TECHNISCHE UNVERSITÄT DRESDEN

THEORY AND IMPLEMENTATION OF IBM MODEL 2

Author: Arezoo Kashefi Pour Dezfooli
Supervisor: Toni Dietze
Professor in charge: Professor Heiko Vogler

Declaration

I, Arezoo Kashefi Pour Dezfooli, hereby declare that this thesis including the text and figures – except for parts explicitly specified – is my personal work. Furthermore, I have not used any resources other than those that are duly cited.

Date: 31.07.2013

Signature:

Acknowledgement

I would like to take this opportunity to thank all who assisted in completing this work, specially my supervisor, T. Dietze who spent a great amount of time and energy in helping me correct and improve my work. I would also like to thank Prof. H. Vogler for his invaluable input towards this work. Finally, I would like to thank my husband for his patience and understanding. Thank you all.

Abstract

In statistical machine translation (SMT), we use statistical models based on parallel corpora to help translate a sentence from one language to another. Our SMT requires us to build a model of the English language, a model for translation based on an English-French parallel corpus and to used these to find the best translation of a French sentence to English. These are called the *language model*, *translation model* and the *decoder*, respectively.

In this work, the inner workings of a SMT system that used a trigram language model, the IBM model 2 translation model and a stack decoding algorithm as a decoder, are examined and implemented.

Contents

Chapter 1

Introduction

In the field of *machine translation*, where the intention is to use software to translate text or speech from one natural language to another, the use of data-driven techniques and statistical methods is growing. In the sub-field of *statistical machine translation*, one main idea is to use statistical models and find the parameters of such models by making use of and analyzing bilingual parallel corpora. We will be discussing one such model, namely, IBM model 2, in this work.

In this thesis, we will be working towards building a statistical machine translation (SMT) system. Our system will make use of the translation model, IBM model 2, first introduced by Brown et al. [BPPM93], a trigram language model for which the original idea dates back to Shannons work in information theory in 1948 [Sha48] and we utilized Chapter 7 of the book "Statistical Machine Translation" by P. Koehn [Koe10] as the main reference, and a decoder for which we used the stack decoding algorithm introduced by Wang and Waibel [WW97].

We will now give an overview of the contents of each chapter in this thesis. Chapter 1 is our introduction to the topic of this thesis and overview. Chapter 2 contains mathematical preliminaries needed for the better understanding of the topic. It is split into two sections. Section 2.1 covers probability distributions and uses [Koe10] as a reference for basic structure. Section 2.2 is about estimation and uses some definitions given by D. Prescher [Pre04].

In Chapter 3, we endeavour to explain in detail the workings of our SMT system. Chapter 3 contains four sections, section 3.1 is the chapter introduction and will explain what we need to build our SMT system and why we need them. The next three sections will then explain in detail, the language model, translation model and decoding.

In Section 3.2, we define a language model and particularly explain n-gram language models in detail. We then provide a means to asses the quality of a language model by way of perplexity. And finally, we conclude the section by explaining the necessity for count smoothing and providing a simple smoothing method, namely α-smoothing.

Section 3.3 describes the translation model. In this section, we cover IBM models 1 and 2, we give their parameters and algorithms to estimate these parameters. To better understand these algorithms, we give a brief explanation of the Expectation-Maximization Algorithm using [Pre04] as our main reference. We explain how training works and use model 1 to help initialize model 2.

Section 3.4 describes a stack decoding algorithm for IBM model 2. This section describes the algorithm and a method of scoring hypothesis using IBM model 2.

Chapter 4 follows the same pattern as chapter 3 for its main topic division. It is split into three sections, each of which explains the manner in which the language model, the translation model and the decoding were implemented using Java. We highlight the different steps, the difficulties along the way and the final results.

Chapter 5 concludes this thesis by giving a summary and discussion on the topic.

Chapter 2

Mathematical Preliminaries

In order to better understand the upcoming text, we will review here some notions of probability distribution and estimation.

2.1 Probability Distributions

In this section we will give a brief review of notions in probability theory that we will encounter in the coming text. This section is based mainly on [Koe10]. We will start with some of the important formal definitions. Let us start with a countable set Ω known as the *sample space*. We can define a function p on the sample space as follows:

$$\mathrm{p}\colon \Omega \to [0,1] \tag{2.1}$$

such that:

$$\sum_{\omega\in\Omega} \mathrm{p}(\omega) = 1 \tag{2.2}$$

This function is known as the *probability function*. In an experiment of chance, Ω is the set of all possible outcomes. Any subset $A \subseteq \Omega$ is then known as an *event*. The probability of the outcome of the experiment being event A is then as follows:

$$\mathrm{p}(A) = \sum_{a\in A} \mathrm{p}(a) \tag{2.3}$$

Then the pair (Ω, p) is called the *probability space*.

For example, in throwing a die all the possible outcomes are the die landing on the numbers 1-6 so we have:

$$\Omega = \{1, 2, 3, 4, 5, 6\}$$

Since ω has 6 elements and we assume our die is fair and according to equation 2.2 we have that $\sum\limits_{\omega\in\Omega} \mathrm{p}(\omega) = 1$, then we can conclude that: $\forall\omega \in \Omega\colon\ \mathrm{p}(\omega) = \frac{1}{6}$
We can then have the events $A = \{2\}$ or $B = \{a \in A \mid a < 4\}$. The probabilities for these events will be:

$$\mathrm{p}(A) = \frac{1}{6} \quad \text{and} \quad \mathrm{p}(B) = \frac{1}{2}$$

Definition 1 (Random Variable)**.** *Let A be an arbitrary set. The function $X\colon \Omega \to A$ is called a* random variable over A*. For all $a \in A$ we define:*

$$\mathrm{p}(X = a) = \mathrm{p}(X^{-1}(a))$$

In our die throwing experiment we can define X as a random variable over the events odd and even numbers. Then $X(1) =$ odd, $X(2) =$ even, $X(3) =$ odd and so on. What if we now want to calculate the probability of a die landing on a number both even and smaller than 4? We would need to calculate the intersection of two different event sets *even* and *smaller than 4*.

Definition 2 (Probability Distribution)**.** *Given a random variable X over Ω such that it can take all values $x_1, ..x_n \in \Omega$, the* probability distribution of X *is the list of probabilities* $\mathrm{p}(x_i)$*, $1 \leq i \leq n$, for all $x_i \in \Omega$ and it satisfies the following:*

(1) $0 \leq \mathrm{p}(x_i) \leq 1$

(2) $\sum\limits_{i=1}^{n} \mathrm{p}(x_i) = 1$

Definition 3 (Joint Probability Distribution)**.** *Given the random variables $X_1, ..., X_n$ over a set of events $A_1, ..., A_n$ such that $x_1 \in A_1, x_2 \in A_2, ..., x_n \in A_n$, the* joint probability distribution *is calculated as follows:*

$$\mathrm{p}(X_1 = x_1, ..., X_n = x_n) = \mathrm{p}(X_1^{-1}(x_1) \cap ... \cap X_n^{-1}(x_n))$$

So p(even and smaller than 4) $= \frac{1}{6}$.

Notation 1: From this point on we may refer to elements of events as directly belonging to the random variable representing the event. So we may write:
$x \in X$ *instead of* $x \in A$ and X is a random variable over A.

Notation 2: From this point on we will write only the elements and not the random variables in the probability functions. So we may write:
$\mathrm{p}(a)$ *instead of* $\mathrm{p}(X = a)$ where X is a random variable over A and $a \in A$.

Definition 4 (Variable Independence)**.** *Two random variable X and Y are* independent *if and only if for all $x \in X$ and $y \in Y$,* $\mathrm{p}(x, y) = \mathrm{p}(x) \cdot \mathrm{p}(y)$ *holds.*

Definition 5 (Conditional Probability Distribution)**.** *A* conditional probability distribution *is a probability distribution describing the outcome of an event y given event x has already occurred. It is denoted by* $\mathrm{p}(y \mid x)$ *and is defined as:*

$$\mathrm{p}(y \mid x) = \tfrac{\mathrm{p}(x,y)}{\mathrm{p}(x)}$$

Reformulating this definition we arrive at the *chain rule*:

$$\mathrm{p}(x, y) = \mathrm{p}(x) \cdot \mathrm{p}(y \mid x) \tag{2.4}$$

Adding one more event to this we have:

$$\mathrm{p}(x, y, z) = \mathrm{p}(x) \cdot \mathrm{p}(y \mid x) \cdot \mathrm{p}(z \mid x, y)$$

And we can generalize this to:

$$\mathrm{p}(x_1, ..., x_n) = \mathrm{p}(x_1) \cdot \mathrm{p}(x_2 \mid x_1) \cdot ... \cdot \mathrm{p}(x_n \mid x_1, ..., x_{n-1}) \tag{2.5}$$

In order to find the probability of x given y, according to definition 7 and using the chain rule to replace $\mathrm{p}(x, y)$, we have:

$$\mathrm{p}(x \mid y) = \frac{\mathrm{p}(x) \cdot \mathrm{p}(y \mid x)}{\mathrm{p}(y)} \tag{2.6}$$

which is known as the *Bayes rule*.

2.2 Estimation

In this section we will discuss estimation methods used in this text. Starting with the countable set $\mathcal{X}$, we define a *corpus* as a real-valued function $f\colon \mathcal{X} \to \mathcal{R}_{\geq 0}$ where $\mathcal{R}_{\geq 0}$ is the set of all real numbers greater or equal to zero. All $x \in \mathcal{X}$ is known as a *type* and each value of f is known as a *type frequency* and we will then define the *corpus size* as follows:

$$|f| = \sum_{x \in \mathcal{X}} f(x) \tag{2.7}$$

Let p now be a probability distribution on $\mathcal{X}$. We define the *relative-frequency estimate* on f as follows:

$$p(x) = \frac{f(x)}{|f|} \tag{2.8}$$

Definition 6 (Probability Model)**.** *Let $\mathcal{X}$ be a set of types and let $\mathcal{M}$ be a set of probability distributions on $\mathcal{X}$. $\mathcal{M}$ is called a* probability model *on $\mathcal{X}$ and elements of $\mathcal{M}$ are known as* instances *of the model. The set of all probability distributions on $\mathcal{X}$, denoted by $\mathcal{M}(\mathcal{X})$, is known as the* unrestricted probability model*. A model is known as* restricted *otherwise.*

Definition 7 (Likelihood Estimate)**.** *Let $\mathcal{X}$ be a set of types, f a finite corpus and p a probability distribution over $\mathcal{X}$. We define the* likelihood *of f under p as:*

$$L(f;p) = \prod_{x \in \mathcal{X}} p(x)^{f(x)}$$

Definition 8 (Maximum-Likelihood Estimate)**.** *Let $\mathcal{X}$ be a set of types, f a finite corpus, $\mathcal{M}$ a probability model on $\mathcal{X}$ and $L(f;p)$ the likelihood of the corpus. The* maximum-likelihood estimate *of f with respect to $\mathcal{M}$ is then defined as:*

$$\operatorname*{argmax}_{p \in \mathcal{M}} L(f;p)$$

Chapter 3

Statistical Machine Translation

3.1 Introduction

In *statistical machine translation* we wish to translate sentence f from a source language F (e.g. French) to a sentence e in a target language E (e.g. English) using statistical models.

We start with the basic assumption that any sentence e can, with a certain probability, be the translation of f. In order to narrow it down, we will need to know "How likely is e as the translation, given f?" in other words we need $\mathrm{p}(e \mid f)$. Obviously the best translation will then be the e that maximizes $\mathrm{p}(e \mid f)$, i.e. $\operatorname*{argmax}_{e \in E} \mathrm{p}(e \mid f)$.

We use the Bayes rule to break our problem into smaller sub-problems as follows:

$$\mathrm{p}(e \mid f) = \frac{\mathrm{p}(e) \cdot \mathrm{p}(f \mid e)}{\mathrm{p}(f)} \tag{3.1}$$

Maximizing $\mathrm{p}(e \mid f)$ depends only on finding the right e and since the denominator is independent of e, we must maximize the numerator, i.e. $\operatorname*{argmax}_{e \in E} \mathrm{p}(e) \cdot \mathrm{p}(e \mid f)$. The numerator consists of two probabilities. The first part, $\mathrm{p}(e)$ is known as the *language model.* It assigns to all sequences of English words the probability that they form an English sentence spoken by a native.

The second part, $\mathrm{p}(f \mid e)$ is known as the *translation model.* This model assigns probabilities to French strings being translations of English strings. Though this may seem like it is adding to our work to estimate two probability models instead of our original one, it is not. Since we will be translating from French to English we care whether our final e is a well-formed English sentence but when estimating $\mathrm{p}(f \mid e)$, our French sentences need not be.

Once we have our probability models, we will need to search for the e that will maximize this product. The process of finding an efficient and optimal search is known as *Decoding*.

In the following sections we will discuss each of these problems in more detail.

3.2 The Language Model

In this section we will discuss the *language model*, which is necessary for any statistical machine translation system. We wish for our translations to be fluent sentences spoken in English. For this purpose we need to know how probable it is that a sequence of words form a sentence of a native speaker. A good language model should give a sentence with the correct word order a higher probability than a sentence containing the same words in the wrong order.

For example, consider a sentence with the correct translation of “the house is small”, but a word for word translation of “small the is house”. Our language model p should have a higher probability for the former and a lower one for the latter, meaning

$$\mathrm{p}(\text{the house is small}) > \mathrm{p}(\text{small the is house}).$$

Correct word order is not the only job of the language model. In the cases of words with multiple possible translations, it is the language models duty to pick the correct translation. If, for example, the choice was between the sentences “I'm going home” and “I'm going house”, the former should have the higher probability.

Definition 9 (Language Model)**.** *Let A be a finite non-empty set called an* alphabet*. A* Language *E over A is a subset of A^*, $E \subseteq A^*$. A* Language Model *is then a probability distribution* $\mathrm{p}\colon E \to [0, 1]$.

3.2.1 n-Gram Language Models

An *n-gram* is a sequence of n items, in our case words. By analyzing large amounts of English text we can check how probable it is for a group of n words to occur in a certain sequence. Then we use these statistics to for example see that the word “home” follows “going” far more often than that word “house” does so. In doing so, we have an *n-gram language model.*

Let w be a sequence of words $w = w_1 ... w_m$, we calculate $\mathrm{p}(w)$ by analyzing large amounts of texts and go through it and count how often w appears.

Even though we would like it if our language model could perfectly predict the exact probability of every sequence of possible words, that is unfortunately not possible. Our model is limited by the amount of text we have to analyze and the context of that text.

Also most longer sequences may never appear in our text. However, we may be able to collect sufficient data for shorter sequences, so it is prudent to break the problem of finding the probability of w into smaller problems, namely we will try to predict one word at a time. This means predicting w_1 being the start of the sequence, that w_2 will follow given w_1, that w_3 will follow given w_1 and w_2 and so on. In other words we can break our probability of w down using the chain rule where the final probability ensures a sequence of length m:

$$\begin{aligned} \mathrm{p}(w) &= \mathrm{p}(w_1 w_2 ... w_m) \\ &= \mathrm{p}(w_1)\, \mathrm{p}(w_2 \mid w_1) ...\, \mathrm{p}(w_m \mid w_1 w_2 ... w_{m-1})\, \mathrm{p}(m | w_1 ... w_m) \end{aligned} \tag{3.2}$$

In order to estimate the word probability distributions that make up $\mathrm{p}(w)$ we will limit ourselves to the previous $n-1$ words for history assuming only those words affect the probability of the next word. So we get:

$$\mathrm{p}(w_m \mid w_1 w_2 ... w_{m-1}) = \mathrm{p}(w_m \mid w_{m-n+1} ... w_{m-2} w_{m-1}) \tag{3.3}$$

This is strictly untrue but because of data sparsity, reliable statistics can only be collected for shorter word histories. Our models are then named for the number of words they take into account and this is their *order* meaning an n-gram model has an order of n.

In practice we will have # symbols to mark the start and end of our sentences. This helps us better determine which words are more likely to appear at the beginning or end of a sentence. So in the case of *bigrams* where n is one, in Equation 3.2 we replace $\mathrm{p}(w_1)$ by the equivalent $\mathrm{b}(w_1 \mid \#)$ and $\mathrm{p}(m|w_1...w_m)$ by $\mathrm{b}(\# \mid m)$. The probabilities in between will be split into conditional probabilities with only a one word history. Thus, our probability model will be as follows for the bigram model b:

$$\begin{aligned} \mathrm{p}(w) &= \mathrm{p}(w_1 ... w_m) && (3.4) \\ &= \mathrm{b}(w_1 \mid \#) \cdot \mathrm{b}(w_2 \mid w_1) \cdot ... \cdot \mathrm{b}(w_m \mid w_{m-1}) \cdot \mathrm{b}(\# \mid w_m) && (3.5) \\ &= \mathrm{b}(w_1 \mid \#) \cdot \Big(\prod_{i=2}^{m} \mathrm{b}(w_i \mid w_{i-1})\Big) \cdot \mathrm{b}(\# \mid w_m) && (3.6) \end{aligned}$$

The n we will use in our work here is 2, meaning we will use the word sequence $w_1 w_2$ to predict w_3. This will be a *trigram* model.

Definition 10 (Trigram Language Model)**.** *Given the English words $w_1, w_2, ..., w_m$ and the sequence $w = w_1 w_2 ... w_m$, a* trigram language model *gives* $\mathrm{p}(w)$*, i.e. the probability of the words $w_1, w_2, ..., w_m$ occurring in exactly the sequence w in the English language.*

$$\mathrm{p}(w) = \mathrm{p}(w_1 \mid \#\#) \cdot \mathrm{p}(w_2 \mid \# w_1) \cdot (\prod_{i=3}^{m} \mathrm{p}(w_i \mid w_{i-2} w_{i-1})) \cdot \mathrm{p}(\# \mid w_{m-1} w_m)$$

Since we are only taking a limited history into account our model is a type of *Markov Chain* [Nor98]. What is usually the case with Markov chains is that for transitioning between states only the information of the current state is utilized without looking at previous states. In our case, we could group words into single states, essentially considering our limited word history as that states property.

We will go through the texts available counting how many times w_3 follows the sequence $w_1 w_2$. Let w_1, w_2, w_3, w' be words in language E. Using the maximum likelihood estimation we have:

$$\mathrm{p}(w_3 \mid w_1 w_2) = \frac{count(w_1 w_2 w_3)}{\sum\limits_{w' \in E} count(w_1 w_2 w')} \tag{3.7}$$

In reality E is limited to our available text. For example, in the Europarl corpus [Koe05], the number of trigrams starting with the phrase "the red" is 255, in 123 cases, this phrase is followed by the word "cross". So we can compute $\mathrm{p}(\text{cross}|\text{the red}) = \frac{123}{255} \simeq 0.547$.

3.2.2 Perplexity

As previously mentioned, language models are not perfect and can be very different considering the amount of text they were trained on and what order n-gram was used and so on. It is essential that we have a way to compare models and to measure their quality. This is where *perplexity* comes in.

When we defined a language model, we said that it should give higher probabilities to correct English sentences. In other words, probability mass shouldn't be wasted on sentences that would never occur in English language so there is more probability left over for more correct sentences.

We define the perplexity PP as a transformation of the cross-entropy H given by the formula below:

$$\mathrm{H}(\mathrm{p}, w_1 ... w_n) = -\frac{1}{n} \log \mathrm{p}(w_1 w_2 ... w_n) \tag{3.8}$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \log \mathrm{p}(w_i \mid w_1 ... w_{i-1}) \tag{3.9}$$

Then for perplexity we have:

$$\mathrm{PP(p)} = 2^{\mathrm{H(p)}} \tag{3.10}$$

The value for perplexity is an indication of how confused a language model is about the correctness of a sentence. Therefore, the lower the perplexity for a model, the better the model is at giving correct sentences. For n-gram language models with a higher order, the perplexity is generally less, but of-course there is a trade-off between perplexity and the data sparsity that comes with n-gram language models when n is too large.

The language model we implemented for this work, (for implementation details, see Chapter 4), has an average perplexity of 25.484 over all the sentences of the training data. This means, the confusion of the model is the same as having the choice between 25.484 different words for each word in a sentence.

Let us now look at an example of how perplexity is calculated for one sentence. The example sentence is: "It will, I hope, be examined in a positive light." The preprocessing done on the sentence removes punctuation, lower cases all the text and adds # s to the beginning and end of the sentence, (as indicated by Definition 10) and so the sentence we are calculating the perplexity for is: "# # it will i hope be examined in a positive light # ".

In Table 3.1, in the first column you will find the probability form of the trigrams in this sentence and in the second column the value of them that was calculated using the α-smoothing technique with $\alpha = 0.0001$, we will explain smoothing in the next section. In the bottom row of the table, the average of these values, which is the cross-entropy can be seen.

Having the cross-entropy, we can now calculate the perplexity for this sentence:

$$\mathrm{PP(p)} = 2^{H(\mathrm{p})} = 2^{4.403606271009044} = 21.1649660740799$$

The perplexity for this sentence is slightly less than our overall average.

3.2.3 Count Smoothing

It is highly unlikely that a language model will have seen every linguistically possible n-gram during training. This problem is specially concerning in models of higher order, where most n-grams occur only once or at most a few times in a large training corpus. Simply put, our *empirical count* which is our exact observation of n-grams in training, is insufficient to determine the *expected counts* of n-grams in unseen texts.

Table 3.1: The probabilities of trigrams and the cross-entropy for the sentence "# # it will i hope be examined in a positive light # ".

p(*it* | # #)	0.06077947197038887
p(*will* | # *it*)	0.03609514913620813
p(*i* | *it will*)	$8.735677965915312 \times 10^{-4}$
p(*hope* | *will i*)	0.33285914151997104
p(*be* | *i hope*)	0.0026647355022889224
p(*examined* | *hope be*)	0.013787104280743548
p(*in* | *be examined*)	0.20247675336426466
p(*a* | *examined in*)	0.04907237624458777
p(*positive* | *in a*)	0.007489755136792163
p(*light* | *a positive*)	0.010006683364620813
p(# | *positive light*)	0.3350513967129334
Average/cross-entropy	4.403606271009044

If we were to use the formula of maximum likelihood then we would end up with a zero for all unseen n-grams and therefore for any sentence containing an unseen n-gram we will also have zero. This would mean that if all the translations of a sentence contain unseen n-grams we cannot compare their quality. In the models with higher orders, n-grams seen only once or a few times would end up with very low probabilities if only the empirical count are considered.

This is where we introduce a *count smoothing* method for adjusting empirical counts to zero probability sentences in unseen texts.

Add-One Smoothing

In this method our aim is to get rid of zero probabilities for unseen n-grams. The easiest way to do this is by adding a fixed number to every count. If we add one to every count, we assume that every n-gram possible has been seen at least once. So if our maximum likelihood estimation of probability is $\mathrm{p}(w_1 w_2 w_3) = \frac{\mathrm{count}(w_1 w_2 w_3)}{\sum\limits_{w \in V} \mathrm{count}(w_1 w_2 w)}$ where V is the set of all possible words, it will be adjusted to:

$$\mathrm{p}(w_1 w_2 w_3) = \frac{\mathrm{count}(w_1 w_2 w_3) + 1}{\sum\limits_{w \in V} (\mathrm{count}(w_1 w_2 w) + 1)} \tag{3.11}$$

To make it a little neater, we bring the constant out of the sum:

$$p(w_1w_2w_3) = \frac{\text{count}(w_1w_2w_3) + 1}{(\sum_{w \in V} \text{count}(w_1w_2w)) + |V|} \tag{3.12}$$

The problem with adding one is that most possible n-grams shouldn't be given such a high probability as they will never be seen and the count is an undeserving one. So the solution then is to add a smaller number $\alpha < 1$ to the count:

$$p(w_1w_2w_3) = \frac{\text{count}(w_1w_2w_3) + \alpha}{(\sum_{w \in V} \text{count}(w_1w_2w)) + \alpha \cdot |V|} \tag{3.13}$$

The α used in practice can be determined with experimenting. Meaning trying different values and seeing which best optimizes perplexity in a certain test set.

3.3 The Translation Model

In this section we will describe two IBM translation models (Brown et al. 1993) and the methods of estimating their parameters. This section is mostly based on [BPPM93] and [Col11].

Recall that a translation model will try to compute the conditional probability $p(f \mid e)$, which is the probability of sentence f being the translation for sentence e.

Calculating this probability depends on a few parameters, for most of which the values must be estimated during *training*. We will initially guess values then use the EM algorithm to approach a local maximum in our *training data* set of translations.

Both models discussed here make the assumption that any reasonable length sentence for sentence f is equally likely. Model 1 then makes the further assumption that word order in the sentence does not affect $p(f \mid e)$. This further assumption is not made in model 2. The attempt is then made to make an alignment between the sentences, in other words, connect each position in f with a position in e. More formally, an *alignment* is a set of connections from words in one sentence to the words in the other. We use "words" freely here and the definition is not "words" as we have them in natural language but a larger concept of "words" as a (possibly empty) set of words from a natural language. An example of an alignment can be seen in Figure 3.1.

We will be looking for the probability that sentence f is the translation of sentence e and the "words" are connected using alignment a. Formally,

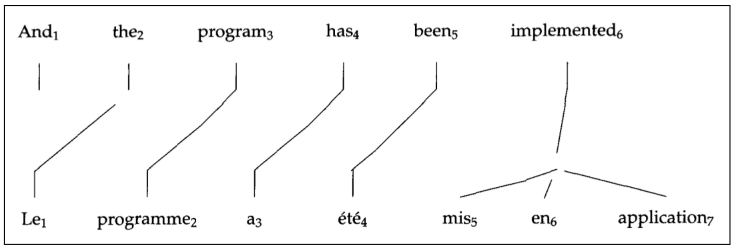

Figure 3.1: In this figure from [BPPM93], there is an alignment between an English sentence (top) and a French sentence (bottom). As can be seen 'And' in the English sentence is aligned with the 'empty word' and 'implemented' is aligned to a set of French words.

we will be looking at the conditional probability distribution $\mathrm{p}(F = f, A = a \mid E = e)$.

For any two sentences f and e there is a finite number of possible alignments. In order to calculate $\mathrm{p}(f \mid e)$, we need to sum over all the possible alignments between the two sentences. The reasoning behind the quantification is explained below. Therefore:

$$\mathrm{p}(f \mid e) = \sum_{\substack{a_i \in \{0,\ldots,|e|\} \\ i \in \{1,\ldots,|f|\}}} \mathrm{p}(f, a_i \mid e) \tag{3.14}$$

We will assume that the "words" in the alignments made here are either single words or empty, meaning each word in sentence f is connected to at most one word in sentence e. In doing this, we can label the "words" as sentence positions with the empty word as position zero.

Let e be a string with l positions (words) $e = e_1 e_2 \ldots e_l$ denoted by e_1^l, meaning words in sentence e from 1 to l, and f a string with m positions $f = f_1 f_2 \ldots f_m$ denoted by f_1^m. We can define an alignment of m values $a = a_1 a_2 \ldots a_m$,denoted by a_1^m , so that each a_i, $0 \leq i \leq m$ takes a value $0 \leq j \leq l$ such that if the i^{th} position in f is connected to the j^{th} position in e then $a_i = j$ and if it is not connected then $a_i = 0$.

We can write $\mathrm{p}(f, a \mid e)$ as the following product of conditional probabilities using the chain rule:

$$\mathrm{p}(f, a \mid e) = \mathrm{p}(m \mid e) \prod_{i=1}^{m} \mathrm{p}(a_i \mid a_1^{i-1}, f_1^{i-1}, m, e) \cdot \mathrm{p}(f_i \mid a_1^{i}, f_1^{i-1}, m, e) \tag{3.15}$$

Lets take a look at the intuition behind this formula that is given in [BPPM93]. We can view equation 3.15 in the following way; in generating a French string, f, together with an alignment, a, from an English string, e, we can first choose the length of f depending on e, namely $\text{p}(m \mid e)$. Next we make the choice of where our first alignment connection should be depending on our knowledge of e and the length we have for f, namely m. Finally, we can choose what identity to give the first word in f, depending on our knowledge of e, the length of f and the alignment connection we made between the first position of f and a position in e. We continue and do the same for the next positions in f, always taking into account our full knowledge of e and all previous choices made for alignment connections and identities in f.

While this is a good equation for understanding the intuition, it is a little difficult to work with when it comes to implementation. So, here we will give another way of expanding $\text{p}(f, a \mid e)$ that is given in [Col11].

In this version of $\text{p}(f, a \mid e)$, we will split our string e into its set of words and its length and have them separately in the expansion. In essence, this expansion will not have a different meaning from equation 3.15, but taking it farther into the next steps and then training will be easier to comprehend.

We will introduce some random variables here, so that we can later make some independence assumptions between them. So let L be a random variable for the length of the English sentence and let $E_1, ..., E_l$ be a sequence of random variables denoting the words in the English sentence. In the same manner, let M be a random variable for the length of the French sentence and let $F_1, ..., F_m$ be a sequence of random variables denoting the words in the French sentence. Also, let $A_1, ..., A_m$ be a sequence of random variables denoting the alignment variables. Our goal is to find $\text{p}(F_1 = f_1, ..., F_m = f_m, A_1 = a_1, ..., A_m = a_m \mid E_1 = e_1, ..., E_l = e_l, L = l, M = m)$.

The first thing we can do now, is to use the chain rule to get the following:

$$\begin{aligned}
&\text{p}(F_1 = f_1, ..., F_m = f_m, A_1 = a_1, ..., A_m = a_m \mid \\
&E_1 = e_1, ..., E_l = e_l, L = l, M = m) \\
&= \text{p}(A_1 = a_1, ..., A_m = a_m \mid E_1 = e_1, ..., E_l = e_l, L = l, M = m) &&(3.16)\\
&\times \text{p}(F_1 = f_1, ..., F_m = f_m \mid \\
&A_1 = a_1, ..., A_m = a_m, E_1 = e_1, ..., E_l = e_l, L = l, M = m) &&(3.17)
\end{aligned}$$

Next we will consider each of the two term 3.16 and 3.17 in turn. First for Term 3.16, we will write it as a product of all its separate probabilities, Equation 3.18, and then make the strong independence assumption that no A_i depends on the words of e and that all A_i are independent of A_j for $j \in \{1, ..., |e|\} \setminus \{i\}$. Therefore they solely depend on the random variable M and L, as is shown in equation 3.19.

$$
\begin{aligned}
&\mathrm{p}(A_1 = a_1, ..., A_m = a_m \mid E_1 = e_1, ..., E_l = e_l, L = l, M = m) \\
&= \prod_{i=1}^{m} \mathrm{p}(A_i = a_i \mid A_1 = a_1, ..., A_{i-1} = a_{i-1}, E_1 = e_1, ..., E_l = e_l, \\
&L = l, M = m) \qquad (3.18) \\
&= \prod_{i=1}^{m} \mathrm{p}(A_i = a_i \mid L = l, M = m) \qquad (3.19)
\end{aligned}
$$

Our final assumption is that $\mathrm{p}(A_i = a_i \mid L = l, M = m) = q(a_i \mid i, l, m)$ and that all $q(a_i \mid i, l, m)$ are parameters for our translation model. The added i within the parameter $q(a_i \mid i, l, m)$ is used to separate the random variables A_i for each individual i.

Now we do the same for Term 3.17, namely we will write it as a product of probabilities, Equation 3.20 , and then make the strong assumption that F_i only depends on E_{a_i}, meaning it only depends on the English word that is aligned with F_i and is completely independent of L and M. This is shown by equation 3.21.

$$
\begin{aligned}
&\mathrm{p}(F_1 = f_1, ..., F_m = f_m \mid \\
&A_1 = a_1, ..., A_m = a_m, E_1 = e_1, ..., E_l = e_l, L = l, M = m) \\
&= \prod_{i=1}^{m} \mathrm{p}(F_i = f_i \mid F_1 = f_1, ..., F_{i-1} = f_{i-1}, A_1 = a_1, ..., A_m = a_m, \\
&\quad E_1 = e_1, ..., E_l = e_l, L = l, M = m) \qquad (3.20) \\
&= \prod_{i=1}^{m} \mathrm{p}(F_i = f_i \mid E_{a_i} = e_{a_i}) \qquad (3.21)
\end{aligned}
$$

As we did before, we now make one more assumption that for all i, $\mathrm{p}(F_i = f_i \mid E_{a_i} = e_{a_i}) = t(f_i \mid e_{a_i})$ and that all $t(f_i \mid e_{a_i})$ is are parameters in our translation model. In the following, it will become clear exactly what these parameters correspond to from equation 3.15.

Considering all these assumptions we have:

$$
\mathrm{p}(f, a \mid e) = \prod_{i=0}^{m} q(a_i \mid i, l, m) \cdot t(f_i \mid e_{a_i}) \qquad (3.22)
$$

3.3.1 Model 1

Equation 3.15 is much simplified in model 1. The assumptions in model 1 are the following:

- p$(m \mid e)$ does not depend on e and m so we take p$(m \mid e) = \epsilon$ as some small fixed number

- $q(a_i \mid i, l, m) = $ p$(a_i \mid a_1^{i-1}, f_1^{i-1}, m, e)$ called the *alignment probability* depends only on the length of e,i.e. l and is therefore $\frac{1}{l+1}$ and,

- p$(f_i \mid a_1^i, f_1^{i-1}, m, e)$ is dependent on f_i and e_{a_i} and so we have p$(f_i \mid a_1^i, f_1^{i-1}, m, e) = t(f_i \mid e_{a_i})$ which we call the *translation probability* of f_i given e_{a_i}.

Putting these assumptions within the equation we have:

$$\text{p}(f, a \mid e) = \frac{\epsilon}{(l+1)^m} \prod_{i=1}^{m} t(f_i \mid e_{a_i}) \tag{3.23}$$

We can use equation 3.14 and sum over the alignments so that we have an equation for p$(f \mid e)$. To specify the alignment sum we must specify the values of a_i for all $0 \le i \le m$. Recall that their values can be from 0 to l. Thus:

$$\text{p}(f \mid e) = \frac{\epsilon}{(l+1)^m} \sum_{a_1=0}^{l} \dots \sum_{a_m=0}^{l} \prod_{i=1}^{m} t(f_i \mid e_{a_i}) \tag{3.24}$$

In this model there really is only one parameter we must estimate, namely t. To train our model we will use a corpus containing a set of n translations $\{(f^{(1)}, e^{(1)}), (f^{(2)}, e^{(2)}), ..., (f^{(n)}, e^{(n)})\}$ which we will denote by $\{f^{(k)}, e^{(k)}\}_{k=1}^{n}$, where k is used to indicate the number of the translation.

Now, we can see that there is a problem, namely, our corpus contains no alignment information. The fact is, it would be too costly and eventually ineffective if we were to annotate all alignments manually for every corpus. So the problem then is how to estimate the parameter(s) without having all the data. This is why we will have to use the *Expectation-Maximization algorithm.*

3.3.2 Expectation-Maximization Algorithm

Here, we will give a brief definition of the *Expectation-Maximization algorithm* or EM algorithm then specify how we will use it in practice.

The goal of the EM algorithm is to find maximum-likelihood estimates where one does not have all the necessary data. The EM algorithm is given an *incomplete data corpus* and aims to map it to a *complete data corpus* where it is known how we can perform the maximum likelihood estimation. It does this by what is known as the *symbolic analyzer*.

In the following, allow $\mathcal{X}$ and $\mathcal{Y}$ to be two countable non-empty sets. We will define a *symbolic analyzer* as a function $\mathcal{A}\colon \mathcal{Y} \to 2^{\mathcal{X}}$ and name $\mathcal{A}(y)$ for all $y \in \mathcal{Y}$ as *sets of analyzes*. The sets $\mathcal{A}(y)$ form a partition on $\mathcal{X}$ meaning they are pairwise disjoint and their union is complete, i.e. $\mathcal{X} = \sum\limits_{y\in\mathcal{Y}} \mathcal{A}(y)$.

Here we can conclude that for each $x \in \mathcal{X}$ exists a unique $y \in \mathcal{Y}$ such that x is an analysis of y and y is then called the *yield* of x, i.e. $y = \text{yield}(x)$ iff $x \in \mathcal{A}(y)$.

If we now pair our symbolic analyzer with a probability distribution p on $\mathcal{X}$ we will have what is called a *statistical analyzer*. This analyzer will help us induce probabilities on the set $\mathcal{Y}$ as follows. For all $y \in \mathcal{Y}$:

$$p(y) = \sum_{x\in\mathcal{A}(y)} p(x)$$

Now, we can see what we need to input into our EM algorithm and how it all comes together. Obviously we need our incomplete data corpus and our symbolic analyzer. Then we need our *complete probability model* $\mathcal{M} \subseteq \mathcal{M}(\mathcal{X})$ (Definition 6). This means that an incomplete data model $\mathcal{M} \subseteq \mathcal{M}(\mathcal{Y})$ is implied. It is induced by the symbolic analyzer and the complete probability model. The symbolic analyzer becomes a statistical analyzer and then induces instances of the incomplete probability model as such:

$$p\colon \mathcal{Y} \to [0,1] \text{ and } p(y) = \sum_{x\in\mathcal{A}(y)} p(x)$$

Finally we also need a random starting point p_0 of the complete probability model.

Now we move onto the procedure of the EM algorithm. The procedure has two main steps, the *expectation step (E)* and the *maximization step (M)*. The procedure is as follows:

- for each $i = 1, 2, 3, \ldots$ do
    - $q = p_{i-1}$
    - **E-step:** compute the complete data corpus $f_q\colon \mathcal{X} \to \mathcal{R}$ expected by q

$$f_q(x) = f(y) \cdot q(x \mid y) \text{ where } y = \text{yield}(x)$$

- **M-step:** compute a maximum-likelihood estimate p' of $\mathcal{M}$ on f_q

$$L(f_q; p') = \max_{p \in \mathcal{M}} L(f_q, p)$$

- $p_i = p'$

- end for each i

- print $p_0, p_1, p_2, ...$

3.3.3 Training

Now we will describe how to use this algorithm for the IBM model 1.

Lets simplify things first and imagine our data are not incomplete and we do indeed have alignment variables for all our translations. Let us define our corpus as follows. Let $\mathcal{F}, \mathcal{E}, \mathcal{A}$ be three non-empty sets denoting the sets of French sentences, English sentences and possible alignments respectively. We define corpus $\mathcal{C}$ as $\mathcal{C}\colon \mathcal{F} \times \mathcal{E} \times \mathcal{A} \to \mathcal{R}_{\geq 0}$. For easier use we define the set $T = \{f^{(k)}, e^{(k)}, a^{(k)}\}_{k=1}^{n}$ where $\mathcal{C}(t) = \begin{cases} 1 & \text{if } t \in T \\ 0 & \text{otherwise} \end{cases}$. Using set T as our working translation set, we have for each $t \in T$ of k translations:

- a French sentence $f^{(k)} = f_1^{(k)}, ..., f_{m_k}^{(k)}$

- an English sentence $e^{(k)} = e_1^{(k)}, ..., e_{l_k}^{(k)}$ and

- an alignment $a^{(k)} = a_1^{(k)}, ..., a_{m_k}^{(k)}$

Given all these, calculations are very simple but we will describe this case keeping in mind our intention to develop it further later on. So we first define the following:

- $c_T(e, f)$ will be the number of times there is an alignment between word e and word f

- then $c_T(e)$ is the number of times e has an alignment with any french word

- $c_T(j \mid i, l, m)$ is the number of times an English sentence of length l is the translation of a French sentence of length m where word i in French is aligned to word j in English

- then $c_T(i, l, m)$ is the number of times we have a French sentence of length m and an English sentence of length l

Having defined these we can go back to the parameters of model 1. Even though for model 1 only one parameter needs estimating we will describe the other in interest of developing this for model 2 later on.

So now we define our translation probability and alignment probability of before as follows:

$$t_T(f \mid e) = \frac{c_T(e, f)}{c_T(e)} \tag{3.25}$$

$$q_T(j \mid i, l, m) = \frac{c_T(j \mid i, l, m)}{c_T(i, l, m)} \tag{3.26}$$

Now in the case of a training corpus with alignment variables, the objective will be to count all the above and calculate. So we will have to go through all translations and find all possible word pairs that could be aligned, i.e. for a triple (k, i, j) where $1 \leq k \leq n$, $1 \leq i \leq m_k$ and $1 \leq j \leq l_k$ we have $a_i^{(k)} = j$ if the two words are aligned. We will define a function $\delta(k, i, j)$ to help us calculate these counts, this function is defined as follows for the complete case:

$$\delta(k, i, j) = \begin{cases} 1 & \text{if } a_i^{(k)} = j \\ 0 & \text{otherwise} \end{cases} \tag{3.27}$$

Meaning that, as you can see in figure 3.2, the purpose will be simply to increment if there is an alignment and not to, if not.

Now for the case in which we have no alignment variables. We will now use the EM algorithm described above. Naturally, this means an iterative algorithm that will run for a predefined number of times, that is R in figure 3.3, or until it converges (that is not shown in the figure for better clarity in the pseudo-code), whichever happens first. In each iteration, our model parameter, namely t, will be updated for the next step using its current value. Our starting point will be random.

Our training set will be $T = \{f^{(k)}, e^{(k)}\}_{k=1}^{n}$ so this will be the incomplete data corpus we will be trying to complete. To do that we need to somehow add the alignment variables. This can be done probabilistically.

We will first redefine $\delta(k, i, j)$ for the incomplete case as follows:

$$\delta(k, i, j) = \frac{q(j \mid i, l_k, m_k) \cdot t(f_i^{(k)} \mid e_j^{(k)})}{\sum_{j=0}^{l_k} q(j \mid i, l_k, m_k) \cdot t(f_i^{(k)} \mid e_j^{(k)})} \tag{3.28}$$

```
Input: A training corpus $T = \{f^{(k)}, e^{(k)}, a^{(k)}\}_{k=1}^{n}$ where $f^{(k)} = f_1^{(k)}...f_{m_k}^{(k)}$,
  $e^{(k)} = e_1^{(k)}...e_{l_k}^{(k)}$, $a^{(k)} = a_1^{(k)}...a_{m_k}^{(k)}$.

  set all counts $c_T(...) = 0$
  for $k = 1...n$ do
      for $i = 1...m_k$ do
          for $j = 0...l_k$ do

              if $a_i^{(k)} = j$ then
                  $\delta(k,i,j) = 1$
              else
                  $\delta(k,i,j) = 0$
              end if

              $c_T(e_j^{(k)}, f_i^{(k)}) \leftarrow c_T(e_j^{(k)}, f_i^{(k)}) + \delta(k,i,j)$
              $c_T(e_j^{(k)}) \leftarrow c_T(e_j^{(k)}) + \delta(k,i,j)$
              $c_T(j \mid i,l,m) \leftarrow c_T(j \mid i,l,m) + \delta(k,i,j)$
              $c_T(i,l,m) \leftarrow c_T(i,l,m) + \delta(k,i,j)$

          end for
      end for
  end for

Output: $\forall e, f \in T$ set $t_T(f \mid e) = \frac{c_T(e,f)}{c_T(e)}$ and $\forall j,i,l,m$ set $q_T(j \mid i,l,m) = \frac{c_T(j|i,l,m)}{c_T(i,l,m)}$
```

Figure 3.2: In this figure, we can see how the parameters are estimated with a complete corpus that includes alignment variables.

This is in fact giving us a probability of alignment variable $a_i^{(k)}$ being equal to j. To see this let us note the following identity:

$$\mathrm{p}(A_i = j \mid e_1 ... e_l, f_1 ... f_m, m) = \frac{q(j \mid i, l, m) \cdot t(f_i \mid e_j)}{\sum\limits_{j=0}^{l} q(j \mid i, l, m) \cdot t(f_i \mid e_j)} \tag{3.29}$$

In this way the alignment variables will be filled probabilistically using current parameter estimates in each iteration.

Now, of course, for model 1, we stated that the alignment probability depends only on the English sentence length, so we modify our δ as follows:

$$\delta(k, i, j) = \frac{\frac{1}{(l^{(k)}+1)} t(f_i^{(k)} \mid e_j^{(k)})}{\sum\limits_{j=0}^{l_k} \frac{1}{(l^{(k)}+1)} t(f_i^{(k)} \mid e_j^{(k)})} = \frac{t(f_i^{(k)} \mid e_j^{(k)})}{\sum\limits_{j=0}^{l_k} t(f_i^{(k)} \mid e_j^{(k)})} \tag{3.30}$$

The algorithm for estimating $t(f \mid e)$ is shown in figure 3.3.

In this version of the EM algorithm, one can say that the E-step consists of the inner **for** loops k, i and j and the M-step is the outer loop r.

The good thing about IBM model 1 is, that despite its naivety in its assumptions, using the algorithm in figure 3.3, it will always converge to the global maximum and therefore provides a good starting point for model 2 [BPPM93].

3.3.4 Model 2

The mathematical form used for model 1, provides us with a unique global maximum after a series of EM iterations, so the derived parameter does not depend on our initial guess. We will now use the results of model 1 as an initial estimate in model 2.

Our algorithm and conditions will be much the same as before with $T = \{f^{(k)}, e^{(k)}\}_{k=1}^{n}$ and our δ returning to its original form in equation 3.28.

As seen in figure 3.4 iterations will happen for a total of R times or until convergence (not shown in pseudo-code). Unlike model 1, model 2 may have many local maximums and therefore the possibility of converging to a local maximum rather than the global maximum exists.

Having these parameters we can can now calculate the conditional probability of IBM model 2 as follows:

$$\mathrm{p}(f_1 ... f_m, a_1 ... a_m \mid e_1 ... e_l, m) = \prod_{i=1}^{m} q_T(a_i \mid i, l, m) \cdot t_T(f_i \mid e_{a_i}) \tag{3.31}$$

```
Input: A training corpus $T = \{f^{(k)}, e^{(k)}\}_{k=1}^{n}$ where $f^{(k)} = f_1^{(k)}...f_{m_k}^{(k)}$, $e^{(k)} = e_1^{(k)}...e_{l_k}^{(k)}$.

  Initialize $t_T(f \mid e)$ randomly.
  for $r = 1...R$ do
      set all counts $c_T(...) = 0$
      for $k = 1...n$ do
          for $i = 1...m_k$ do
              for $j = 0...l_k$ do

                  $\delta(k,i,j) = \frac{t_T(f_i^{(k)}|e_j^{(k)})}{\sum_{j=0}^{l_k} t_T(f_i^{(k)}|e_j^{(k)})}$
                  $c_T(e_j^{(k)}, f_i^{(k)}) \leftarrow c_T(e_j^{(k)}, f_i^{(k)}) + \delta(k,i,j)$
                  $c_T(e_j^{(k)}) \leftarrow c_T(e_j^{(k)}) + \delta(k,i,j)$

              end for
          end for
      end for
      $\forall e, f \in T$ set $t_T(f \mid e) = \frac{c_T(e,f)}{c_T(e)}$
  end for

Output: parameter $t_T(f \mid e)$
```

Figure 3.3: In this figure, we can see how the parameter $t_{\text{TS}}(f \mid e)$ is estimated using a version of the EM algorithm.

```
Input: A training corpus $T = \{f^{(k)}, e^{(k)}\}_{k=1}^{n}$ where $f^{(k)} = f_1^{(k)}...f_{m_k}^{(k)}$, $e^{(k)} = e_1^{(k)}...e_{l_k}^{(k)}$.

  Initialize $t_T(f \mid e)$ to the value gained from IBM model 1. Initialize $q_T(j \mid i, l, m)$ randomly.
  for $r = 1...R$ do
      set all counts $c_T(...) = 0$
      for $k = 1...n$ do
          for $i = 1...m_k$ do
              for $j = 0...l_k$ do

                  $\delta(k,i,j) = \frac{q_T(j|i,l_k,m_k)t_T(f_i^{(k)}|e_j^{(k)})}{\sum_{j=0}^{l_k} q_T(j|i,l_k,m_k)t_T(f_i^{(k)}|e_j^{(k)})}$
                  $c_T(e_j^{(k)}, f_i^{(k)}) \leftarrow c_T(e_j^{(k)}, f_i^{(k)}) + \delta(k,i,j)$
                  $c_T(e_j^{(k)}) \leftarrow c_T(e_j^{(k)}) + \delta(k,i,j)$
                  $c_T(j \mid i,l,m) \leftarrow c_T(j \mid i,l,m) + \delta(k,i,j)$
                  $c_T(i,l,m) \leftarrow c_T(i,l,m) + \delta(k,i,j)$

              end for
          end for
      end for
      $\forall e, f \in T$ set $t_T(f \mid e) = \frac{c_T(e,f)}{c_T(e)}$ and $\forall i,j,l,m$ set $q_T(j \mid i,l,m) = \frac{c_T(j|i,l,m)}{c_T(i,l,m)}$
  end for

Output: parameters $t_T(f \mid e)$ and $q_T(j \mid i, l, m)$
```

Figure 3.4: In this figure, we can see how the parameters $t(f \mid e)$ and $q(j \mid i, l, m)$ are estimated for IBM model 2 using a version of the EM algorithm.

To find $\mathrm{p}(f \mid e)$, which is our primary goal, we now need to sum up over the alignments to get:

$$\mathrm{p}(f \mid e) = \sum_{a_1=0}^{l} \dots \sum_{a_m=0}^{l} \prod_{i=1}^{m} q_T(a_i \mid i, l, m) \cdot t_T(f_i \mid e_{a_i}) \tag{3.32}$$

We can simplify this slightly by replacing a_i by the counter j and it can easily be shown that the below equation is equivalent to the above. An example for this can be found above Equation 16 in [BPPM93].

$$\mathrm{p}(f \mid e) = \prod_{i=1}^{m} \sum_{j=0}^{l} q_T(j \mid i, l, m) \cdot t_T(f_i \mid e_j) \tag{3.33}$$

3.4 Decoding

The following section is based on the work of Ye-Yi Wang and Alex Waibel, [WW97]. In this section we will discuss *decoding* for this statistical machine translation system. What is meant by the term decoding, is the search process performed on a given translation model and language model to find the best approximated translation for a given sentence. It was proven by K. Knight, [Kni99], that performing this process and decoding for such translation systems is NP-complete, so any decoding algorithm will strive to find the best approximation of a translation it can.

Recall that our statistical machine translation system is trying to find $\operatorname*{argmax}_{e \in E} \mathrm{p}(e) \cdot \mathrm{p}(e \mid f)$. In previous sections we covered how to build a language model $\mathrm{p}(e)$, and a translation model $\mathrm{p}(e \mid f)$. Now we will discuss an algorithm that can help us find the best e.

Decoding is important because it directly impacts the performance of the entire system and the quality of the resulting translations. The algorithm we will discuss here is the stack decoding algorithm presented by Wang and Waibel, [WW97].

The basic idea of a stack algorithm is to start with a guess or hypothesis and a score for this hypothesis, both notions will be further explained below. Then we extend the hypothesis by adding words and re-scoring, always keeping the hypothesis with the highest score, until we have a complete sentence.

In other words the algorithm can be written as:

1. Initialize the stack with a null hypothesis.

2. Pop the hypothesis with the highest score as `current-hypothesis`.

3. If `current-hypothesis` is a complete sentence, then output it and terminate.
4. For all words w in V from Equation 3.12, extend `current-hypothesis` by adding w to its end and compute new score. Insert all new hypothesis into stack.
5. Go to 2.

3.4.1 Scoring

Let us define a hypothesis H as $H = l\colon e_1e_2...e_k$ where l is the length of sentence e and $e_1e_2...e_k$ is the sequence of the first k words in e. The score for H namely f_H is composed of two parts, $f_H = g_H + h_H$, where g_H is the prefix score or the score for what we have so far, namely $e_1...e_k$ and h_H is the heuristic score for words that have yet to be appended to complete the sentence, namely $e_{k+1}...e_l$.

This score needs to give us an idea of how well we are building our translation. To this end, we need to calculate separately, the probability given by the language model for words so far, the probability given by the language model for potential words to come and the same for the translation model. This makes four different values and since we would like to be able to just sum them up neatly at the end, we will use the logarithms of these models. The process of finding f_H begins as follows and later it can be seen how we split f_H into its two composing parts:

$$\begin{aligned} f_H &= \underset{e\in E}{\operatorname{argmax}}(\mathrm{p}(e)\cdot \mathrm{p}(e \mid f)) \\ &= \underset{e\in E}{\operatorname{argmax}}(\log(\mathrm{p}(e)\cdot \mathrm{p}(e \mid f))) \\ &= \underset{e\in E}{\operatorname{argmax}}(\log \mathrm{p}(e) + \log \mathrm{p}(e \mid f)) \end{aligned}$$

This can be done because the logarithm function is monotone increasing. Next we will replace $\mathrm{p}(e)$ with a condensed version of the formula from Definition 10. Please note that n is the order of n-gram used in the language model and the first $n-1$ probabilities must be listed separately to avoid negative js, in reality they are the probabilities containing sentence markers (#). However, here we choose to write it as one product for more readability. We also replace $\mathrm{p}(f \mid e)$ with its equivalent in Equation 3.33. So we now have:

$$
\begin{aligned}
f_H &= \operatorname*{argmax}_{e\in E}\Big(\log\prod_{j=0}^{l}\mathrm{p}(e_j\mid e_{j-n+1}...e_{j-1})+\log\prod_{i=1}^{m}\sum_{j=0}^{l}q(j\mid i,l,m)\cdot t(f_i\mid e_j)\Big)\\
&= \operatorname*{argmax}_{e\in E}\Big(\sum_{j=0}^{l}\log\mathrm{p}(e_j\mid e_{j-n+1}...e_{j-1})\\
&+\sum_{i=1}^{m}\log\sum_{j=0}^{l}q(j\mid i,l,m)\cdot t(f_i\mid e_j)\Big)
\end{aligned}
$$

What we have so far is for a complete sentence, but our hypotheses are incomplete, so at this point we need to split our models into the first k words which are known, and the next $k+1$ to l words which are yet to be added to the sentence. Thus we continue as follows:

$$
\begin{aligned}
f_H = \operatorname*{argmax}_{e\in E}(&\overbrace{\sum_{j=0}^{k}\log\mathrm{p}(e_j\mid e_{j-n+1}...e_{j-1})}^{(a)}+\overbrace{\sum_{j=k+1}^{l}\log\mathrm{p}(e_j\mid e_{j-n+1}...e_{j-1})}^{(b)}\\
&+\sum_{i=1}^{m}\log(\overbrace{\sum_{j=0}^{k}q(j\mid i,l,m)\cdot t(f_i\mid e_j)}^{(c)}+\overbrace{\sum_{j=k+1}^{l}q(j\mid i,l,m)\cdot t(f_i\mid e_j)}^{(d)}))
\end{aligned}
\tag{3.34}
$$

Parts (a) and (c) are represent the known part of our sentence and will be used in calculating the g_H part of our score. Parts (b) and (d) cannot be calculated as the words are not yet known. Therefore, we need to define heuristics that can approximate these values well and the heuristics for these two parts will be used in calculating h_H for our score.

Calculating the score g_H

To better understand the scoring method let us now interpret the IBM translation model 2 in the following way. Each word e_j in the hypothesis, contributes the amount $q(j\mid i,l,m)\cdot t(f_i\mid e_j)$ to the probability of f_i. We will use $S_H(i)$ to denote the probability for the word f_i contributed by the words currently in the hypothesis. Therefore we have:

$$S_H(i)=\sum_{j=0}^{k}q(j\mid i,l,m)\cdot t(f_i\mid e_j) \tag{3.35}$$

Now, we replace the inner sum of Equation 3.33 with $S_H(i)$ and take the logarithm as shown, and we have $\sum\limits_{i=1}^{m} \log S_H(i)$ which is the contribution of the translation model to the g_H part of our score.

We now have to consider the contribution of the language model to the hypothesis score. So, we also take the logarithm of the language model probability of the hypothesis and we get:

$$\begin{aligned} g_H = &\sum_{i=0}^{m} \log S_H(i) + \\ &\sum_{j=0}^{k} \log \mathrm{p}(e_j \mid e_{j-n+1} ... e_{j-1}) \end{aligned} \tag{3.36}$$

where n is the order of the n-gram model. To calculate g_H, in each step, we want to be able to use the g-score of its parent hypothesis which we will call P. In other words we would like to calculate the g-score for a hypothesis $H = l\colon e_1 ... e_k$ using the hypothesis $P = l\colon e_1 ... e_{k-1}$, therefore the value $S_H(i)$ is separated into the value for the parent hypothesis and the contribution of e_k and so we have $S_H(i) = S_P(i) + q(k \mid i, l, m) \cdot t(f_i \mid e_k)$. The same is applied to the language model probability and our new g_H then is:

$$
\begin{aligned}
g_H &= \sum_{j=0}^{k-1} \log \mathrm{p}(e_j \mid e_{j-n+1}...e_{j-1}) + \log \mathrm{p}(e_k \mid e_{k-n+1}...e_{k-1}) \\
&+ \sum_{i=1}^{m} \log \overbrace{(\sum_{j=0}^{k-1} q(j \mid i, l, m) t(f_i \mid e_j)}^{S_P(i)} + q(k \mid i, l, m) t(f_i \mid e_k)) \\
&= \sum_{j=0}^{k-1} \log \mathrm{p}(e_j \mid e_{j-n+1}...e_{j-1}) + \log \mathrm{p}(e_k \mid e_{k-n+1}...e_{k-1}) \\
&+ \sum_{i=1}^{m} \log S_P(i) \cdot (1 + \frac{q(k \mid i, l, m) t(f_i \mid e_k)}{S_P(i)}) \\
&= \overbrace{\sum_{j=0}^{k-1} \log \mathrm{p}(e_j \mid e_{j-n+1}...e_{j-1})}^{g_P} + \log \mathrm{p}(e_k \mid e_{k-n+1}...e_{k-1}) \\
&+ \overbrace{\sum_{i=1}^{m} \log S_P(i)}^{g_P} + \sum_{i=1}^{m} \log(1 + \frac{q(k \mid i, l, m) t(f_i \mid e_k)}{S_P(i)}) \\
&= g_P + \log \mathrm{p}(e_k \mid e_{k-n+1}...e_{k-1}) \\
&+ \sum_{i=1}^{m} \log(1 + \frac{q(k \mid i, l, m) t(f_i \mid e_k)}{S_P(i)})
\end{aligned} \tag{3.37}
$$

To avoid having a zero denominator in the initial stages of the algorithm, the null hypothesis H_0 will have a small value π as its score, rather than zero.

Calculating the heuristic h_H

The first thing to consider for our heuristic function, is how far to extend a hypothesis. If we underestimate the value of extending our hypothesis, the search may end too early without giving an optimal result, and if we overestimate and extend it too much, the algorithm will waste a lot of time to safeguard optimality. So a balance must be reached here.

It is stated by N. Nilsson, [Nil71], that in order to guarantee an optimal search result the heuristic function must be an upper-bound of the score for all possible extensions $e_{k+1}...e_l$.

Calculating h_H will follow a similar pattern to calculating g_H, namely, we must estimate the contributions of the language model and the translation model in the extension.

For estimating the heuristic score for the language model, h_H^{LM}, we will use the negative of the perplexity of our language model over the training data, PP_{train}. This will be the logarithm of the average probability of predicting a new word in the extension. We assume that, on average, PP_{train} overestimates the likelihood of the next word. We will also use a constant C to help avoid underestimating the likelihood of the next word. So we will define h_H^{LM} as follows:

$$h_H^{LM} = -(l-k) \cdot \mathrm{PP}_{train} + C \tag{3.38}$$

We can see in Equation 3.38, that the dominating expression is determined by k. Meaning, that when k is much smaller than l, the expression $-(l-k) \cdot \mathrm{PP}_{train}$ plays a more significant role and h_H^{LM} has a value closer to the average perplexity. However, when k gets closer to l and there are only a few words left, the language model probability for the words left may be much higher than average and so we need to choose a constant that will help us avoid underestimation. The constant that was chosen is $C = \mathrm{PP}_{train} + \log \mathrm{p}_{max}$, where p_{max} is the maximum n-gram probability in the language model.

Now we move onto calculating the heuristic for the contribution of the translation model. To this end, we will introduce a variable $v_{jl}(i)$ to denote the maximum probability of f_i given any possible word from V at any position between j and l, where V is the set of all possible different words in our language. Therefore we have:

$$v_{jl}(i) = \max_{\substack{j \le k \le l \\ e \in V}} q(k \mid i, l, m) \cdot t(f_i \mid e) \tag{3.39}$$

It is enough to calculate $v_{jl}(i)$ once for a given target sentence as it is independent of hypotheses. Now, we need to calculate the maximum that a new word can contribute to the likelihood of the i-th target word so we take the logarithm of the variable $v_{jl}(i)$ when $j = k+1$ and subtract from it the logarithm of the probability mass for the i-th word given by words in the hypothesis, namely $S_H(i)$, so we have:

$$\log(v_{(k+1)l}(i)) - \log S_H(i) \tag{3.40}$$

Expression 3.40 can have a negative result so we will keep the maximum between it and 0. Our final h_H when $k < l$ is then:

$$\begin{aligned} h_H = & \sum_{i=1}^{m} \max\{0, \log(v_{(k+1)l}(i)) - \log S_H(i)\} \\ & - (l-k) \cdot \mathrm{PP}_{train} + C \end{aligned} \tag{3.41}$$

When $k = l$, no more words can be added to extend the hypothesis and since h_H only considers words yet to be added, it is clear that $h_H = 0$ for this case. By adding g_H and h_H we can compute the score f_H for a hypothesis $H = l \colon e_1 ... e_k$:

$$
\begin{aligned}
f_H &= g_H + h_H \\
&= g_P + \log \mathrm{p}(e_k \mid e_{k-n+1} ... e_{k-1}) \\
&+ \sum_{i=1}^{m} \log(1 + \frac{q(k \mid i, l, m) t(f_i \mid e_k)}{S_P(i)}) \\
&+ \sum_{i=1}^{m} \max\{0, \log(v_{(k+1)l}(i)) - \log S_H(i)\} && (3.42) \\
&- (l - k) \cdot \mathrm{PP}_{\mathit{train}} + C && (3.43)
\end{aligned}
$$

Expression 3.42 and Expression 3.43, represent approximations of parts (d) and (b) of Equation 3.34 respectively.

Chapter 4

Implementation Details and Results

In this chapter we will go through the programming techniques used in order to implement our statistical translation system. All the programming is done using the programming language Java and the finished programs were run on a personal laptop. All implementation code and some files can be found at this web address: (https://www.dropbox.com/sh/hc8wffhjhk32h0c/SA9UU5TUaR)

4.1 The Language Model

In this section we will specify how we created our trigram file from Europarls English data files. We decided to create our own trigram file rather than use available ones online in order to have better control over the exact forms our trigrams take. Of course, this means that we are working from a much more limited data set.

4.1.1 Creating our own Trigram File

The main thing to think about when writing a program to create our trigram file is the amount of data we have to deal with. The task appears to be a relatively simple one; when given a file containing one sentence per line, split the line into its trigrams, adding special characters to the beginning and end of the sentence, count them, and write the results in a new file.

It may seem at first glance that the task can be done in any number of ways and that is true when working with only a small amount of data. But some methods cost more time and memory as the amount of data gets larger. To help deal with these large amounts of data we will be using a Java structure known as a `HashMap`.

A `HashMap` is an object that stores pairs of keys and values in a table within the RAM. The key is unique and this structure has a constant time performance for its basic `get` and `put` functions, retrieving and adding to the `HashMap` respectively. The only problem with using `HashMap` is again related to the amount of data. The more data, the more memory space needed. In our particular case we had to manually increase the amount of memory allocated to our JVM and stack in order to store the full `HashMap` in memory. We used the Europarl English text v7. [Koe05] to make this trigram file. Our text file was spilt into lines, where each line was a complete sentence.

The algorithm we used was as follows:

- for every line in the file do the following:
    - split the sentence into a `List` of words
    - for each word make the trigram of previous and/or following words and place into a second `List`
    - for each trigram in the `List` do the following:
        - create the `key` using the formula: `word1+" "+word2+" "+word3`
        - search for the `key` in the `HashMap`
        - `if` the `key` does not exist, then `put` it in the `HashMap` with a `value` of `1`
        - `if` the `key` does exist, then `put` in the `key` with the new `value` of the old `value` plus one
- write all `key` and `value` pairs to a new file

The file that is built is in the format `frequency` followed by the three words separated by tabs. This algorithm has linear time complexity with respect to the number of words.

Our final file of trigrams in this case contained 15,484,624 trigrams and their frequencies.

4.1.2 Creating our Language Model Database

To create our language model database, we read the trigram file we made in the previous, place it once more in a different `HashMap` and use that to calculate the maximum likelihood with and without smoothing. The final result is then placed in a Lucene Database for faster later retrieval.

The Lucene Database [luc11], is a full-featured text search engine library written entirely in Java. The database contains an index of documents. For our purposes, each document added to the index will be of one trigram and its frequency taken directly from our trigram file, the calculated maximum likelihood without smoothing and the maximum likelihood with a number of different alphas used for add-α smoothing.

To be able to calculate maximum likelihood for each trigram t, we will need two main components that we do not have in our trigram file. The first component is the total of all frequencies of trigrams that have the same first two words as t. The second component we will need, is the total number of those trigrams with the same first two words as t. The latter is used during smoothing.

Let us explain the difference between the two with a small example. If a file contained only the two trigrams (x, y, x) with frequency a and (x, y, z) with frequency b, then it can be seen that both have the same first two words so the total frequency for these trigrams is $a + b$ while the total number is 3. The reason the total number is 3 is because with the given words in this file, we can have the trigram (x, y, y) with the frequency 0 and this would make our $|V|$ from Equation 3.12 equal 3. This $|V|$ is the total number and is in fact equal to the number of different words the file contains because the number of different words in the file determine the number of possible trigrams with the same first two words. In the case of our file, the number of different words in our V is 277,875.

Recall that our trigram file contained the frequencies followed by the three words. Reading from that file we get for each trigram t, the three words `word1,word2,word3` and the frequecy `f`. Now we will create a `HashMap` using the first two words of our trigrams as the `key` and an `Integer` denoting total frequency as the value. The algorithm for calculating this total is as follows:

```
- For every trigram t do:
    - Set key as word1+" "+word2
    - if key is in HashMap then value is set to a new Integer such
      that:
      totalfrequency= “old total frequency”+f
    - else if key does not appear in HashMap then value is set to a
      new Integer such that:
      totalfrequency =f
```

Using the `HashMap` described above, we now go through our trigram file again this time calculating the maximum-likelihood with and without smoothing factors and place all these in a new document which is then added to our Lucene index. Since at this point we don't know which smoothing factor will give the best results we calculated maximum likelihood with the following smoothing factors: $1, 0.1, 0.01, 0.001, 0.0001$ and 0.00001.

As an example, document 54 in the index we created, is for the trigram "services have opened" and contains the following information:

- frequency = 2,
- word1 = services,
- word2 = have,
- word3 = opened,
- max-likelihood (no smoothing) = 0.006896551724137931,
- with smoothing (max-likelihood, alpha used):
  - (1.078496575773372E-5,1),
  - (7.479298370581427E-5,0.1),
  - (6.549898167006109E-4,0.01),
  - (0.003523662777900066,0.001),
  - (0.006293828423081462,0.0001),
  - (0.006831131016168353,0.00001)

4.2 The Translation Model

Writing the program for the translation model was achieved in three parts which we will discuss below.

4.2.1 Changing Text Files to Integer Files

In implementing this project, our resources were limited to a personal laptop with a 4GB RAM. We therefore had to create a way to make the most of our limited memory space. For this we switched both the French and English text files from the Europarl parallel corpus, [Koe05], to files containing integer numbers. We created files that would replace each word with an integer. This would make the `HashMaps` we later create take less space. For each file we saved a `dictionary` that can map the files back to their original state if needed.

4.2.2 Initializing Model 2 by Model 1

We discussed in the previous chapter that initializing model 2 with the translation probabilities, or the t parameter, from model 1 would, in most cases, give better results. For that reason the implementation of model 1 found in our project is limited to finding only the translation probabilities and does not calculate the end sentence probabilities of the model. In other words, model 1 is used only as an initialization tool and not as an actual working model for translation. The algorithm we followed for calculating the t parameters of model 1 is found in Figure 3.3. To keep track of the counts and the t parameter we created three `HashMaps`. Two of these keep track of $c(e, f)$ and $c(e)$ and the other keeps track of $t(f \mid e)$.

In the programming, t is initialized to 0.5 for all f and e. This technically doesn't matter as parameter values in model 1 converge to the only local maximum of the model. The program goes through a maximum of 10 full iterations to achieve convergence. The `HashMaps` for $c(e, f)$ and $t(f \mid e)$ become very large very quickly due to the need to record every pairing of words for every sentence. Unfortunately even with our transference of texts to integers, our limited resources did not allow for the full English and French Europarl files, from [Koe05], to be utilized in our model and we had to settle for sentences containing no more than 20 words. At the end that meant a total of 290,991 sentences from these files were used in creating the translation probabilities for model 1.

4.2.3 Model 2

The algorithm for calculating the parameters of model 2 is seen in Figure 3.4. Implementation of this algorithm took the creation of six `HashMaps`, three of which are the same as in model one and the other three keep track of the counts $c(j \mid i, l, m)$, $c(i, l, m)$ and $q(j \mid i, l, m)$.

The program uses the t parameters from model 1 to initialize the t parameters for model 2 and the alignment probabilities, parameter q, is arbitrarily set to 0.5 for initialization. Unlike in model 1 where the initialization value had no effect on the end result, the chosen value here may have affected the final results as model 2 does not have a single local maximum. The algorithm may have become stuck in a local maximum instead of finding the global maximum in some cases. The program again goes through 10 iterations to achieve convergence.

Again, our limited resources and also our initialization of parameter t force us to limit the programming to only sentences with no more than 20 words. The final results were calculated using our newly found t and q

parameters and Equation 3.33. We kept the results in the following way; a file to keep all translation probabilities t which is named `Model2tparameters`, a file to keep all alignment probabilities q named `Model2qparameters` and a file keeping all sentence pairs and their translation model probability as found by Equation 3.33 named `TranslationModel2`.

4.3 Decoding

4.3.1 Perplexity

We have chosen to explain the implementation of the perplexity here, because we will be using the average perplexity of our training data in our heuristics. The `class`, `PPandPMax`, in the `Decoder` package, calculates both the average perplexity and the maximum probability in the language model.

Finding the maximum probability was a simple matter of searching through the index for it. For the perplexity, we again defined a `HashMap` and put into it from the index, each unique trigram from the language model as the `key` and the maximum likelihood of that trigram with α-smoothing when $\alpha = 0.0001$ as the `value`. Experiments showed, that this α between those that we had previously calculated gave the lowest average perplexity.

We used this `HashMap` as a quick way to search for our trigrams. The algorithm, then went through our original training file one sentence at a time and calculated the perplexity for that sentence using Equation 3.10. There was some preprocessing involved here to add the beginning and end sentence markers. Then we calculated the average perplexity of the language model by taking the average of the individual sentence perplexities over all sentences in the model.

The end result of this was $p_{max} = 0.9989130381798151$ and `Average Perplexity`$= 25.484278727716106$. In the program, the values were used as is and have not been rounded.

4.3.2 Stack Decoding

The algorithm for a stack decoder was implemented using a Red-Black bianary search tree, [CLR90], instead of an actual stack as this can search more efficiently and has a complexity of $O(\log(n))$ in the worst case. We used an implementation of the Red-Black Tree by Robert Sedgewick and Kevin Wayne available online.

To implement our decoder, we need to have the probabilities from our language model, and the t and q parameters of our translation model at hand and ready for search. To this end we created three `HashMaps` containing the probabilities as `values`. The `key` for each is of the type `String`, `Pair` and `tuple4` for the language model, t parameters and the q parameters, respectively where `Pair` and `tuple4` are simple Java classes defined to keep the values we need. `Pair` contains two words in integer form for any t and `tuple4` similarly keeps the values for j, i, l, m for any q.

We define a Java class called `Hypothesis` and in this class, we keep the values of the hypothesised sentence, this is an integer list, the length l, the values g_H and h_H and the array S_H. We use this array to calculate the array in the next hypothesis and in calculating g_H.

The class `RedBlackBST` which implements the Red-Black tree has nodes with a `key` and a `value`. The `key` is what the maximum and minimum functions are based on. Since we want to find the best score, we will have f_H for each `Hypothesis` be its `key`. The `value` is then an instantiation of the class `Hypothesis`.

Our decoder implementation now divides into three main classes (not counting the Red-Black tree class). We will describe them one by one here.

In the class `CalculateScore` we use the equations in Section 3.4 to calculate the score for a given Hypothesis. This class is given a `Hypothesis`, the afore mentioned `HashMaps` and the french sentence to be translated.

It has four methods, one each for calculating g_H, h_H and f_H where f_H just adds the results of the previous two, and a static function to calculate $v_{(k+1)l}(i)$. In the method for calculating g_H, we also update our S_H.

Next, we have the class `StackDecoder`. This class, given the `HashMaps`, the french sentence and a Red-Black tree tries to create a new tree. In actual fact it is implementing the step of adding one additional word e, calculating the score and adding it to the tree.

It first checks for a complete sentence, if we have a complete sentence, then the method returns without making any changes to the tree. When the sentence is incomplete, it checks whether k has reached l. If that is the case, it adds the sentence end marker (#) to the end of the sentence and calculates scores for it and adds it to the tree.

If not, then it adds to the hypothesis, every $e \in V$, creating a new hypothesis for each. These hypothesis must also have their scores calculated and be added to the tree. For the sake of efficiency in both time and space, we sacrifice some weaker hypothesis in favour of stronger ones here.

Before adding a hypothesis to a tree we check the size of the tree. If the tree has grown bigger than a size M, then we compare any new hypothesis score with the minimum score in the tree, if the new score is larger than the minimum, the minimum is deleted and the new hypothesis added. If not, the hypothesis is ignored. The M used in this work was 20000.

The last class we implemented is called `MultiStackDecoder`. This class does the initialization for trees for each l and runs an instantiation of the `StackDecoder` class. Here, we had to decide for how long we would search in the tree and continue updating it. For lack of time, as the run time is very long, we decided to search $10l$ many times. This is likely not long enough to produce good results. In their paper, [WW97], Wang and Waible had suggested searching for as many as 6000 times for each sentence length, however here the choice was made to search longer for longer sentences.

Having a different tree for each possible length helps balance out some of the bias this algorithm has towards longer sentences. We keep the best`Hypothesis` for each length l and choose the translation from among them. When making this choice, we consider a minimum distance between l and the length of the french sentence being translated.

Unfortunately, even after many attempts at making the program more time efficient, the sheer amount of data that needed to be examined caused running time to be longer than ten hours per sentence. Due to this, and lack of time for concluding this work, testing could not be done on an extensive amount of data. We only tested the algorithm for a few sentences and never got a perfectly translated sentence. We had some partial sentences as results. It is possible that should the search have been allowed to continue, the algorithm would have given the correct translations.

Unfortunately however, at this time and using our limited resources, it is not possible to produce definite results and estimate how well the algorithm performs.

What can be said, is that the decoder was based solely on Wang and Waibels' work, [WW97], and since the algorithm was followed as precisely as possible when implementing, given sufficient resources, it should perform with results close to theirs. Their results were, that the algorithm produced correct translations with 54.2% accuracy.

Chapter 5

Conclusion

In summary, this work provided a demonstration of a SMT system that implements the translation model, IBM model 2. We explained the inner workings and idea behind a SMT system and then proceeded to explain each part separately and implement these ideas using Java.

We explained how we can model a language using n-grams, and implemented a trigram model with α-smoothing. Then, we described the translation models, IBM model 1 and 2. We implemented both these models, using the results for model 1 as initialization for model 2. In calculating the parameters for these models, we gave a brief explanation of the Expectation Maximization algorithm and used it in implementation.

We next, explained how these results can be used to build a stack decoder. We proceeded to explain how we can form hypotheses for a translation and how to give each hypothesis a score so they can be judged. The hypothesis with the best score is our final translation.

Like any system that depends on data quantity, our SMT suffers from a data sparsity problem, preventing us from getting good results. Another problem we encountered, was dependant on physical resources. The resources we had at our disposal were not large or powerful enough. Thus, to save space and time, we were forced to use even less data and sacrifice accuracy for efficiency. Even with limiting our data, the final decoder had a very long run time for a single sentence and lack of time prevented us from producing substantial results in the final decoding stage of this work.

[]

Bibliography

[BPPM93] Peter F. Brown, Vincent J. Della Pietra, Stephen A. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIt Press, 1990.

[Col11] Michael Collins. Statistical machine translation: Ibm models 1 and 2, 2011. Lecture notes, Columbia University.

[Kni99] Kevin Knight. Decoding complexity in word-replacement translation models. *Computational Linguistics*, 25(4):607–615, 1999.

[Koe05] Philipp Koehn. Europarl: A parallel corpus for statistical machine translation. In *MT summit*, volume 5, 2005.

[Koe10] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, 2010.

[luc11] Apache lucene core. http://lucene.apache.org/core/, 2011. Version 3.5.

[Nil71] Nils J. Nilsson. *Problem-solving methods in artificial intelligence*. McGraw-Hill computer science series. McGraw-Hill, 1971.

[Nor98] James R. Norris. *Markov Chains*. Number Nr. 2008 in Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1998.

[Pre04] Detlef Prescher. A tutorial on the expectation-maximization algorithm including maximum-likelihood estimation and em training of probabilistic context-free grammars. *arXiv preprint cs/0412015*, 2004.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[WW97] Ye-Yi Wang and Alex Waibel. Decoding algorithm in statistical machine translation. In *Proceedings of the eighth conference on European chapter of the Association for Computational Linguistics*, pages 366–372. Association for Computational Linguistics, 1997.