401

6.3 Tensorflow

Tensorflow[6] is a software library for machine intelligence, open sourced by Google in November 2015. The source code is available on GitHub[7] and the project is still under development.

Tensorflow includes a library to learn sequence to sequence models with neural networks. The implementation follows the encoder-decoder RNN model described in Section 3.2.

[6] `https://www.tensorflow.org/` (accessed: 13 May 2016)
[7] `https://github.com/tensorflow/tensorflow` (accessed: 13 May 2016)

6.3.1 Training

To train the models, we followed the tutorial on sequence-to-sequence models [2]. The python script was modified to fit our needs:

- We could skip the part where Tensorflow downloads and prepares the sample data since we want to train the models on our own data.
- The Tokenizer was replaced with another function doing nothing because our corpus is already tokenized.
- Additional parameters for the training process were added to make the training script more generic. Parameters include the source and target language and the path to the parallel corpus.

Training requires the following model parameters to be set. The values in square brackets represent the chosen default values. In our evaluation (see Chapter 8) we trained models with different values.

- **Path to training and development corpus** The parallel training data is used to train the models. Development data is required to evaluate how well the model performs after each training step.
- **Source vocabulary size** Size of the vocabulary of the most common words for the source language. Words that are not part of the vocabulary are replaced with an `UNK` token. [40'000]
- **Target vocabulary size** Size of the vocabulary for the target language. [40'000]
- **Size of model layer** How many cells are used in the RNNs. [2]
- **Number of layers** Number of layers used in the RNNs. [256]
- **Steps per checkpoint** How many training steps to do before writing the current state of the network from memory to disk. [50]
- **Learning Rate** The initial learning rate used when evaluating the model after each training step. [0.5]
- **Learning Rate Decay Factor** The learning rate decays by this factor. [0.99]

Training one model (e.g. from English to French) was during about two days with the parameters indicated above. At each checkpoint, when Tensorflow saves the current model to disk, some statistics are printed:

```
global step 71000 learning rate 0.2708 step-time 1.88 perplexity 1.08
  eval: bucket 0 perplexity 1.07
  eval: bucket 1 perplexity 1.03
  eval: bucket 2 perplexity 1.10
  eval: bucket 3 perplexity 1.65
global step 71600 learning rate 0.2576 step-time 1.79 perplexity 1.07
  eval: bucket 0 perplexity 1.07
  eval: bucket 1 perplexity 1.04
  eval: bucket 2 perplexity 1.07
  eval: bucket 3 perplexity 1.55
```

One can see the current *perplexity* of different *buckets* at timestep t. Tensorflow uses bucketing in combination with padding to efficiently handle sentences of different lengths. Each sentence pair is put in a bucket according to the lengths of the source and target sentence. The sentences are then padded to fit the length of the bucket. The bucket (x, y) contains sentences of length x in the source language and sentences of length y in the target language. Shorter sentences are padded with a special `PAD` symbol. For example, the English sentence "I go." and corresponding French sentence "Je vais." are put into the

bucket $(5, 10)$ with vectors padded to the correct size: `[PAD PAD "." "go" "I"]` and `[GO "Je" "vais" "." EOS PAD PAD PAD PAD PAD]` [2]. Behind the scenes, Tensorflow uses k different models for the k defined buckets. Reversing the source sentence was shown to improve results in [12]. We used the default buckets from the tutorial: `[(5, 10), (10, 15), (20, 25), (40, 50)]`.

The perplexity indicates how well the model of each bucket predicts the samples of the development set. A low perplexity means that the model is good in predicting the development sentences. Again, we used 80% as training data and 20% as development data. The (infinite) training loop was stopped after the perplexities didn't improve any more.

6.3.2 Decoding

In contrast to Moses, the decoding process in Tensorflow does not offer any options to influence the translation result. One problem is that unknown words not being part of the target language vocabulary are output with an `UNK` token. Our Baseline System and Moses both replace unknown words with the original word in the source language. Not having this behaviour *can* impact the BLEU score: If the unknown word is the same as in the source language, we get a penalty when comparing unigrams, because the word is not present in the output sentence (see Section 8.1). To overcome this problem, the decoding process was extended by the following steps:

1. After reading the sentence to translate, we store the words that were converted to `UNK` tokens in a list.

2. Let the decoder produce the output sentence and check if it contains `UNK` tokens. If so, replace each `UNK` token with a stored unknown word from input sentence.

The correct position of the unknown word in the output sentence is not guaranteed, as one can't map `UNK` tokens from the input to `UNK` tokens from the output. However, at least in BLEU, the position of the unigram is neglected.

Example Let us again translate the sentence "Open the settings to change your username" to French. Assume that the word "change" is not part of the English vocabulary. Firstly, the sentence is mapped to tokens-ids corresponding to the words in the vocabulary, e.g.

$$\begin{bmatrix} Open \\ the \\ settings \\ to \\ change \\ your \\ username \end{bmatrix} = \begin{bmatrix} 675 \\ 1654 \\ 201 \\ 53 \\ 2 \\ 98 \\ 6997 \end{bmatrix}$$

where ID 2 represents the `UNK` token. The unknown word "change" is stored. If the decoder derives the sentence "ouvrir les paramètres pour `UNK` votre nom d' utilisateur", the `UNK` token is then replaced by "change".

To be consistent with the examples of the Baseline System and Moses, here is the translation produced by Tensorflow: "modifiez affichage Paramètres Paramètres votre utilisateur de votre nom". This translation is obviously worse than the results of the other two systems.

7 Web Application

A simple web application was built to test and compare the different translation systems. It offers a common interface to translate strings with the Baseline System, Moses and Tensorflow. Furthermore, there is the possibility to translate XML files containing translations, so the web application *could* be used by developers to translate apps.

Additionally, the web application offers a functionality in terms of data analysing, called *Term Variations*. Given a string, source and target language, it returns all different translations from Solr that were used among all apps to translate the string, together with a count. For example, for the English term "settings" the following translations are available in French: "paramèteres", "réglages", "configuration", "settings". The last one indicates that at least one app did not translate "settings" correctly to French.

Visit the following URL to access the web application: `http://diufpc114.unifr.ch`[1]

7.1 User Interface

The user interface is divided into three steps. First of all, one needs to specify the source and target language, translation input and the system that should be used to perform the translations. The input can either be plain strings or a XML translation file (see Figure 7.1). Secondly, there is the possibility to set decoding options depending on the chosen translation system. Lastly, the translation results are displayed together with some debug information.

[1] You must be connected to the LAN of the university of fribourg in order to access the web application.

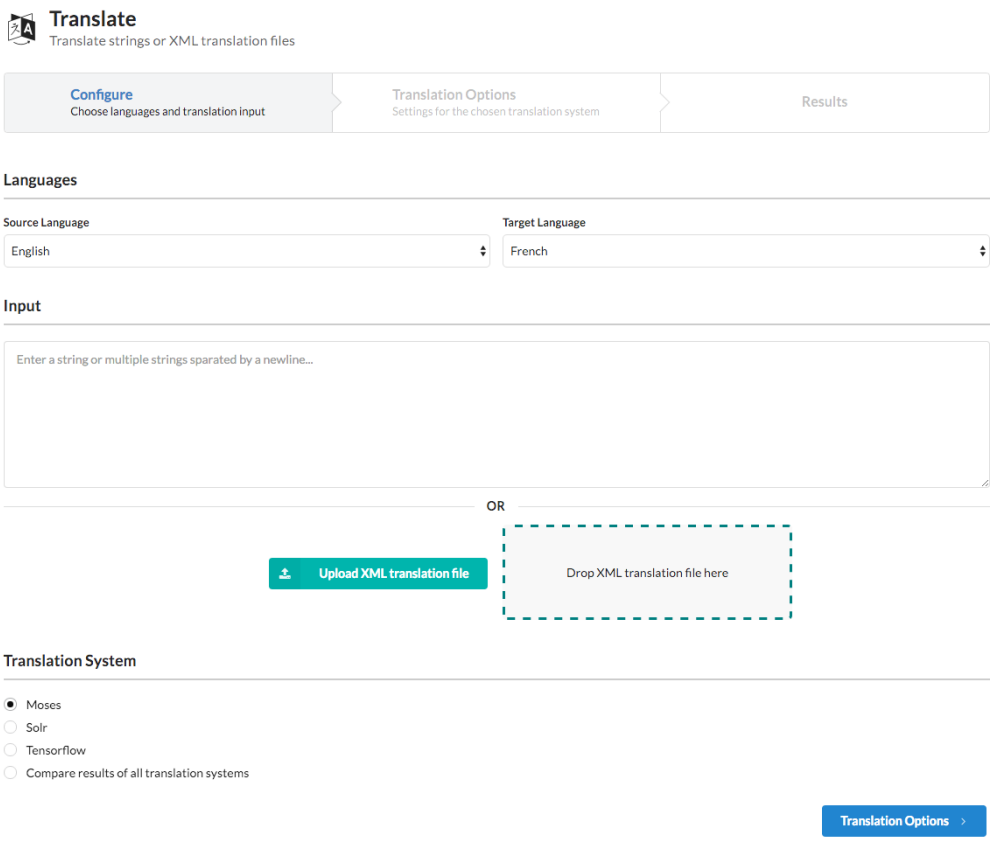


Figure 7.1: First step to use the web application: Set source and target languages, translation input and choose the translation system.

7.1.1 Translation Options

Moses

- **Drop unknown words** Words not in the vocabulary are either dropped or replaced by the word in the source language. [No]
- **Verbose** Verbosity level of the debug output. [2]
- **Stack size** Number of hypotheses kept on each stack while decoding. [100]
- **Tune weights manually** If enabled, the four weights for the translation model can be set manually (see Section 6.2.2). Normally, the optimal weights from the tuning process are used. [No]

Baseline System (Solr)

- **Number of Rows** Number of rows returned by Solr when searching for a translation. [100]

Tensorflow

- **Number of Layers** Number of layers in the RNNs. [2]

- **Size** Number of Cells in each layer. [256]

Note that the provided parameters must match a model that was trained with the same values, otherwise Tensorflow will fail to load the model.

7.2 Architecture

Each translation system implements a common interface (see UML in Figure 7.3). This makes it easy to support additional translation engines at any time.

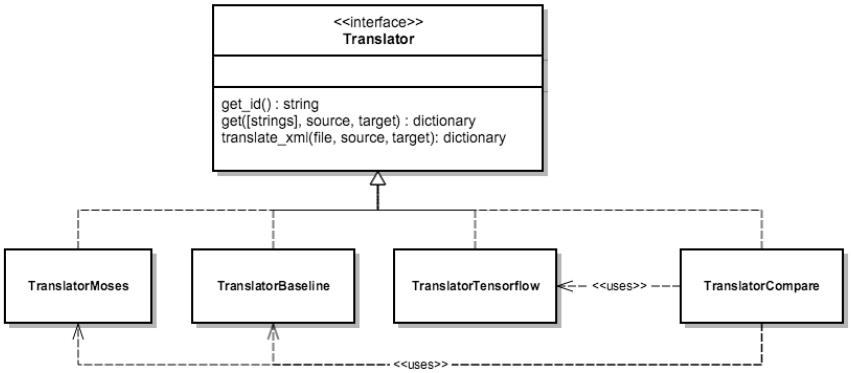


Figure 7.2: High level overview of the application. Each translation system implements the *Translator* interface. The class *TranslatorCompare* is special in a way that it gets the translations from all implemented systems, so that the results can be compared next to each other.

The backend offers a REST based interface for the different functionalities. The frontend communicates through Ajax requests with that interface and handles the JSON responses accordingly.

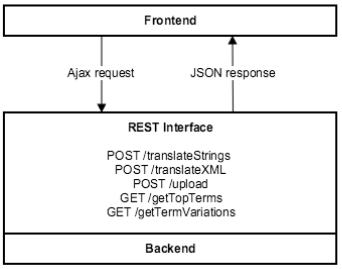


Figure 7.3: The frontend communicates with the backend through a REST interface.

The business logic is implemented in *Python*, the REST interface is powered by a microframework called *Flask*[2]. Flask has a built-in server and offers *RESTful request dispatching*, among many other features. Table 7.1 shows the possible endpoints of the REST interface together with the expected request data.

Endpoint	Request Data
translateStrings (POST)	decoder: The ID of the translation system to use, e.g. “moses” decoder_settings: JSON object of settings for the decoder strings: Array of strings to translate lang_from: Source language code (e.g. “en”) lang_to: Target language code (e.g. “fr”)
translateXML (POST)	decoder: The ID of the translation system to use decoder_settings: JSON object of settings for the decoder xml_filename: Hashed filename of the previous uploaded XML file lang_from: Source language code lang_to: Target language code
getTermVariations (GET)	source: Source language code target: Target language code term: String in source language
upload (POST)	file: Data from uploaded XML translation file

Table 7.1: REST endpoints

[2] `http://flask.pocoo.org/` (accessed: 16 May 2016)

The frontend uses *AngularJS*[3] together with the *Semantic UI*[4] CSS framework.

[3] `https://angularjs.org/` (accessed: 16 May 2016)
[4] `http://semantic-ui.com/` (accessed: 16 May 2016)

8 Evaluation

How can we automatically measure the quality of machine translated text? There exists an algorithm called *BLEU* [11] which is mostly used for this task. We first shortly describe how BLEU works, then explain our evaluation setup and finally present the BLEU scores achieved for the different translation systems.

8.1 BLEU (Bilingual Evaluation Understudy)

"The closer a machine translation is to a professional human translation, the better it is." This is the central idea behind the BLEU algorithm [11]. BLEU computes a score by comparing machine translated sentences (called *candiates*) against single or multiple reference translations. The score is averaged over the whole corpus to estimate the overall translation quality. The output of BLEU is a number between 0 and 1 that indicates the similarity of the candidate and reference translations. The closer a value is to 1, the more similar are the texts.

8.1.1 Algorithm

BLEU compares n-grams of the candidate with n-grams of the reference translation and counts the number of matches. These matches are position-independent. The more the matches, the better the candidate translation is [11].

The score is based on a modified n-gram precision. One counts up the maximum number of times a n-gram occurs in any single reference translation. Next, one clips the total count of each candidate n-gram by its maximum reference count: $count_{clip} = min(count, max\_ref\_count)$. The clipped counts for all n-grams are added up and divided by the total number of candidate words [11].

Example Consider this example from [11] that computes a modified unigram precision:

- Candidate: the the the the the the the
- Reference 1: The cat is on the mat.
- Reference 2: There is a cat on the mat.

The word "the" appears twice in the first reference and once in the second reference, it is therefore clipped to: $count_{clip} = min(7, 2) = 2$. The final modified unigram precision is $P = \frac{2}{7}$.

To compute the modified precision score p_n for the entire corpus, we compute the n-gram matches for each sentence. Next, the clipped n-gram counts for all candidates are added up and divided by the number of candidate n-grams in the corpus:

$$p_n = \frac{\sum_{C \in \{Candidates\}} \sum_{n-gram \in C} count_{clip}(n-gram)}{\sum_{C' \in \{Candidates\}} \sum_{n-gram' \in C'} count(n-gram')}$$

In order to prevent very short candidates from receiving a too high precision score, a *brevity penalty* is introduced. This penalty is 1.0 if the candidate's length is the same as any reference translation's length. The closest reference sentence length is called *best match length*. The brevity penalty is computed over the entire corpus. First, we compute the reference lengths r by summing the best match lengths for each candidate sentence in the corpus. We choose the brevity penalty to be a decaying exponential in r/c, where c is the total length of the candidate translation corpus [11].

The final BLEU score is a combination of the brevity penalty PB and the geometric mean of the modified precision scores p_n of the corpus:

$$BP = \begin{cases} 1 & c > r \\ e^{(1-\frac{r}{c})} & c \leq r \end{cases}$$

$$BLEU = BP \, exp\left(\sum_{n=1}^{N} w_n \, log \, p_n\right)$$

The baseline in [11] used $N = 4$ and uniform weights $w_n = 1/N$.

8.2 Setup

The BLEU scores were computed in different evaluation runs for the results of translating English to French and German. We used *k-fold cross-validation* to predict how the different models would perform if they were applied to independent data. The parallel corpus was split into 6 equal sized samples. One sample was always reserved as development set for Moses (Tuning) and Tensorflow, the remaining 5 samples were used to evaluate BLEU. For each evaluation run, four samples were merged into training data and a single sample was used as test set (see Figure 8.1). One sample contains 19'158 parallel sentences for English-French and 19'129 sentences for English-German.

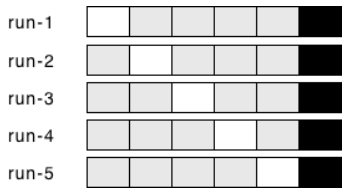


Figure 8.1: Splitting the parallel corpus: The white rectangle represents the test data, grey means training data and the black rectangle is the fixed development sample.

The Baseline System needed a special setup. First of all, the development set is ignored since there is no evaluation or tuning involved in this model. Secondly, we needed to index the training data for each

evaluation run separately into Solr. This was achieved by creating new cores in Solr according to the following schema:

$$source - target - run - k - (source|target)$$

where $source$ and $target$ are the source and target language and k is the index of the evaluation run (1-5). For example, in the first evaluation run for translating English to French, the French training data is put into the core `en-fr-run-1-fr`.

8.3 BLEU Scores

A script[1] included in Moses was used to compute the scores. As proposed in [11], n-grams up to $n = 4$ are considered in the calculation. Our corpus offers only one reference translation per sentence since apps are not translated multiple times into the same language. Note that the BLEU scores are reported as percentages, this format seems mostly used in machine translation literature. The left table shows BLEU scores achieved for translating English to French, the right one for English to German. The BLEU scores are reported for each evaluation run together with the average ø and standard deviation σ.

8.3.1 Moses

EN-FR	BLEU
run-1	47.45
run-2	53.19
run-3	49.91
run-4	51.5
run-5	51.74
ø	**50.76**
σ	1.95

EN-DE	BLEU
run-1	41.6
run-2	44.53
run-3	44.12
run-4	46
run-5	43.73
ø	**44.00**
σ	1.42

8.3.2 Tensorflow

EN-FR	BLEU[a]	BLEU[b]	BLEU[c]	BLEU[d]
run-1	19.86	19.36	23.13	20.53
run-2	26.7	21.29	25.68	22.44
run-3	25.27	19.57	23.2	21.07
run-4	20.64	20.58	20.48	26.86
run-5	27.43	21.42	22.21	28.49
ø	**23.98**	**20.44**	**22.94**	**23.88**
σ	3.13	0.85	1.68	3.2

[a] 2 Layers, 256 Units, 40'000 words
[b] 3 Layers, 512 Units, 40'000 words
[c] 1 Layers, 128 Units, 40'000 words
[d] 2 Layers, 256 Units, 50'000 words

EN-DE	BLEU[a]	BLEU[b]	BLEU[c]	BLEU[d]
run-1	18.72	18.58	20.02	25.56
run-2	23.95	20.34	19.93	24.13
run-3	22.14	19.96	22.04	23.62
run-4	19.66	18.82	19.86	26.87
run-5	17.03	19.71	20	24.11
ø	**20.3**	**19.48**	**20.37**	**24.86**
σ	2.46	0.67	0.84	1.2

[a] 2 Layers, 256 Units, 40'000 words
[b] 3 Layers, 512 Units, 40'000 words
[c] 1 Layers, 128 Units, 40'000 words
[d] 2 Layers, 256 Units, 50'000 words

[1] `https://github.com/moses-smt/mosesdecoder/blob/master/scripts/generic/multi-bleu.perl` (accessed: 16 May 2016)

8.3.3 Baseline System

EN-FR	BLEU
run-1	25.08
run-2	29.28
run-3	25.35
run-4	26.2
run-5	25.7
ø	**26.32**
σ	1.53

EN-DE	BLEU
run-1	8.74
run-2	9.57
run-3	9.03
run-4	9.68
run-5	8.98
ø	**9.2**
σ	0.36

8.4 Observations

One can see that Moses produces the best translations according to the BLEU metric. The achieved BLEU scores are doubled compared to Tensorflow and the Baseline System. While deep learning neural networks achieve a slightly higher BLEU score than phrase based systems in [12], this does not apply to our context. One reason probably is that our corpus is too small. In machine translation literature, the translation models are usually trained on much more data. The parallel corpus English-French from the *European Parliament Proceedings Parallel Corpus* [6] contains over 2 million sentences, which means ~105 times more sentences than our corpus holds. Other than that, Moses has the advantage that its phrase table contains all possible words, while in Tensorflow, there is a fixed limit of the vocabulary size. Compared to Moses, Tensorflow also lacks of a tuning process.

Surprisingly, increasing the model parameters in Tensorflow (add more layers and units to the network) did not improve translation quality. We achieved the best results with 2 layers and 256 units. Using 3 layers and 1204 units reduces the standard deviation though. Increasing the vocabulary size to 50'000 words did improve the BLEU score only for translating English to German.

The Baseline System produces similar results as Tensorflow for the translations from English to French. However, BLEU score is very low for translations from English to German. The reason is that the grammar of English and German is very different. Most sentences in English follow the *Subject-Verb-Object* (SVO) syntax, which is not the case for German sentences. "In German, the main verb must be the second element in the independent clause. This often requires an inversion of subject and verb"[2]. The Baseline System simply translates substrings without reordering, so n-grams in the candidate won't match with n-grams in the reference sentence. While the BLEU score for unigrams is reasonable, bigrams, trigrams and 4-grams do not contribute much to the overall BLEU score. The French grammar also follows the SVO syntax, which explains the better results for the translations from English-French.

[2] `http://esl.fis.edu/grammar/langdiff/german.htm` (accessed: 16 May 2016)

9 Conclusion

The question if it is possible to provide high-quality translations for apps with machine learning can be answered with "Yes". Still, one might argue that this answer depends on the definition of *high-quality*. On one hand, we achieved good BLEU scores with a phrase-based system, as shown in Chapter 8. The achieved BLEU scores for Moses are higher than the scores achieved in [5, 12] on the *European Parliament parallel corpus* [6], though the context is different, so a direct comparison is not possible. However, it still proofs our initial claim: Based on existing data and the similarity of sentences among different apps, we achieved a reasonable quality for machine translated sentences. On the other hand, machine translated text is almost never perfect. Bad app translations can do more harm than good, users might get confused if they cannot understand the content. Because of that, machine translated text *should* be reviewed and corrected or completed by humans.

Is there a market for a system proposed in this thesis? Maybe, but there is also the question how legal it is from a laws point of view. The *terms of service* from the Google Playstore include the following paragraph: "You may not sell, rent, lease, redistribute, broadcast, transmit, communicate, modify, sublicense or transfer or assign any Content or your rights to Content to any third party without authorization, including with regard to any downloads of Content that you may obtain through Google Play. Use of any tool or feature provided as an authorized part of Google Play (for example, "Social Recommendations") shall not violate this provision so long as you use the tool as specifically permitted and only in the exact manner specified and enabled by Google." [1] According to this quotation, authorization would be necessary to use existing content, which might also include the translation data.

9.1 Future Work

The current state of the project could be improved in many ways. Currently, all available translations are used to train the SMT models, which also includes wrong or bad translations. We could rate translations based on metadata of the app they are coming from, e.g. number of downloads, average rating, number of ratings ect. In general, the more existing high-quality translations we collect to train the SMT models, the better the quality of our produced translations is.

[1] `https://play.google.com/intl/en_ch/about/play-terms.html` (Accessed: 16 May 2016)

Another possibility is to analyse what kind of sentences are translated well on which system. With this knowledge in mind, we could choose the optimal system based on the input sentence.

Lastly, the prototype web application could be improved as well. After translating an XML file, we could offer to download the results as XML file again, among other formats. An interesting option could be to connect the web application with a crowdsourcing service, where the produced translations can be corrected and extended by people from a community.

Bibliography

[1] Google play app translation service. http://android-developers.blogspot.ch/2013/11/app-translation-service-now-available.html. Accessed: 12 May 2016.

[2] Sequence-to-sequence models with tensorflow. https://www.tensorflow.org/versions/r0.8/tutorials/seq2seq/index.html#sequence-to-sequence-models. Accessed: 13 May 2016.

[3] Website dedicated to research in statistical machine translation. http://www.statmt.org/. Accessed: 12 May 2016.

[4] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[5] Philipp Koehn Franz, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. pages 127–133, 2003.

[6] Philipp Koehn. European parliament proceedings parallel corpus 1996-2011. http://www.statmt.org/europarl/. Accessed: 17 May 2016.

[7] Philipp Koehn. Moses user manual and code guide. http://www.statmt.org/moses/manual/manual.pdf. Accessed: 17. May 2016.

[8] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, 2010.

[9] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-burch, Richard Zens, Marcello Federico, Nicola Bertoldi, Chris Dyer, Brooke Cowan, Wade Shen, Christine Moran, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation, August 12 2015.

[10] Franz Josef Och and Hermann Ney. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51, 2003.

[11] Kishore Papineni, Salim Roukos, Todd Ward, and Wei jing Zhu. Bleu: a method for automatic evaluation of machine translation. pages 311–318, 2002.

[12] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

[13] Analyst at Distimo Tiuri van Agten. The impact of app translations. https://web.archive.org/web/20150327084808/http://www.distimo.com/publications, 2012. Accessed: 5 May 2016.

A
Source Code

The complete source code is available on GitHub:
`https://github.com/wanze/AppTranslator`

Note that the code does not include external libraries like Moses or Tensorflow.