Charlie Seymour

Machine Translation System Combination for Resource Poor Language Pairs

Computer Science Tripos – Part II

Corpus Christi College

May 19, 2017

Proforma

Name:	**Charlie Seymour**
College:	Corpus Christi College
Project Title:	**Machine Translation System Combination for Resource Poor Language Pairs**
Examination:	Computer Science Tripos – Part II, June 2017
Word Count:	**11,000**
Project Originator:	Zheng Yuan
Supervisor:	Zheng Yuan, (Dr E. Kochmar)

Original Aims of the Project

To investigate the effectiveness of combining statistical machine translation systems via pivoting. To evaluate the results, learning the contributions made by choice of language pair and corpus size. To further investigate the effectiveness when combining via an alternative system output combination technique, namely confusion network decoding.

Work Completed

With target language English: baseline systems for later comparison trained; system for every source-pivot language pair possible for the corpus' remaining 5 languages trained; results of the above evaluated. In addition (unplanned) experiment management system implemented, allowing rapid specification and execution of system training. Confusion network hypothesis combination algorithm implemented and tested, system union implemented, weighting models incomplete.

Special Difficulties

Misinformation in external documentation resulted in entire experiment management system stage being required. Further system/tool issues discussed in Conclusion.

Declaration

I, Charlie Seymour of Corpus Christi College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: *Charlie Seymour*

Date: May 19, 2017

Contents

List of Figures

Acknowledgements

A huge thanks to my supervisor, Zheng Yuan, who's support throughout the project was invaluable. My project would also not have been able to be completed to anywhere near this standard were it not for the generous grant given to me by Microsoft for the use of their Azure cloud computing platform—and for the kindness and assistance of Geoff Hughes and Lee Stott of Microsoft Academic Relations.

I also thank Hieu Huang for his inclusion of my amendment to the Moses manual.

Chapter 1

Introduction

1.1 Project aims

The essence of the problem my projects attempts to tackle is that Statistical Machine Translation systems (i.e. translating human language) are less effective if the pair of languages don't share the appropriate resources. I try to combat these issues by combining more than one of these systems. Firstly, combining them during translation, by redirecting translation through a new "pivot" language ("pivoting"), then secondly, after translation, by combining different versions of the *same text* outputted by *multiple different* systems—using a confusion network to try and extract a better version of each sentence.

To achieve this, I gave myself the following aims:

- Train "baseline" translation systems
- Train a range of pivot systems
- (Using these,) improve on the scores of the baseline systems
- Either:
  - Continue tuning your systems to further increase scores, or
  - Research and develop an alternative system combination technique

1.2 Introduction to technical background

Whilst early researchers in Machine Translation (MT) predicted the most successful approach would be *rule based*, with source language being transformed to target language by means of some formal specification of each of the languages' syntax and semantics, the intricacies of natural languages have proved too difficult to master. This complexity (and other difficulties), combined with increases in computational power since the birth of the field, have resulted in the *statistical* approach (SMT) seeing the greatest successes.

This approach is dependent on bilingual text (known as 'parallel corpora') in the two languages to be translated. The variant I am concerned with, namely 'phrase-based' SMT, goes on to train a system to learn correspondences between (continuous) *sequences*

of words, which it later uses to translate unseen text. Whilst this dodges the need for a detailed understanding of the two languages, it certainly has problems of its own, the foremost being its dependence on these corpora. Both the quality of the translation and the size of the parallel corpus are critical to the system's success, with uncommon language pairs having restricted corpus availability, thus producing worse systems. Other problems include lower translation quality of general texts when systems are trained on domain specific corpora, and the time and resource requirements of training, tuning, and translating with SMT systems.

The core topic for this project, whilst requiring the training of many SMT systems to enable it, is the *combination* of such systems. System combination encompasses a variety of techniques designed to overcome some of the aforementioned issues with the SMT methodology. Take the issue of resource poor language pairs. If you consider the language pair Arabic–Chinese, it should be clear that the number of available texts translated between the two of them will be far lower than for either language and English. If this disparity causes a significant enough difference in the quality of the 'direct', Source–Target translation system (S-T), then it can be beneficial to introduce a third, 'pivot' language. This will sit between the source and target languages, with its input and output being directed accordingly. In our example, English might be a good choice of pivot language, as there is a large amount of parallel texts between it and either language. However, it isn't a perfect selection, as it isn't syntactically similar to either language. Both language similarity and corpus size are important metrics to predict the quality of an SMT system, and likewise for picking pivot languages.

The intuitive method to combine two translation systems into a single pivot system is 'sentence pivoting'. Each sentence is translated from Source–Pivot (S-P), and then each of these sentences is translated from Pivot–Target (P-T). This is the least complex of the various pivoting techniques, and can suffer from error compounding—errors in the S-P stage can lead to greater errors in the P-T stage. A much more involved technique is 'phrase pivoting'. In phrase based MT, corresponding phrases are stored in a 'phrase table', containing other information such as the likelihood of appearance. Phrase pivoting combines the S-P and P-T tables' phrases (where they match) and weights (using fairly detailed statistics), giving a final S-T phrase table. Another technique is a variant on sentence pivoting, where each of the 'n-best' S-P sentences are put though the P-T, and the best result chosen. Pivots are a fairly recent development in SMT, and their usage in earlier work is mentioned in further detail throughout my proposal.

[1] cite the lack of sufficient parallel data to be a major bottleneck for SMT between certain language pairs, such as Arabic and Italian, finding success when using English as a pivot language between the two. Another advantage of using a pivot when dealing with many languages is the reduction from order quadratic to linear in the number of systems needed; [2] using the JRC-Acquis, a parallel corpus of EU Law in 22 languages with over a billion words, reduced their required language pair systems from an initial 462 to 42 by simple chaining using a pivot language. They found a general increase in performance when using English as a pivot, and small but consistent gains when using a

novel multi-pivot system.

This isn't to say that using a bridge language is without problems. Naturally, when using imperfect systems, such as SMT systems in their current state, multiple applications can lead to errors being compounded, thereby lowering overall translation quality. [3] found a slight reduction in performance when using two pivoting techniques with Spanish, French and German. [4] didn't find an improvement when using a bridge language over direct MT, although they improved the direct system using phrases learnt via pivoting.

Lastly, an even more recent development in system combination is doing so via Confusion Networks (CNs). My use involves the combination of the final output of multiple S-P-T systems, and as little more than an outline was known prior to my development of a suitable algorithm, they shall be discussed in further detail in the implementation chapter.

Chapter 2

Preparation

2.1 Requirement Analysis

In terms of the resource requirements I went into the project believing necessary, no changes were made to the originals in the project proposal. These were two of my own laptops, the more powerful of the two being a Mid-2014 Macbook Pro (2.8GHz, 8GM RAM). My estimates for the amount of time it would take to train individual systems were based upon timings given in the manual for the baseline system, which referred to a laptop of very similar specifications training a system on a 150,000 sentence corpus, quoting the time taken for the entire pipeline to complete as approximately 6 hours. Originally planning to consider and test corpus sizes between 100K and 1M sentences, I predicted the corresponding timings to sit between roughly 3 and 30 hours, as the examples in the manual were only using 2 out of an available 4 cores. Naturally, these estimations, as well as my supervisor's assurance of their accuracy, were based on the (apparently unreasonable) assumption that the timings given in the manual would reflect my almost identical situation.

Ultimately it turned out that even when using all the cores available on my machine (translation is an "embarrassingly parallelisable" task), the time it was taking was almost twice the quoted figures. This was a serious blow, as it would mean I would have to train smaller systems if I was going to be able to finish them quickly enough—this may have meant it would have been harder to demonstrate the benefits of system combination I was trying to show, as the component systems would all be of lower quality, causing greater error compounding. Fortunately, it would turn out, whilst I was looking at server hiring to meet my needs, I was able to get a generous grant from Microsoft's Academic Relations (around January 2017), enabling me to use their cloud computing platform 'Azure' for the remainder of my project.

Finally, the tools I would be using were all those mentioned in the proposal, with Moses being the predominant, plus OpenFST and METEOR, which were later added when it was decided that I would be pursuing the confusion network branch of the later stages of my project. These tools, as well as the timing of their selection, are discussed below.

2.2 Moses

Moses[8] is an open-source SMT toolkit that is widely used in academia, as well as increasingly in industry. It's successful coverage of wide areas of current SMT technology as well as its long development time (it started as successor to "Pharaoh" in 2005) result in an effective, albeit large and at times complicated tool. This encompassing nature means there is a high level of choice and customisability (it has a 350 page manual), a large number of tools (the "scripts" folder contains just over 200 scripts), with each having a large number of options (the decoder itself having just over 150, as well as over 70 possible feature functions).

The decoder just mentioned is one of the two "main components" of Moses, the other being the training pipeline, which is less of a single component, consisting of a collection of tools which perform various operations on a corpus to transform it into a translation model for use with the decoder. Starting with the plain, bilingual text (which we assume has already been aligned at the sentence level), the first step in the training pipeline is to 'tokenise' the text, separating words from punctuation (e.g. `It's HIS dog!` ⇒ `It ' s HIS dog !`), and the second is to 'truecase' (⇒ `it ' s his dog !`, such a step unnecessary for e.g. Chinese). After removing excessively long or short sentences, the sentences are aligned at the word level (using statistical models). These word-alignments will then be used during system training.

In order to evaluate the decoder's output, Moses must also train a 'Language Model' (LM—an external tool, in our case 'KenLM'[1]), this is a tool that measures fluency of output after being trained on *monolingual* text—i.e. text in our target language. With the ability to evaluate its own output (enabled by the LM), Moses can now "tune", a lengthy and computationally expensive process in which a "development set" (devset) is repeatedly translated with continually adjusting scores in order to find the best weights for the system.

A series of major installation complications resulted in Moses not being able to include its Experiment Management System (EMS), an absolutely critical component for situations like mine (lots of similar systems to be trained and tested), that provides a unified interface to the numerous scripts that have to be called on each run-through of the training pipeline. As a result, if I were to be able to train the large number of systems I would need to accurately test my results, I would need to implement my own entire EMS from scratch—a mammoth task to be given with so little notice.

As for the corpora I would be using, I chose the United Nations Parallel Corpus (UNPC). This is a large (11 million sentences), high quality (official, professional United Nations translators) corpus, in the six official languages of the UN (Arabic, English, Spanish, French, Russian, Chinese), consisting of official documents from 1990–2014, already aligned to the sentence level. It is a strong resource well suited to the task, but recently released (May 2016), so at the initiation of my project, mine was one of the earliest works to use it.

2.3 OpenFST

Whilst the planning involved in the confusion network part of the project took place after my EMS was operational—under the heading of the possible extensions specified in the project proposal—I place the preliminary description of the tools used here for consistency's sake. The library I utilised for the handling of the Finite State Transducers (FSTs) I would be producing, OpenFST, is the standard such tool, and has seen various usages in machine translation, NLP and machine learning. In this regard, there was little choice, and the tool fulfilled my needs perfectly. These FSTs would be what represented the confusion networks I would be building. Whilst an FST typically transforms an input string to a different output string (it thus defines a relation between strings), my interest is only in the fact the library can produce *weighted* automata; as such the networks I build don't affect the strings themselves.

OpenFST[4] is an open source library that is well written and efficient, providing all the tools I needed (or could reasonably want) for the creation and transformation of FSTs, for example, taking the 'Union' of two, with the result being consistent with what's expected when both are considered as the relations they each define. Whilst there were moments where it forced a particular way of thinking, e.g. the 'n shortest' paths functioning returning another FST from which the paths themselves had to be manually extracted, for the most part the library was consistent and well documented. OpenFST necessitated C++ as the language used, which was (coincidentally) in line with my original plans. Despite its benefits, it was still another tool that had to be learned to use, which took a non-negligible amount of time.

The last tool I used was 'METEOR', a word-alignment performing a very similar function to the word-aligning mentioned in the technical introduction. As with the rest of the confusion network stage, it will be discussed in the implementation chapter.

2.4 Proposal refinement

The early stages of my proposed work items, that is to say the section regarding sentence pivoting, as well as the plan for the "fork point", remained the same going into the implementation stage. This fork was a planned point in my project, after the completion of an initial working pivot system was made, I had planned to choose between concentrating on the refinement of the current system, or on implementing an alternative combination method—the later was chosen, and thus the decision to move into CNs was made at that time.

The most significant addition, and the most unplanned, was the Experiment Management System that I now had to implement. Other than that, out of the two possible

extensions I offered in the proposal, namely phrase pivoting and multi-pivot, the former was not attempted, whilst the latter describes a similar approach to the one I took with the confusion networks, albeit it on a far smaller scale than that of the paper cited there.

2.5 Plan

In stating here my plan, I once again reiterate that the confusion network sections were planned after the EMS was operational, but this remains a "pre-coding" plan (as the CNs were fully planned before implementation) despite the lack of consistent timing.

2.5.1 General structure

My primary concern when considering the general structure I wished my program to take was *linearity*. In order to aid the design process, to facilitate testing, and to keep the project in line with proper software engineering principles, every stage was to be completely separated from every other, and each section of each stage was to be separated as much as possible.

Similarities can be drawn to Moses' pipeline, in which each stage passes its output directly on to the next, without any cycles in its data flow. This, paired with sensible sectioning of the pipeline's components, provides a high degree of modularity. Whilst the primary benefit from a user's perspective with Moses' modularity is the interchangeability of components it provides, from a software engineering perspective, or more specifically, from my perspective as the developer, modularity would hugely simplifies the testing process. It would allow each section to be tested completely independently, and once results had been verified, development (and subsequent testing) could move on to the next section.

Another benefit of the linear structure planned, and another key design principle in general, is abstraction. Clearly, for each stage to be successfully modularised, there has to be a good degree of abstraction between consecutive stages in order for each to operate independently without requiring knowledge of the internals of other stages. On the note of abstraction, the EMS that I had to now build, and Moses' EMS it replaced, exist for that very purpose. With the intricacy of all the tools in the Moses pipeline, the depth of options, and the number to choose for each stage, such abstraction was absolutely critical for me, as I planned to train many systems of slightly differing parameters. I also needed the ability to tweak general training parameters as I empirically discovered the best settings for my needs.

Although trying to ensure modularity and abstraction, alongside the core linearity I had deemed necessary, would increase planning time significantly, the application of such design principles tends to pay for itself, as the ease of use it should provide will greatly increase efficiency and reduce errors in general, also leading to reductions in

testing times. It was also necessary to give me freedom of choice at the later stages of my project, as (were I to choose the "refinement" path) improvements could be made on each module independently, or (with the "further technique" path I took) allow me to just continue the data flow's linear progression by simply adding additional systems to the "end" of what I had completed so far. These principles were also applied internally to the extension I chose, to maintain the same benefits discussed.

2.5.2 Continuous validation

As one of the key reasons for planning to maintain linearity throughout the project's implementation was for testing, I opted not to take the "traditional" route of testing, whereby the system is extensively checked for bugs after it has been implemented, instead borrowing ideas from "agile" methodologies. Each module was to be tested thoroughly upon completion, with the entire project "so far" being tested each time a significant portion was completed. As such, each module was arguably to be tested in the traditional manner—if they were to be considered small projects in their own right.

2.5.3 Final objectives

My choice when it came to deciding which aspect of the pivot systems I would be training to vary the most was mainly between corpus size and pivot language. As I wouldn't have time to train a large number of systems if their size was approaching the size of the corpus, I chose to maximise the pivot variety, as I thought this effect would be the most unpredictable of the two. I chose to keep my target language as English, as it would make it easier to evaluate system output and do further operations on the final output (as the CNs later did).

With six available languages and with English selected as target language for all the pivot systems, there are a possible five P–T systems (which can be reused), each of which can be paired with a further four S–P systems. On top of these, I will need five "direct" systems (S–T), which will use smaller corpus sizes and will serve as a baseline for measuring the effectiveness of pivoting. This gives a total of $5 + (4 \times 5) + 5 = 30$ systems possible for a given baseline/pivot pair of corpus sizes.

I chose to implement all 30 of these systems, with corpus sizes of 55,000 and 550,000 sentences (pre-filtering) for the baseline and pivot systems respectively, with the corpus sizes decided partially empirically to have the system training/tuning time at a reasonable level.

2.5.4 Milestone plan

The setbacks I had suffered so far, especially the introduction of the EMS implementation stage of the project, meant that any arrangement of timings was going to be tight if my project was going to achieve any of its original objectives. My revised timings were as follows:

Remainder of Michaelmas:	First quarter of EMS
Christmas Vacation:	Remainder of EMS
Decision point:	Choose between refinement and further techniques
First quarter of Lent:	CN research, progress report
Second quarter of Lent:	METEOR implementation
Second half of Lent:	Confusion Network implementation
Easter Vacation:	Evaluation and Dissertation

2.6 Starting point

The libraries I will be using have been discussed in detail already, but to summarise

- **Moses**, but without the use of its Experiment Management System
- **External tools for Moses**, such as MGIZA (word-alignment) and KenLM (language model)
- **OpenFST**, for the creation and manipulation of Confusion Networks
- **METEOR**, another word-aligner, for data preprocessing for the Confusion Networks

ar	Arabic
en	English
es	Spanish
fr	French
ru	Russian
zh	Chinese

(a) Official language codes

Source	Pivot	Target
es fr ru zh	ar	en
ar fr ru zh	es	
ar es ru zh	fr	
ar es fr zh	ru	
ar es fr ru	zh	

(b) Possible pivot systems

Table 2.1: Pivot system summary

Chapter 3

Implementation

Implementation went mostly to plan, with all parts of the plan completed except the very final stages of the confusion network stage. The setbacks faced during implementation did cause significant impacts to timing, but none caused any additional parts of the plan to be incomplete.

In accordance with my plan for system testing already given, I shall describe the testing done at the appropriate points throughout the Implementation chapter, so please note its partial absence from the Evaluation chapter.

3.1 Experiment management system

Moses dedicates around 6% of its entire manual to the provided EMS, as it is a nearly 4000 line perl script with a large number of features, the main one being the automation of all the main stages of the system training pipeline. Whilst it was only this core functionality I intended to implement, this shows the potential size of the task.

3.1.1 Development

The main script in my EMS was "`new-system.sh`". Naturally, it follows the direction of Moses' training pipeline closely—this has been briefly outlined, but I will discuss it here in further detail. The steps, in terms of other scripts of my EMS called, of the `new-system.sh` script are as follows:

- `prepare-corpus.sh`
- `clean.sh`
- `train-lm.sh`
- `train-moses.sh`
- `tune-moses.sh`
- `binarise.sh`

- `filter-model.sh`
- `translate-for-bleu.sh`

The `prepare-corpus.sh` scripts was one of the largest, as it had to handle multiple cases of language pairs that resulted in calls to different tokenising tools. For European languages (e.g. en, es, fr, ru), Moses' included tokeniser is called, followed by its truecaser. Throughout my EMS, files are named in a consistent manner (e.g. p0.ar-fr.true.ar for the truecased source of a (pivot, not baseline) ar-fr system) to aid the exchanges between scripts. For zh and ar, two segmenters provided externally by Stanford[3] are used, and no truecasing is required for those languages.

A Moses script must be run on the original source corpus for zh and ar to remove particular punctuation otherwise handled by the tokeniser script. All languages go through the same cleaning script, removing any sentences whose lengths violates given bounds. The `prepare-corpus.sh` script is the only place that is affected by the language of the input, in line with proper principles of abstraction.

Then a language model is trained (and binarised) on the target corpus if necessary, as language models are reusable. Afterwards the actual training takes place, with a core count passed on to the Moses script to allow parallelisation. Similarly, tuning then takes place with a given thread count. To speed up translation, the translation model is binarised, which requires some careful (automated) editing of the ini file (using the stream editor `sed`).

Moving on to the evaluation stage of the pipeline, the translation model is filtered for only the phrases it needs for the testset. This speeds up translation (which is effectively a big search/optimisation problem), and the output is fed into the BLEU script for scoring. This becomes more involved for pivot systems, but the `new-system.sh` script handles that case when needed.

My main original contribution to the EMS was my "intelligent control file creation", whereby an EMS control file (specifying file locations and script parameters) is created using the minimum information possible, filling in likely defaults and working out extras when needed. This drastically reduced the amount of time needed to describe each system to the EMS, and avoided errors such as file name typos, as it would infer their location from e.g. the provided languages and the default folders. Naturally, these can all be changed per case if needed (e.g. for a baseline system).

3.1.2 Testing

The gradually incrementing nature of the testing pipeline meant that a good deal of effort was required for testing between each of its stages. Whilst this modularity helped in separating the required testing into segments that could be completed in isolation, it also meant that the system had to be repeatedly run from the beginning each time a component had been tested, to ensure the master script was handling the interactions between them correctly.

The EMS did have the benefit that every part of it utilised Moses' tools, as such, malformed inputs or misuse should provide "built-in" error messages, although this ended

up being far from always the case. A quick and useful technique for checking the pipeline as a whole was inspecting the languages present in the output—naturally the target language will be clearly visible, but as the translations are never perfect, there will be source language words present in the final output. This holds for pivot languages too, providing a means of verification against major errors, such as incorrect files (i.e. wrong language) being passed through.

Many errors were less helpful, with effectively silent errors causing empty files to be written before continuing with execution, only to fail later on when the files were needed. One in particular was caused by the training system to run out of disk space—despite there appearing to be plenty (given the size of the corpora and models), temporary files and later binarising meant that the trainer silently exceeded and failed, writing and empty phrase table that isn't recognised until the tuning stage, which then fails cryptically.

3.2 Confusion Networks

As described in the Introduction to Technical Background, a confusion network is a weighted automaton that accepts a particular set of strings. In our case, our 'alphabet' will be words from a set of "input hypotheses" (supplemented by the empty word, epsilon); paths through the CN will be particular combinations of these hypotheses; and the weight of each path will be a measure of its expected quality—thus the weight of each individual arc between states should measure the suitability of choosing the word on that arc, given the path so far (that led us to our current state).

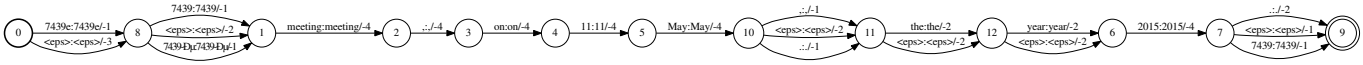


Figure 3.1: Example CN

I took the concepts of MT system combination using CNs from a paper outlining a novel MT system combination framework as part of the 'Jane' toolkit[6], although with some noted differences. Their approach was very broad, and experimental—this breadth meant that implementing every component they attempted would be infeasible, but this experimental nature meant that I wouldn't need to, as I could evaluate their results and use only the parts that contributed the greatest differences.

Aside from the core hypothesis combination step where the main CN is created, the authors also include a word reordering technique, as well as weighting based on:

- The utility of each hypothesis' source system
- Whether or a hypothesis is the primary
- The results of a (3-gram) language model, trained on the hypotheses themselves
- A word penalty, to stop results differing greatly from the average hypothesis length
- Additional model: "big-LM" trained on a larger target-language corpus

- Additional model: “IBM-1” lexical translation model

Naturally, the core CN generating part of the method was essential and included. I also selected the ‘word-penalty’ feature, as I foresaw the likely gaps in the alignments between hypotheses potentially causing under-selection caused by the epsilon arcs being excessively weighted. The language model feature I chose too, as the other main problem I imagined was generally poor results selection when using the simple path extraction function that had no knowledge of what made a sensible sentence. I decided the LM would sufficiently encompass the effects of the word reordering, as badly ordered sentences suffer under an n-gram model; similarly I decided the union operation taken at the end of the algorithm should account for any differing utility from each of the hypotheses’ original systems. The details of the chosen components are to be discussed. Despite the outline provided in the paper, implementation would require my own algorithm, as their algorithm was only outlined—all those parts I based work on are mentioned below.

The input hypotheses are the “single best” outputs from multiple translation systems, without any additional information, that all correspond to (different versions of) the same target language sentence. Of these hypotheses, a ‘primary hypothesis’ is selected, and the remaining ‘secondary hypotheses’ are word-aligned to it. Using a (fairly detailed) algorithm of my own creation, these alignments are used to build a confusion network around the initial primary hypothesis, introducing alternative word(s) based on the word alignments generated. Weights are introduced during this process, and the from resultant CN (fig. 3.2) alternative paths can be extracted. For n input hypotheses, we create the n networks possible with each hypothesis as the primary, taking the union of these n for our final network. The shortest paths are then extracted, a length penalty is applied, and the best are put through a large language model (big-LM) to select the final output. This process is repeated for every sentence in the original text.

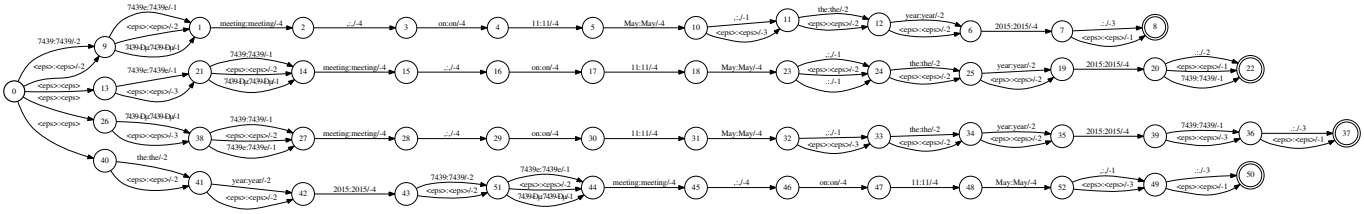


Figure 3.2: Example union of 4 CNs

3.2.1 METEOR

Before being able to construct a CN, hypotheses must first be aligned to each other using an appropriate tool. METEOR has many features to its name, including the ability to align synonyms, words with identical stems, and monogram paraphrases; not just exact matches. However the main reason I use it here is due to its high precision.

Precision and recall are two related concepts in information retrieval that measure the effectiveness with which a system extracts data. Recall refers to the fraction of all the

possible "true-positives" (i.e. correctly extracted data) that the system retrieved; it is calculated as the number of true positives extracted divided by the number of positives in the data set. Precision refers to the fraction of your retrieved data that was correct; calculated by dividing the number of true positives extracted, by the number of extracted items overall.

To understand the importance of precision in this usage, consider where the alignments will actually be used. Each alignment will allow words from another hypothesis to be interchanged—in order not to introduce errors into our CN output, it is more important that we have fewer erroneous alignments, even at the cost of missing some altogether. I will go on to show that these misses can be counteracted by the sensible use of a length penalty function.

METEOR is used also in [6], although they make modifications to the METEOR database in an attempt to improve recall—I instead chose just to focus on precision for the reasons outlined above.

```
Alignment 1
7439 meeting , on 11 May 2015.
7439e meeting , on 11 May by the year 2015
Line2Start:Length       Line1Start:Length       Module          Score
1:1                     1:1                     0               1.0
2:1                     2:1                     0               1.0
3:1                     3:1                     0               1.0
4:1                     4:1                     0               1.0
5:1                     5:1                     0               1.0

Alignment 2
7439 meeting , on 11 May by 2015 .
7439e meeting , on 11 May by the year 2015
Line2Start:Length       Line1Start:Length       Module          Score
1:1                     1:1                     0               1.0
2:1                     2:1                     0               1.0
3:1                     3:1                     0               1.0
4:1                     4:1                     0               1.0
5:1                     5:1                     0               1.0
6:1                     6:1                     0               1.0
9:1                     7:1                     0               1.0

Alignment 3
item 7439 meeting , on 11 May in 2015 .
7439e meeting , on 11 May by the year 2015
Line2Start:Length       Line1Start:Length       Module          Score
1:1                     2:1                     0               1.0
2:1                     3:1                     0               1.0
3:1                     4:1                     0               1.0
4:1                     5:1                     0               1.0
5:1                     6:1                     0               1.0
9:1                     8:1                     0               1.0
```

Figure 3.3: Example alignment output from METEOR

```
void secHypos(std::string *fnames, int fc, std::string hypfn) {
/*  Write secondary hypothesis files, each containing
    ONE of FC-1 files' (e.g.) Kth lines, excluding the
    line of the primary hypothesis they are tied to.    */

  int fl = fileLength(fnames[0]); //they'll all be the same

  int hypi;
  std::string **flinesArr = new std::string*[fc]; //\\DYNA\\//
  for(int fi=0; fi<fc; fi++)
    if(fnames[fi] != hypfn) {
      flinesArr[fi] = new std::string[fl]; //\\DYNA\\//
      getFile(fnames[fi], flinesArr[fi]);
    } else
      hypi=fi;

  for(int k=0; k<fl; k++) { //for each line in (all) file(s)
    std::string outname="scnd."+std::to_string(k)+"."+hypfn;
    std::ofstream fs(outname);
    if(fs.is_open()) {
      for(int fi=0; fi<fc; fi++)
        if(fi != hypi) fs << flinesArr[fi][k] << std::endl;
    } else
      std::cerr << "ERROR OPENING FILE #2E" << std::endl;
  }

  for(int fi=0; fi<fc; fi++)
    if(fi!=hypi)
      delete[] flinesArr[fi];
  delete[] flinesArr;
}
```

Figure 3.4: C++ for creating secondary hypothesis files from a given file list

Hypotheses file generation

Before METEOR can be run, the input data must first be generated. If given n files, each containing k sentences, consider the task of generating primary hypotheses. Taking the first sentence for example's sake (recalling these files must be *sentence* aligned), we must select a file to source the primary hypothesis. Once we have that sentence and that file, we know that we will need to align it with the other $n-1$ files' first sentences, so we create a file with that sentence in it, $n-1$ times. We repeat this for every k lines in the file we chose, so this must all be done n times, giving nk files with a total of $n(n-1)k$ lines.

Similarly for secondary hypotheses, once a primary hypothesis sentence has been chosen, we make a file containing the other $n-1$ versions of that hypothesis. Again, we must do this for all k lines of the current primary file, and then for each of the n files, this time omitting the corresponding primary. This gives us the same file and line count as above, as expected given that METEOR aligns every sentence in a pair of files, and every file is expected to be used.

As we can see in Figure 3.4, the algorithm described is precisely followed in the implementation.

1. It's a brown dog
2. That is not true
3. I think

(a) File 1

1. C'est a brown dog
2. Ce is not true
3. Je think

(b) File 2

1. It's a brown chien
2. That is not vrai
3. I pense

(c) File 3

Table 3.1: Example target language files

File number	Line number	Primary hyp. file	Secondary hyp. file
1	1.	It's a brown dog	C'est a brown dog
		It's a brown dog	It's a brown chien
	2.	That is not true	Ce is not true
		That is not true	That is not vrai
	3.	I think	Je think
		I think	I pense
2	1.	C'est a brown dog	It's a brown dog
		C'est a brown dog	It's a brown chien
	2.	Ce is not true	That is not true
		Ce is not true	That is not vrai
	3.	Je think	I think
		Je think	I pense
3	1.	It's a brown chien	It's a brown dog
		It's a brown chien	C'est a brown dog
	2.	That is not vrai	That is not true
		That is not vrai	Ce is not true
	3.	I pense	I think
		I pense	Je think

Table 3.2: Example generated hypothesis files

```
COUNT=0;
NOW=$(date "+%H:%M")
for LNG in ar es fr ru zh; do
  if [ $LNG != $L ]; then
    for I in $( seq $MIN $PARAC $MAX ); do
      if [ $MEMMAX -gt 0 ]; then
        for J in $( seq $I $(( $I + $PARAC-1 )) ); do
          if [ $J -gt $MAX ]; then break; fi
          echo -e -n "$J $LNG\t"
          java -Xmx${MEMMAX}M -cp $METEOR Matcher \
            $SCNDDIR/scnd.${J}.${PRF}.${LNG}-${L}.${SUF} \
            $PRIMDIR/prim.${J}.${PRF}.${LNG}-${L}.${SUF} \
          > $ALNSDIR/alns.${J}.${PRF}.${LNG}-${L}.${SUF} &
          PIDS[$J]=$!
          COUNT=$(( $COUNT + 1 ))
        done; echo;

        if [ $(printf "%.0f" $P) -gt $IP ]; then
          IP=$(printf "%.0f" $P)
        fi

        progressbar

        for K in $( seq $I $(( $I + $PARAC-1 )) ); do
          wait ${PIDS[$K]}
        done

        P=$( bc -l <<< "100*$COUNT/$TOTAL")

      else
        echo "Something has gone seriously wrong! MEMMAX: $MEMMAX"
        exit 4
      fi
      OLDI=$I
    done
  fi
done
```

Figure 3.5: Main part of `genmeteor.sh` where parallel execution can be seen

Calling METEOR

With the hypothesis files created, METEOR must now be called on each appropriate pair of primary hypothesis and secondary hypotheses files. As there is a very large number ($k = 4000$, $n = 4$ across 5 pivot languages, totalling $4000 \times 4 \times 20 = 320,000$), a parallel script was necessitated to expedite the process. Even with this speed boost, the script still took over 60 hours to complete all the files, on a powerful VM (8 cores, 16GB RAM). This was certainly longer than I anticipated, but was not much more than a small setback, particularly in light of some of the others experienced.

Processing METEOR output

Processing the files efficiently, mainly because of the depth of consideration needed with regard to the design of the data structures in use, was a larger task than apparent. These needed to be allocated and deleted with care due to the number of files I would be processing. Not only this, but they also needed to be forward looking in their design as they would be passed directly on to the CN stage. Taking the time to implement this

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

#ifndef PROCESS_METEOR_HEADER_GUARD
#define PROCESS_METEOR_HEADER_GUARD

typedef struct fileMetaData {
  int length;
  int alnCount;
  int *blankLines; //\\ DYNAMICALLY ALLOCATED \\//
} fmd;

typedef struct wordAlignment {
  int sen2start, sen2len; //ref sentence
  int sen1start, sen1len; //test sentence
} waln;

typedef struct alignmentData {
  std::string *sen1; //(in words) //\\ DYNAMICALLY ALLOCATED \\//
  std::string *sen2; //(in words) //\\ DYNAMICALLY ALLOCATED \\//
  int sen1l, sen2l;
  waln *alns; //every word alnmnt //\\ DYNAMICALLY ALLOCATED \\//
  int alnc;
} alnData;
void mismatch(std::string found, std::string expected);
void getFileMetaData(std::string filename, fmd *pfmd);
void readMeteorFile(std::string filename, fmd *pfmd, std::string *plines);
int wordCount(std::string line);
void wordSplit(std::string line, std::string *words, int len);
void processMeteor(std::string *lines, fmd *pfmd, alnData **dataArr);
void displayAlnData(alnData *data);
void displayAlnData(alnData **dataArr, int len);

#endif
```

Figure 3.6: Header file showing data structures and function prototypes used to process METEOR output

well gave a resultant solution that caused no hindrance to later stages and was fast—thus avoiding a potential bottleneck in my system combination pipeline.

Testing

As seen throughout the project, testing was made easier by an abundance of test data. For the hypothesis generation, excerpts of data were manually selected and inspected. As inspecting the full output was infeasible (thousands of files), these excerpts were run through by hand, and then checked with the output of program to ensure quality control.

As the proper functionality of METEOR itself need not be tested, only the parallelisation internal to the script required checking. Some subtleties in the calculation of how much memory to permit each Java process, namely that the number of languages added a factor of 4 to the total number of calls, gave some unclear (and apparently uncommon) Java garbage collection errors, but the bug was found and fixed.

The processing of the data required more extensive testing of internal components, for which mock alignments were initially penned to keep the size reasonable while checking

results during development. Later, full alignments were used (even if only in part, for testing a particular component), with data abundance again providing assistance. As (the essential parts of) the data were all accessible after processing, this provided a very useful testing tool, as the results could be outputted side by side with the input, and a direct comparison drawn.

3.2.2 Algorithm Development

Figure 3.7: Example CN from [6] with shortest path highlighted

In developing my algorithm, I followed an iterative procedure of improvement, by repeatedly trying a run-through of the algorithm developed so far, then tweaking the algorithm to dodge caveats or when stuck, or to add beneficial improvements. In describing the design process, I will illustrate the steps I took, highlighting each point of alteration.

My starting point, given in [6], was simply that the CN is initialised with the primary hypothesis.

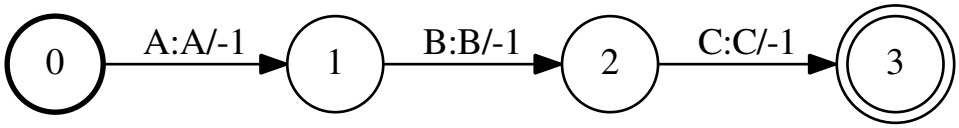


Figure 3.8: Initial CN (PH only)

Typically we would now set state 0 to be the unique "start state", however (as states may be added before it) in the final algorithm we work out the start state at the end. Here you also see that the last state has a double ring, marking it as a "final state", but once again these states will be worked out at the end in the final algorithm (finals are not unique). Note also the -1s that appear on the arcs, these are the initial weights given to every word. The words appearing twice on the arcs is an artefact of the OpenFST tool used to draw them, which expects the input and output label of each arc to (potentially) differ. Our weights are negative simply because in our use case we want a "more weighted" path to be a better path, and so the greatest negative length path will be returned by the shortest path algorithm OpenFST already has implemented.

From this point, I planned to align the secondary hypotheses one by one in no particular order. Of the weighting models used in [6], I started with the core 'majority voting' method. In this technique, each arc is given a weight of 1 when added. We thus want an algorithm which adds these weights together when two arcs "vote" for the same

thing (at the same position), similarly, one that takes count of the fact that when we add an arc, all the other paths so far (and most likely some afterwards) "disagree" with that arc.

Consider a secondary hypothesis that shares an alignment with the first word of the primary. Keeping our PH as $(A\ B\ C)$, and assuming this it the first hypothesis we are aligning, our SH will either begin with A or will have some words followed by A. Take these two possibilities as $(A...)$ and $(d\ e\ A...)$.[1] In the first case, we can just increase the weight[2] of the already present arc labelled A and continue the algorithm with the second word in the SH. In the second case, as there is nothing before the current state (we are on the first SH) we can just add each word as new states and arcs before the current state, but we must remember to add epsilon arcs to each of these. This is how we represent (in this case only the PH) *not* "voting" for the arcs we have just added.

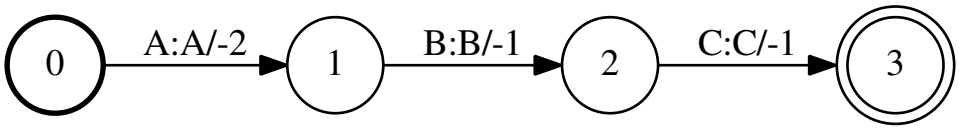


Figure 3.9: Initial alignment: Case 1

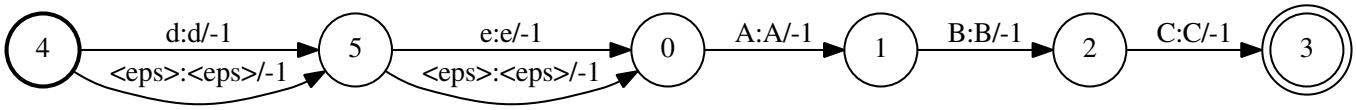


Figure 3.10: Initial alignment: Case 2

Momentarily taking a depth first approach in the case analysis by considering a second SH being aligned (rather than the rest of the first one), we stick to it being a hypothesis aligned to the first word of the PH to further inspect the special case. Initially, my approach was to insert all the arcs that occurred before the alignment with the first word *in between* the PH's start state and the states before it (note this assumes there are words already there, e.g. fig. 3.10, otherwise it is the same process done *for* 3.10). Whilst this is much easier to implement, it is a sub par solution as it would lead to long, low weighted sequences of words which make more sense aligned, given their shared position relative to the first word of the PH. Observe the difference when adding a new SH $(f\ g\ A...)$. Once again we add epsilon transitions for each path that wouldn't otherwise take a newly added arc. In the case of fig. 3.11, as we are adding new states for each arc, the two hypotheses so far (primary and first secondary) both contribute epsilon arcs. However as the newly added arcs don't take the paths of $(d\ e)$, these too must have epsilon weights incremented. Already we see it becoming unlikely that anything at all gets chosen using this incorrect method.[3]

[1] I use upper-case letters to represent words that appear in the PH and lower-case letters otherwise. There is no implicit relation between e.g. A and a

[2] N.B. Whenever I talk about "increasing" weights, the underlying effect is the internal value being

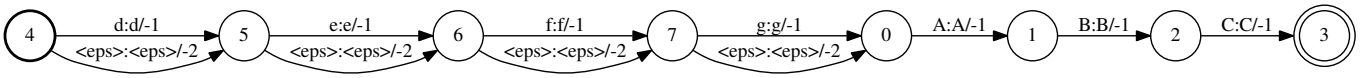


Figure 3.11: Initial alignment: Case 3 (incorrect)

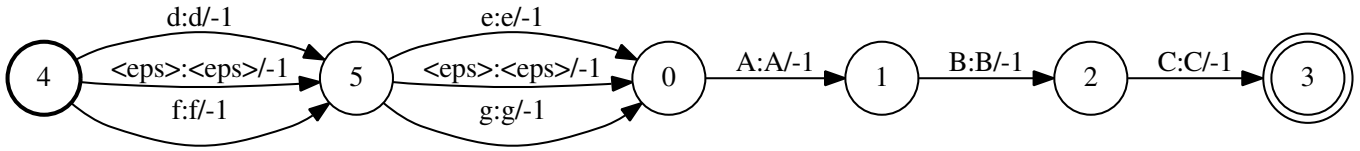


Figure 3.12: Initial alignment: Case 4 (correct)

Originally, I had tried to implement a 'word reordering' technique described in [6]. However, its implementation wasn't immediately clear, and trying to add it at an early stage complicated matters. The main point of contention was a part where it said "the new position gets an epsilon arc for the primary *and all unrelated secondary systems*", thus implying that secondary hypotheses should somehow be kept separate from each other and only connect them when necessary, leading to CNs like Figure 3.13.

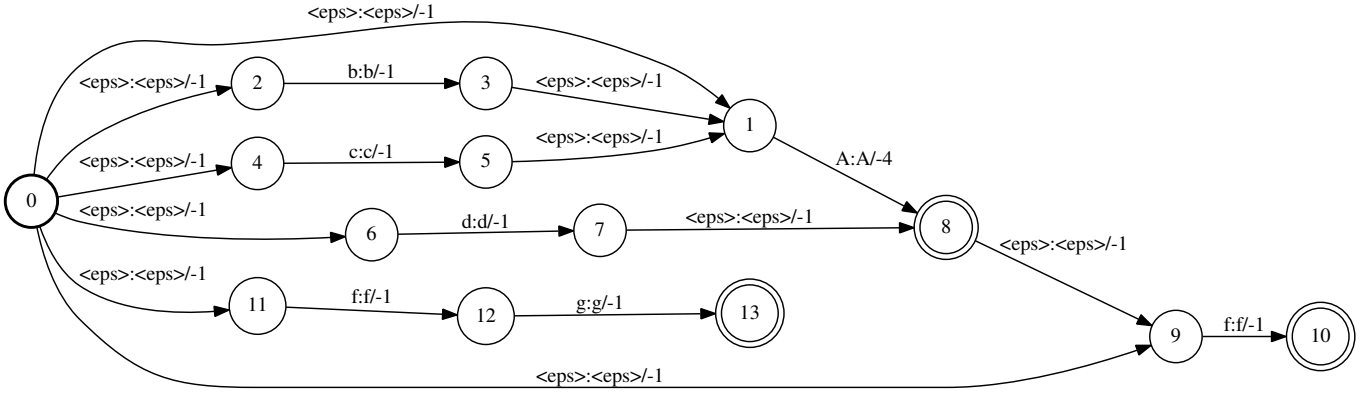


Figure 3.13: Non-linear hypothesis alignment (incorrect)

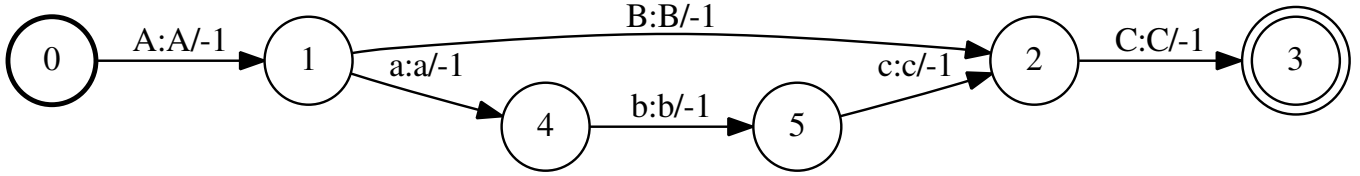


Figure 3.14: Non-linear multi-word alignment (incorrect). (B) is aligned to $(a\ b\ c)$

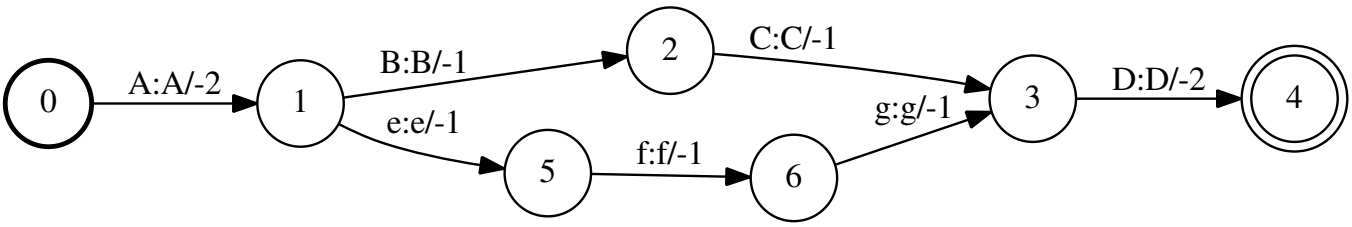
I was also considering at this time how to add alignments when multiple words were aligned to a single one, or more generally when alignments aren't of equal length. My initial approach was to branch around the word(s) being aligned, reconnecting at the state they end on. Unfortunately, if the CN is not kept linear, many useful properties are

decremented by 1, but I leave the terminology unchanged for conceptional consistency.

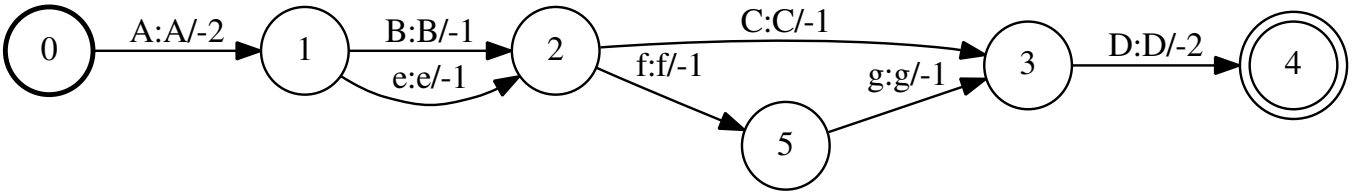
[3]The $A\ B\ C$ arcs have weights unchanged for now—remember these are partial steps in the algorithm, so the CNs given here are for illustration and are by no means final.

lost. The most noticeable is the damage it does to meaning of the weights; if you are on a branch point (say, state 1 in fig. 3.14) later alignments will cause misrepresentation of arcs further down each branch. As a result, we instead wait until the union operation at the end to make up for the lack of alignment between secondary hypotheses.

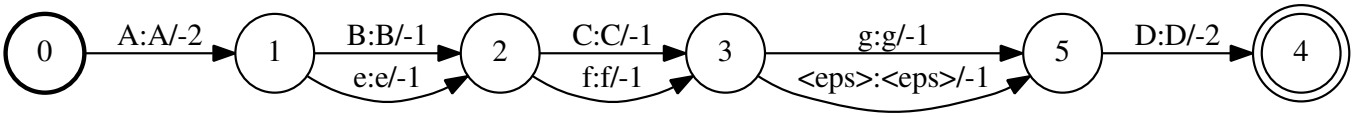
The special case of adding before the first PH state has been discussed, and the equivalent special case of adding after the final PH state can be imagined in a similar way. The more general case is aligning from some point within both hypotheses, after some known number of hypotheses have been aligned so far, when the currently aligned words may be non-equal and of differing lengths. With PH and SH as (A B C D) and (A e f g D) respectively, with alignment [e f g | B C] my first attempt was to add "around" as seen in fig. 3.14, with e.g. e leaving the same state as B and continuing until it rejoins where C finishes (fig. 3.15a). Before I realised the issues with non-linearity, this already caused problems when trying to align secondary hypotheses later on (think f g). My second attempt was add the SH words as alternate arcs to the PH words until there was one PH word left to align, at which point they would be added as new states (fig. 3.15b). This doesn't fully solve the previous problems, and still suffers for not being linear. My solution was to align up to and including the final PH word, after which I insert between that final PH word and the next PH word, or, if SH arcs have already been inserted, I align with those until they run out, then inserting before the next PH word as above (figs 3.15c and 3.15d).



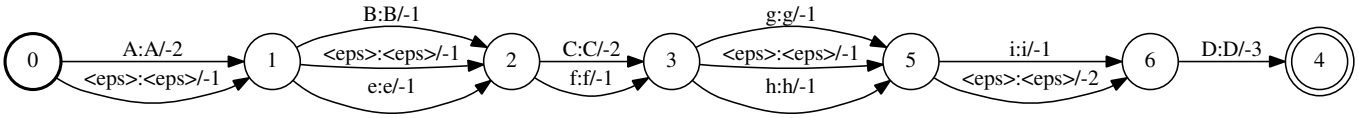
(a) First attempt (incorrect)



(b) Second attempt (incorrect)



(c) Final attempt



(d) Final attempt with *C h i D* aligned too

Figure 3.15: General-case alignment, multiple attempts

```
StateId insertArc(SVF *cn, StateId state0, StateId state1, Label label, int hypn) {
/*/ "Insert" an arc by redirecting arcs to a new state, adding the new arc from that
 state */
/*/ Or, if state0 < 0; add state [at start] but no arc redirecting */
/*/ Or, if state1 < 0; add state [at end] but no arc redirecting */
/*/ Then return new state's ID. */

  // Add the new state
  StateId newState = cn->AddState();

  // Change all previous arcs from 0->1 to this new state
  if(state0 >= 0 && state1 >= 0) {
    for(fst::MutableArcIterator<SVF> aiter(cn, state0); !aiter.Done(); aiter.Next())
 {
      fst::StdArc arc = aiter.Value();
      if(arc.nextstate == state1)
        aiter.SetValue(fst::StdArc(
          arc.ilabel, arc.olabel,
          arc.weight, newState)
        );
    }
  }

  // Add the new desired transition
  if(state0 <= 0 && state1 <= 0) {
    std::cerr << "Both state0 and state1 were less than 0!!" << std::endl;
  } else if( (state0 >= 0 && state1 >= 0) || (state0 <= 0) ) {
    state0=newState;
  } else if(state1 <= 0) {
    state1=newState;
  }

  cn->AddArc(state0, fst::StdArc(label, label, -1, state1));

  // Add additional eps transitions in number equivalent to the number of hyps done
so far,
  //  i.e. an eps for every path that would chose *not* to use this new transition
  if(hypn) createArcOrAddWeight(cn, state0, state1, 0, -hypn);
//  else std::cerr << "Couldn't add epsiolns - 0 weight" << std::endl;

  return newState;
}
```

Figure 3.16: C++ for adding an arc by inserting a new state and redirecting

3.2.3 Development

Here I outline the pseudocode for the final algorithm. The PH sentence and the SH sentence appear as **sentence 2 and sentence 1** respectively in the actual source code. This is simply because METEOR requires the reference sentence to be the second and the test sentence to be the first.

States are marked for a pass at the end of the algorithm that will add the necessary epsilon transition. I omit the simple initialisation of the confusion network, but state numbers follow order of creation, so the ith word of the primary hypothesis will be on an arc from state i to state $i+1$. Functions with included definitions have initial capitalised. Repetitions are replaced with ellipses for brevity.

```
void createArcOrAddWeight(SVF *cn, StateId state0, StateId state1,
    Label label, Weight w) {
  bool exists=false;
  // If an arc with the right label exists, just add the new weight on
  for(fst::MutableArcIterator<SVF> aiter(cn, state0); !aiter.Done(); aiter.Next()) {
    fst::StdArc arc = aiter.Value();
    if(arc.ilabel == label && arc.nextstate == state1) {
      exists=true;
      aiter.SetValue(fst::StdArc(
        arc.ilabel, arc.olabel,
        arc.weight.Value() + w.Value(), arc.nextstate)
      );
    }
  }

  // Otherwise add a new arc with the given weight
  if(!exists)
    cn->AddArc(state0, fst::StdArc(
      label, label, w, state1)
    );
}
```

Figure 3.17: C++ for adding an arc as new, or as a weight if it exists

```
1  initialise(CN, PH)
2  HYPN := 1 // Count the number of hypotheses completed so far (PH counts!)
3  for (each alignment ALN in file):
4      for (each word-alignment WALN in ALN):
5          /// (1):    Add all *aligned* words ///
6
7          // diff. in no. of words to align from each hypothesis
8          LENGTH_DIFF := WALN.PH_LENGTH - WALN.SH_LENGTH
9          if (LENGTH_DIFF = 0): // can align all WALN's SH words, 1:1
10             for (DIFF in [0 , WALN.PH_LENGTH) ):
11                 STATE := WALN.PH_START + DIFF
12                 N_STATE := nextstate(CN, STATE)
13                 LABEL := getlabel(ALN, WALN.SH_START + DIFF)
14                 markstate(STATE)
15                 CreateArcOrAddWeight(CN, STATE, N_STATE, LABEL)
16
17         else if (length_diff > 0): // align SH's, mark rest of PH's
18             for (DIFF in [0, LENGTH_DIFF) ):
19                 STATE := ...; N_STATE := ...
20                 LABEL := ...
21                 markstate(STATE)
22                 CreateArcOrAddWeight(...)
23             for (DIFF in [LENGTH_DIFF, WALN.PH_LENGTH) ):
24                 STATE := ...; N_STATE := ...
25                 markstate(STATE)
26
```

```
27          else: //(length_diff < 0)
28              for (DIFF in [0, -LENGTH_DIFF) ):
29                  STATE := ...; N_STATE := ...
30                  LABEL := ...
31                  markstate(...)
32                  CreateArcOrAddWeight(...)
33              for (DIFF in [-LENGTH_DIFF, WALN.SH_LENGTH) ): // note, SH
34                  // If there are SH-only states between here
35                  // and next PH state, align. Otherwise, add
36                  // new states.
37                  LABEL := ...
38                  if (N_STATE is PH-state): // N_STATE < PH length
39                      STATE := InsertArc(CN, STATE, N_STATE, LABEL, HYPN)
40                      markstate(STATE)
41                  else:
42                      STATE := N_STATE
43                      N_STATE = nextstate(CN, STATE)
44
45                      if (N_STATE doesn't exist): // N_STATE < 0
46                          // at end of CN
47                          N_STATE := InsertArc(...)
48                          markstate(STATE)
49                      else:
50                          CreateOrAddWeight(...)
51                          markstate(STATE)
52
```

So far, the pseudocode has handled the first stage of three of the final algorithm, that of adding all the words in the SH that are aligned to words in the PH. To take account of multi-word alignments, I case split on which side of the current alignment is longer. If this is the secondary, then I take care of the possibility of the arcs being added off the end the graph, which is handled differently in the `insertArc` function too. Continuing:

```
53      //for (...word-alignment...) loop closed
54
55      /// (2) Add all *unaligned* SH words ///
56      WORD_I := 0 // Index to word in SH we're currently on
57      while (WORD_I < ALN.LENGTH_SH):
58          if (aligned(WORD_I)):
59              WORD_I++ // (break)
60          else if (WORD_I = 0):
61              // Start case special, must work backwards
62              FA := firstaligned(CN, ALN) // Index of first aligned SH word
63              N_STATE := getstate(CN, ALN, FA) // State of first aligned word in SH
64              STATE := prevstate(N_STATE)
```

```
65
66              for (each word WI in [FA, 0] ):
67                  LABEL := ...
68                  if (primarystate(STATE)): Need to add new state
69                      STATE := InsertArc(...)
70                      markstate(STATE)
71                  else: // Add to states already created for SH arcs
72                      CreateArcOrAddWeight(...)
73                      markstate(STATE)
74                      N_STATE := STATE
75                      STATE := prevstate(N_STATE)
76
```

In the first two cases of the second stage, handling each unaligned word in the SH, we do nothing for an aligned word, and for the first case we insert *backwards* from the state of the first SH word to be aligned with the PH. If there are additional, "SH-only" states before the first previous PH state, we add the arcs to them. Once the state before the current one has a PH arc, we push it backwards by inserting new states for the SH arcs. The third case of this part:

```
77          else: // Case for general, unaligned, non-initial word
78              // STATE and N_STATE carry over from previous loop run
79              LABEL := ...
80              STATE := getstate(CN, ALN, WORD_I-1) // -1 as we ins. after
81              N_STATE := nextstate(CN, STATE)
82
83              while ( !aligned(WORD_I) & (WORD_I < ALN>LENGTH_SH) ):
84                  LABEL := ...
85                  if (primarystate(N_STATE)): PH state, insert before
86                      STATE := InsertArc(...)
87                      markstate(STATE)
88                  else: Add to SH states made earlier
89                      STATE := N_STATE
90                      N_STATE := nextstate(CN, STATE)
91
92                      if (N_STATE doesn't exist): // N_STATE < 0
93                          // at end of CN
94                          N_STATE := InsertArc(...)
95                          markstate(STATE)
96                      else:
97                          CreateArcOrAddWeight(...)
98                          markstate(STATE)
99
100             WORD_I++
101
```

This general case finds the correct state to insert after, then follows a similar procedure to the previous one, but in the forward direction. Finally, we add epsilon transitions to those states that weren't marked as having an arc added to them this run. Note that we only need to add -1 weighted arcs—the arcs weighted with the number of hypotheses aligned so far are handled when those states are created.

```
    // End of while (WORD_I...)

    /// (3) Add epsilon arcs to all *unmarked* states ///
    for (each STATE in CN):
        if (!marked(STATE)):
            N_STATE := ...
            if (N_STATE exists):
                CreateArcOrAddWeight(CN, STATE, N_STATE, 0)
```

After a CN has been generated for every input hypothesis, these are combined using the Union operation provided by OpenFST. Simply put, when given two CN, the first accepting strings $s_i \in S_1$ with weights $w_i \in W_1$, and the second $s_j \in S_2$ and $w_j \in W_2$, the union of the two CNs accepts strings $s_k \in S_1 \cup S_2$ with weights $w_k \in W_1 \cup W_2$.

Whilst the calling of the union function is easy enough, i.e. accumulating each CN with an initially empty output CN, a major complication arose with regard to the 'symbol tables'. These tables store the correspondences between the integers used internally to represent the different words on the arcs, and the words themselves, needed when drawing the CN and when extracting the paths for the final output. I had been storing these in symbol files (OpenFST's format) for each individual CN, expecting to be able to simply concatenate these when the time came to implement the union. However, although the documentation states[2] "you may use any non-negative integer for a label ID", it fails to mention that each ID must appear uniquely in the file—even if they share the same label. This led to significant changes to the way I handled symbols throughout my code, and required a major refactor, causing a significant delay.

The other two parts that needed to be implemented were the length penalty and the LM feature. If paths are extracted from the CN at the current stage, they are often too short. This epsilon dominance, caused in part by the alignments generated from METEOR being too sparse, is counteracted by giving a weaker score to a path the further it is in length from the average. When construction a length penalty function, I chose to make it quadratic, as the effect of this epsilon dominance was strong in all the test cases I executed.

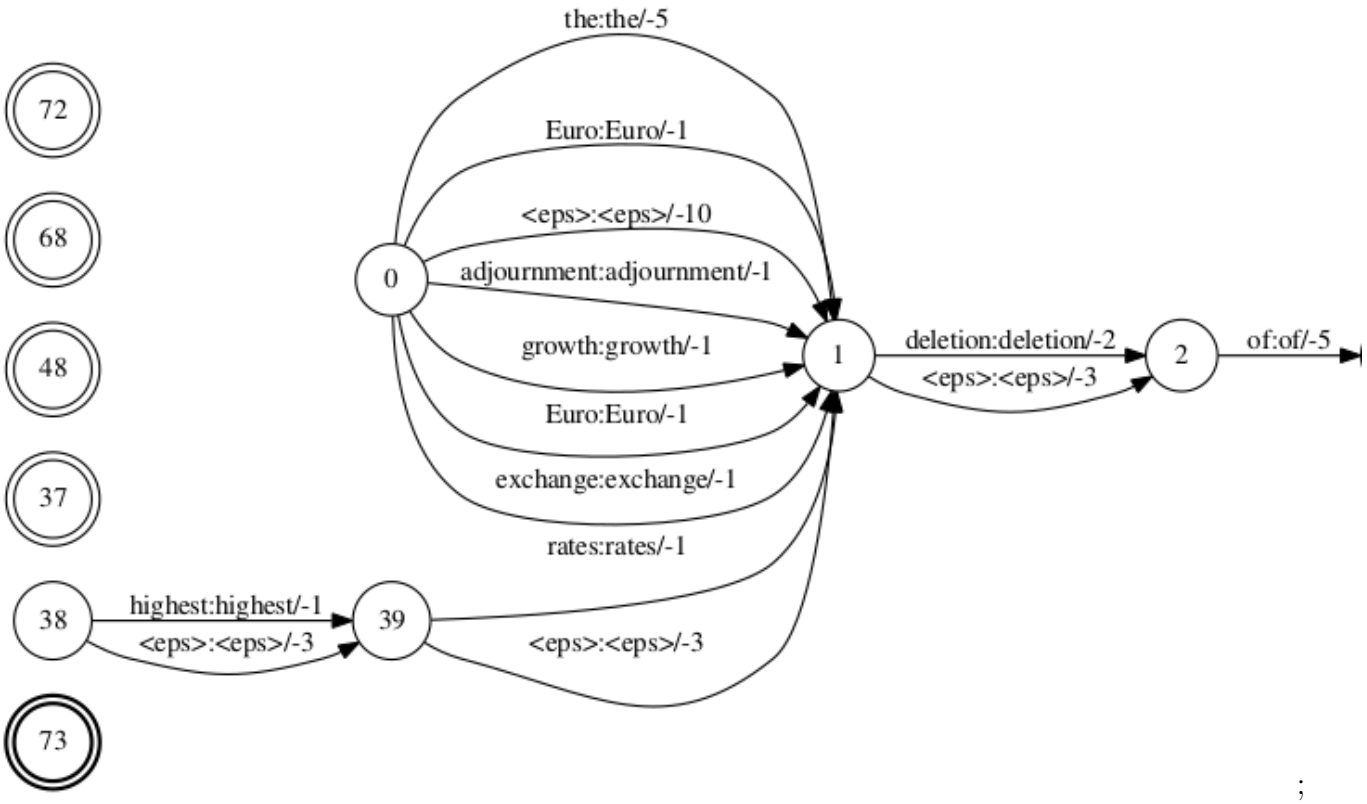
The next step would be to train a language model on all the input hypotheses themselves, then rank outputs using the score this LM gives them. This in turn prevents the

CN from outputting sentences that differ drastically in phrase choice or word order from the hypotheses, as the n-gram model employed by the LM (i.e. frequencies of consecutive word phrases n words long) discounts under both these scenarios. I didn't manage to complete this stage in my implementation, and I discuss this in later chapters.

3.2.4 Testing

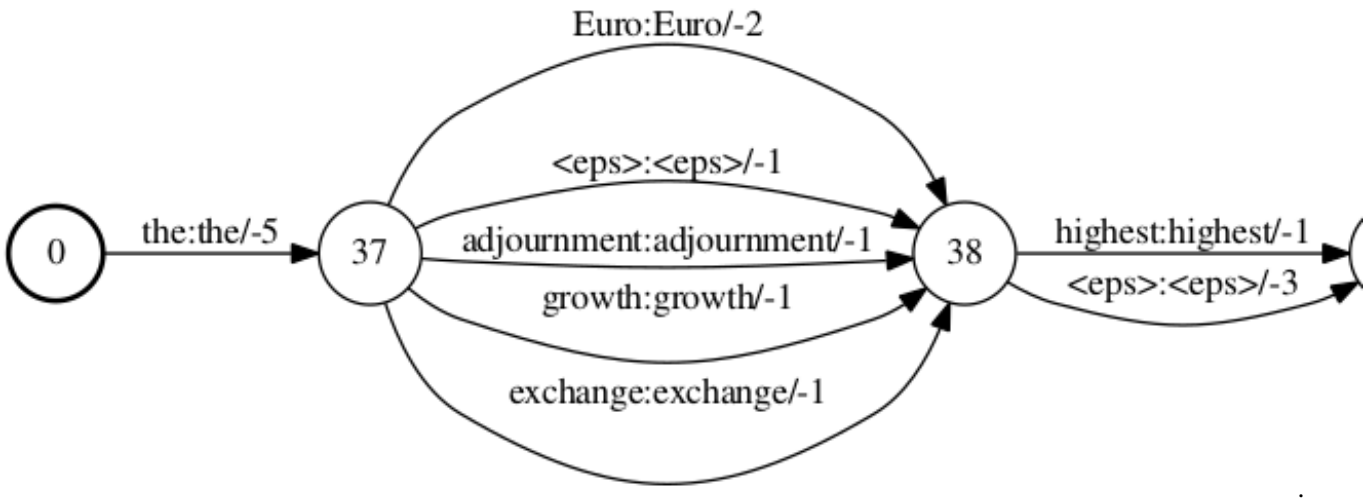
For testing the algorithm, in line with the strategy discussed, I combined manually selected data with manual checking to provide verification of each stage of the algorithm and overall quality control. As with much the rest of the system, data was abundant, so a subset was taken for detailed inspection and testing purposes.

The order in which the algorithm was outlined reflects the order in which it was developed, as such, each stage was tested when implemented. As the algorithm was first written by me on paper, very few modifications were necessary to the actual algorithm itself during its manifestation, so debugging concentrated on finding the program errors that could lead to erroneous output. Linearity assisted once again, as each stage could be tested sequentially and independently of the others. Some manual verifications were simple, as in the case of Figure 3.18a, which shows the result of a < check appearing where a <= check should have been. Most required the checking of each arc and weight, with the correct ones of course taking the longest to verify as every state had to be inspected manually. Figure 3.18b displays the graph with the error corrected.



;

(a) Erroneous CN showing the result of an off-by-one error



;

(b) Corrected CN

Chapter 4

Evaluation

4.1 Fulfilment of Project Aims

Evaluating in terms of the initial aims laid out at the start, and described in the introduction, I consider my project a success. All of the main aims were fully completed, and the further aims saw a high level of completion despite a large number of setbacks.

The baseline systems were all trained, and the pivot systems trained were both broad (covering every source-pivot pair) and effective, seeing improvements in multiple cases. Similarly, having chosen the second of the two further aims, my confusion network *algorithm* was completed, with only the weighting mechanisms standing in the way of it being completed in full and tested against real data.

I discuss the results of the above in this chapter.

4.2 Pivoting

4.2.1 Experiment Management System

Although the EMS was never an intentional part of the project, and although its implementation cost me a fair amount of time, it was certainly successful in its aims. It allowed me to train a large number of systems, and were the project to continue in scope, for example with varying corpus sizes, it would have been equally effective at speeding up the training pipeline and avoiding human error when handling many similar files.

It was fully completed, both in its implementation (raw corpus to pivoted BLEU scores, from one .ini file), and its testing—although a large number of language pairs were used, it was a finite one, and I was able to test every one using a small "toy" corpus, before they were run through the EMS to be trained in full. Due to its complete state, and its effective abstraction, my EMS is also arguably the part of the project currently most suited to reuse from someone entirely external.

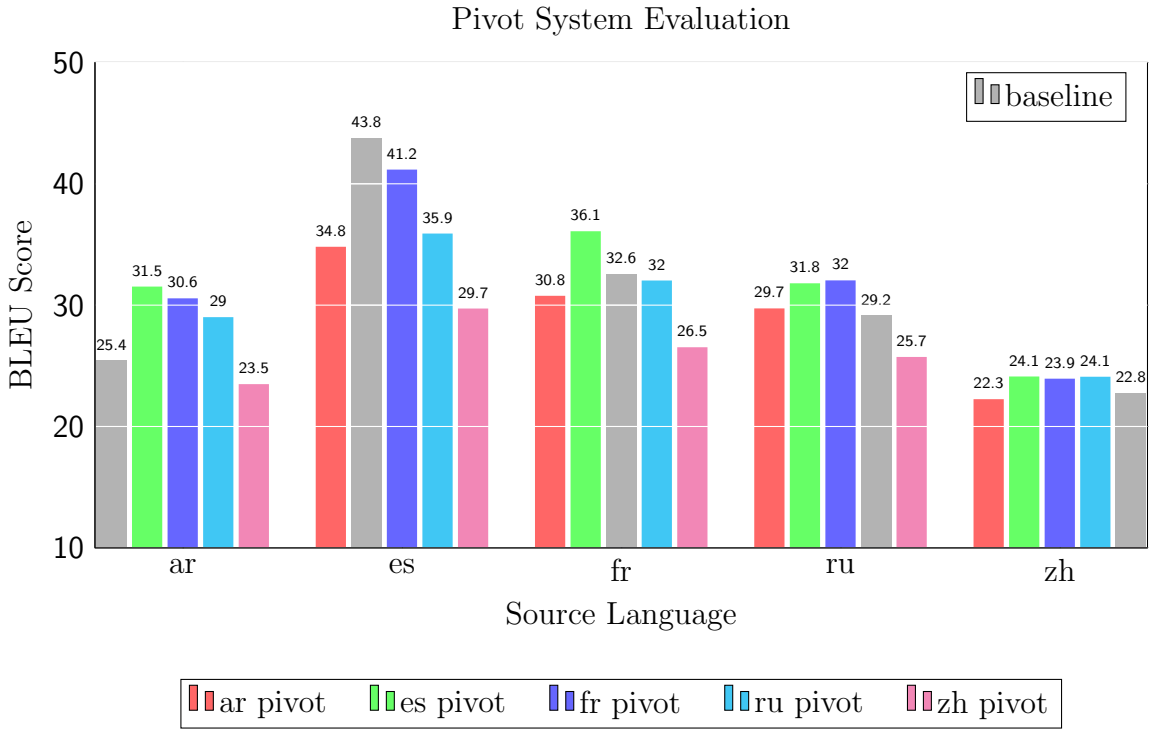


Figure 4.1: System comparison

4.2.2 BLEU scores

To evaluate the quality of a MT output, I use BLEU scores.[9] The BLEU algorithm estimates the similarity between a reference (i.e. human translated) text and an attempted translation (i.e. an MT translated text), sentence by sentence, averaging over the whole text and outputting a percentage score. BLEU uses a modified form of precision, to avoid e.g. "the the the..." from getting a perfect precision score, by limiting the word-count for each word to the number of times it appears in the reference translation. BLEU is a widely used tool for its purpose, and comes with Moses by default.

4.2.3 Results

As figure 4.1 shows, a considerable number of pivot systems showed improvements over their baselines. The most noticeable exception are the es source systems. As the strength of the baseline for es shows, es systems are especially strong, even at small corpus sizes. This effect carries through to the es-en pivot system part too, as those systems using this part (i.e. bars in green) are all the strongest in their group with the exception of ru. ru seeing best results when pivoting via fr is interesting, as historically, French has had influences on the Russian language (French's Tu / Vous is paralleled by Russian's Ti / Vi for example), so this may be a case of language similarity overpowering the effect of just using a strong system component.

fr shows the second strongest baseline, albeit 11 points worse than es'. This is consistent with its good performance across the board, being only marginally weaker than es pivots for ar and zh sources, and beating it for ru as mentioned. fr also does very well with a es source, only being beaten by the es baseline by 2 points.

Despite ru's weak baseline, it generally performs well as a pivot, only just losing to fr for an ar source and marginally beating it for a zh one. However, the all round low scores for zh, and the grouping close to its baseline all suggest that the weakness of any zh component in a system dominates that system's score. In line with the language similarity claims made, ru comes within half a point of fr's baseline score; a full 3 points stronger than its own baseline.

ar beats zh as a pivot for all but zh's baseline, and performs comparably to ru for es and fr sources, even managing to beat ru's baseline.

These results conclusively prove, at least for corpus sizes of 50,000 vs 500,000 words, that system combination via pivoting can cause significant improvements in BLEU score against a baseline, for particular language pairs.

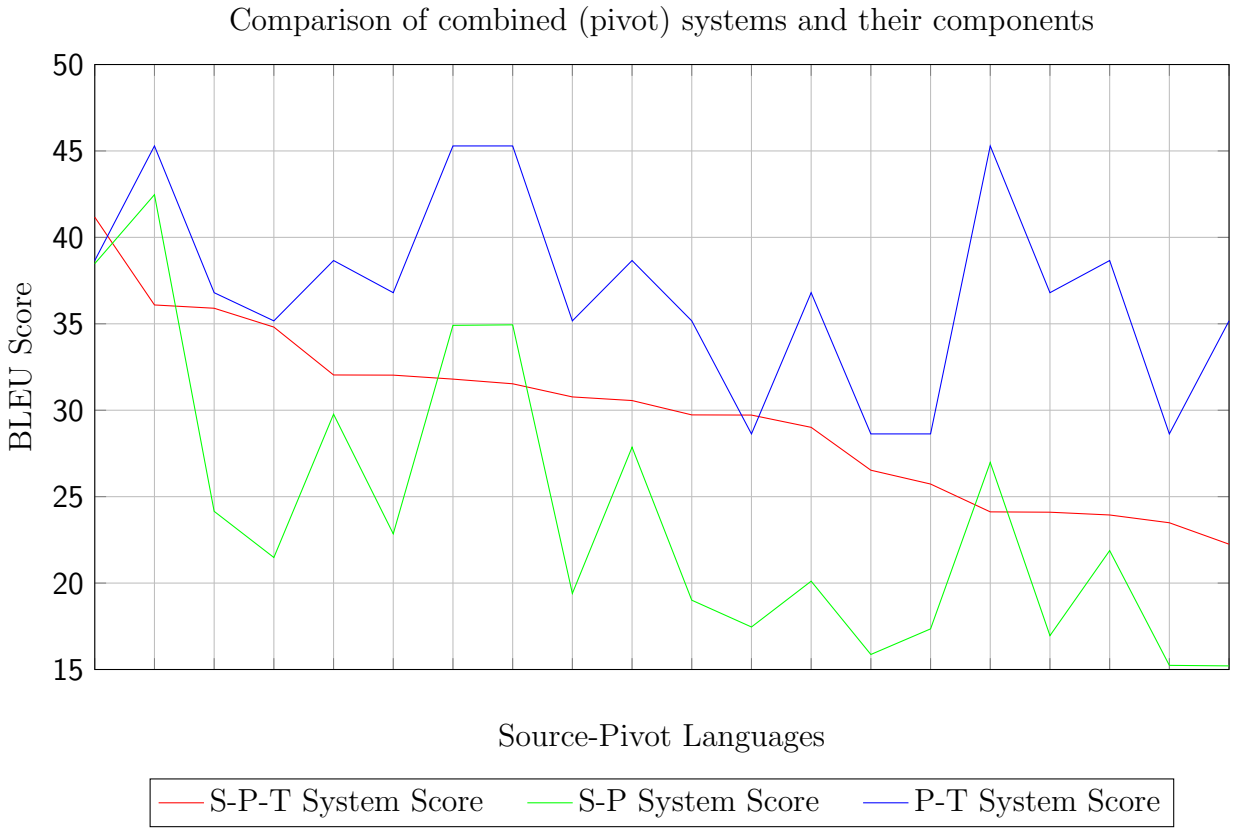


Figure 4.2: System component comparison

The general downward trend of 4.2 shows the expected, that weaker components make weaker systems. Interestingly, there are only two points where both components were weaker than their combination, at the strongest point and in the middle. Components being stronger as well as weaker, as well as both components being stronger, appears throughout the graph, suggesting that component strength alone is less significant a factor than one might initially assume.

4.3 Confusion Networks

The results of the CN part of my system combination pipeline are inconclusive. Ultimately, to accurately evaluate this stage would have required it to be applied to the full data, in this case the outputs of all of the 25 pivot systems trained. It is worth highlighting the fact that the entire "further" system combination (i.e. beyond pivoting) part of the project was technically an extension—one that I intended to reach, but regardless one that was both attempted as an add-on to the main project and also attempted despite the severe setbacks faced during the course of the year. With these facts in mind, even though the CN stage didn't reach the point at which it could be properly evaluated, I consider it a success that the main algorithm was fully implemented.

As has already been extensively discussed in the implementation chapter, the main part of the algorithm, that combines a set of secondary hypotheses with a primary hypothesis to which they each have word alignments, was completed in full and tested extensively. Beyond this, the union operation had also been successfully put in place, allowing all the CNs generated using each hypothesis as primary to be combined.

I had also implemented a promising length penalty function, which worked as a quadratic function of the difference in length between a path and the average, to punish more harshly than a linear one. A contributing factor for this decision was my reliance on my word aligner's (METEOR) precision, even at the cost of recall. This resulted in fewer alignments, meaning a greater weight to the epsilon transitions, giving shorter paths on average.

However, for this to be properly implemented, rather than operating on the paths that are generated at the end, it should be integrated into the CN itself to contribute to the weight of each arc. Similarly, the LM feature, which wasn't implemented, should have been applied in a similar manner. This allows for far fewer paths to be extracted from the CN, as paths that would have previously been extracted and then disqualified afterwards by one of these models won't be extracted at all. This would drastically improve the quality of results as well as being much more efficient.

4.4 Testing

Another benefit of picking a large corpus, other than the increase in system quality, is the abundance of test data it provides. This gives the crucial diversity needed to be certain that the tests taken are not succeeding on an isolated case. My second key testing strategy came from my assurance of linearity at the system level. The modularity given was extremely effective in isolating each stage for testing. This linearity also *creates* this need, as the next stage cannot be tested until test data can be generated for it from the completed stage that sits before it in the pipeline.

The system's interactions being solely with natural language affects testing in multiple ways. The most noticeable is that every output of every part of the system is human readable—thus its validity may be checked by manual inspection. However, a corollary of that is that it is almost impossible for a lot of these results to be checked in an automated

manner. In many cases, this could be a serious blow to the effectiveness of any large scale testing that is attempted, but in my case I believe it to be of more benefit than detriment.

The reasons for this are data 'cleanness' and data 'similarity'. The very first stages of the entire pipeline, namely tokenisation, truecasing, and cleaning (i.e. length sentence filtering), provide absolute assurance[1] that the data used at every other stage in the entire system will be *clean*. When normally dealing with natural language, one can always imagine an unseen test case with perhaps some strange punctuation causing a failure—data cleanness provides a safety net against such possibilities. Along the same lines is the fact that all the data comes from a high quality, domain specific corpus (UNPC). This provides yet another assurance on the cleanness and expected content of the text used throughout the system.

These effects are not entirely beneficial, as data similarity comes at a cost. From a MT perspective, a system is likely to only be effective on the type of data it is trained on. As a result, my systems are effective on the UN meeting transcriptions that they are trained and tested on (as the results show), and assuming similar styles are used in similar organisations, they may also be effective on e.g. European Union minutes. However, they will perform worse on general text, for example translating novels. For my system as a whole, as all the components are theoretically independent of the corpora they are used on, this data similarity means that whilst I expect it to work as already tested, I cannot be *certain* that the differing formats a different corpus could provide wouldn't cause a component failure. The main reason I think such a failure to be unlikely is the data cleaning step, which should make any corpus differences entirely semantic, and thus of no concern to the overall system itself.

[1] Of course, assuming the correct function of those external tools used

Chapter 5

Conclusion

Whilst my project was largely a success, there were major issues with timing, due to several delays that were beyond my control.

Despite Moses' widespread use within the academic SMT community, my project's initial progress was marred by numerous installation and post-installation errors that prevented me from getting even a baseline system working. As the complexity of the install had prevented any serious attempts prior to the finalisation of the project proposal, I was relying on the documentation's assurance of both Linux support (specifically Fedora) and OS X support. The issues began with the dependencies, with even Ubuntu's list (it being by far the best supported of the OS's) appearing differently in two places of the same section. Despite Fedora's supposed support, and the ease of an inbuilt package manager, I ran in to so many installation issues that I had to abandon my attempts completely, and concentrated on trying to get a working system on my Macbook (both of these computers were mentioned in my proposal).

I eventually managed to get a mostly complete installation operative on my Macbook, despite numerous problems with external tools and dependencies (some requiring source editing to fix), misleading information from the manual ("optional" dependencies being later required and requiring a full reinstall), manual inconsistencies (regarding C++ compilers and their corresponding platforms), and a serious manual formatting error (a critical part of a command being cut off the edge of the page!) which was kindly amended at my behest.

This veritable barricade of unforeseeable complications cost a huge amount of time, setting my project back by almost an entire term.

In spite of these set backs, I managed to achieve all of my original goals, and the core algorithmic contribution of my own writing of the original extension. Furthermore, the pivoting technique I applied gave very good results, with gains of even a few BLEU points considered a big improvement in the SMT community. Were I to redo the project with hindsight, I would have concentrated on training fewer pivot systems, e.g. the ones I expected to see the greatest gain in, to make more time for the CN stage, in the hope that it could have been completed in full.

Bibliography

[1] KenLM language model toolkit. https://kheafield.com/code/kenlm/.

[2] Openfst guide. http://www.openfst.org/twiki/bin/view/FST/FstQuickTour.

[3] Stanford word segmenter. https://nlp.stanford.edu/software/segmenter.shtml.

[4] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. OpenFst: A general and efficient weighted finite-state transducer library.

[5] Mauro Cettolo, Nicola Bertoldi, and Marcello Federico. Bootstrapping arabic-italian smt through comparable texts and pivot translation, 2011.

[6] Markus Freitag, Matthias Huck, and Hermann Ney. Jane: Open source machine translation system combination, 2014.

[7] Philipp Koehn, Alexandra Birch, and Ralf Steinberger. 462 machine translation systems for europe, 2009.

[8] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation, 2007.

[9] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation, 2002.

[10] Masao Utiyama and Hitoshi Isahara. A comparison of pivot methods for phrase-based statistical machine translation, 2007.

[11] Hua Wu and Haifeng Wang. Pivot language approach for phrase-based statistical machine translation, 2007.

Appendix A

Intelligent ini creation

Page 1

```
newctrl() {
  echo "]] Creating new skeleton control file..."
  echo \
"Please use absolute paths.
If lm dir is already defined, a language model is expected.

corpus dir=$NPJ/corpus/unpc/
corpus stem=
source language=
target language=

devset dir=$NPJ/corpus/unpc/devset
devset stem=devset
testset dir=$NPJ/corpus/unpc/testset
testset stem=testset

lm dir=$NPJ/lm/unpc/
lm stem=
lm order=3

working dir=$NPJ/working/unpc/

min sentence length=1
max sentence length=80
cores (train)=8
thread number (tuning)=8

pivot dir=none
" > $1
}

inewctrl() {
# expects pN-bl.ss-tt
#      or pN.ss-tt

  cd $1
  SKEM=$(basename $PWD)
  PAIR=$(echo -n $SKEM | tail -c 5)
     S=$(echo -n $PAIR | head -c 2)
     T=$(echo -n $PAIR | tail -c 2)
    DS=$(cd $PWD/../devset; pwd)
    TS=$(cd $PWD/../testset; pwd)
   LTM=$(echo -n $SKEM | head -c -6)
   LMD=$(echo -n $PWD | sed 's/corpus/lm/')
    LM=$(dirname $LMD)/${LTM}.$T

   WRK=$(echo -n $PWD | sed 's/corpus/working/')

  if [ -z $CC ]; then
    CRS=8 # Default cores
  else
    CRS=$CC
  fi

  SEP=$(echo -n $SKEM | head -c 3 | tail -c 1) # `.` if pivot, `-` if baseline
  if   [ $SEP = "-" ]; then PIVDIR="none"      # but not pivot if it's -en!!
  elif [ $SEP = "." ]; then
    if [ $T = "en" ]; then
      PIVDIR="none";
    else
      PIVDIR=$(echo -n "$PWD/../${LTM}.${T}-en" | sed 's/corpus/working/');
    fi
  else
    echo "ERROR! Seperator (e.g. [p0]. or [p0]-) was <$SEP>"; exit 1000
  fi

  echo "]] Attempting intelligent skeleton creation..."
  echo \
"Please use absolute paths.
If lm dir is already defined, a language model is expected.
```

Page 2

```
source language=$S
target language=$T

devset dir=$DS
devset stem=devset
testset dir=$TS
testset stem=testset

lm dir=$LM
lm stem=$LTM
lm order=3

working dir=$WRK

min sentence length=1
max sentence length=80
cores (train)=$CRS
thread number (tuning)=$CRS

pivot dir=$PIVDIR
" > ${SKEM}.ctrl
}

if [ -d $ARG ]; then
  echo "]] $ARG is a directory"
  inewctrl $ARG
  exit 42
elif ! [ -f $ARG ]; then
  echo "]] $ARG doesn't exits, creating..."
  newctrl $ARG
  exit 43
fi

if [ -f $ARG ]; then
  echo "]] $ARG is a file"
  echo "]] Processing control file..."
  s() {
    cat $ARG |sed -n "s/$1=\(.*\)/\1/p"
  }
  CORDIR=$(s "corpus dir");
  CORSTM=$(s "corpus stem");
       S=$(s "source language");
       T=$(s "target language");
  MINSEN=$(s "min sentence length");
  MAXSEN=$(s "max sentence length");
   LMDIR=$(s "lm dir");
   LMSTM=$(s "lm stem");
   LMORD=$(s "lm order");
  WRKDIR=$(s "working dir");
  TRNCRS=$(s "cores (train)");
  DEVSTM=$(s "devset stem");
  DEVDIR=$(s "devset dir");
  TUNTHR=$(s "thread number (tuning)");
  TSTSTM=$(s "testset stem");
  TSTDIR=$(s "testset dir");
  PIVDIR=$(s "pivot dir");
  echo "]] Control file processed."
  echo "]]"
```

Appendix B

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Machine Translation for Resource Poor Language Pairs via Pivoting

Charlie Seymour, Corpus Christi College
Originator: Zheng Yuang
19 October 2016

Project Supervisors: Zheng Yuang, Dr E. Kochmar
Director of Studies: Dr D. Greaves
Project Overseers: Dr P. Lio & Dr T. Griffin

Introduction

Some of the most promising advances in Machine Translation (MT) in recent years have come from methods of Statistical Machine Translation (SMT), in which parallel corpora that consist of the same text in two languages are used to train systems in translation between those languages. When these are properly aligned at a sentence level they are known as a bitext, and are a high quality resource without the noise caused by poorly aligned or poorly translated parallel corpora. These systems are also trained on single language texts to give them a better understanding of the appearance and structure of the target language.

Inherent to the success of such methods is the availability of bitexts for training purposes. Companies which have showed success in utilising these techniques, such as Google's translation service, have had their success limited in those language pairs for which such bitexts are not so readily available. Google Translate's relative quality in common language pairs, such as English and Spanish, is mirrored almost exactly in its lack of quality for uncommon language pairs, for example Chinese and Arabic. This is

due to the lack of websites carrying text in both those languages - such sites being the source of Google's extensive corpora.

For resource poor language pairs such as these, it can be beneficial to introduce a third language, for which both the initial languages have sufficiently strong parallel corpora, and translate from one to the other via this third, "pivot" language.

Starting point

[1] cite the lack of sufficient parallel data to be a major bottleneck for SMT between certain language pairs, such as Arabic and Italian, finding success when using English as a pivot language between the two. Another advantage of using a pivot when dealing with many languages is the reduction from order quadratic to linear in the number of systems needed; [2] using the JRC-Acquis, a parallel corpus of EU Law in 22 languages with over a billion words, reduced their required language pair systems from an initial 462 to 42 by simple chaining using a pivot language. They found a general increase in performance when using English as a pivot, and small but consistent gains when using a novel multi-pivot system.

This isn't to say that using a bridge language is without problems. Naturally, when using imperfect systems, such as SMT systems in their current state, multiple applications can lead to errors being compounded, thereby lowering overall translation quality. [3] found a slight reduction in performance when using two pivoting techniques with Spanish, French and German. [4] didn't find an improvement when using a bridge language over direct MT, although they improved the direct system using phrases learnt via pivoting.

A strong, sizeable corpus is a key component of any project involving pivoting. I will be basing mine primarily around the United Nations Parallel Corpus [6]. The UNPC contains documents from 1990 to 2014 that have been manually translated into the six official language of the UN, namely Arabic, Chinese, English, French, Russian and Spanish, including sentence level alignments. As such it is an extremely rich resource, very well suited to the task, and one released under a liberal license. Despite this, it has only been recently released (May 2016) so this will be one of the earliest works to utilise it. However, despite the quality of the UNPC, its domain is somewhat limited, and as a result I plan on integrating further corpora into the system as the project goes on, such as the OpenSubtitles corpus [7], extracted from film and television subtitles.

As building a fully functioning SMT system is a mammoth task involving decades of research worth of techniques, I will be relying on external tools for the bulk of the translation work, concentrating instead on the use and combination of them to try to improve the translation quality. This system combination is practically an area of research in its own right, and will involve careful planning and execution to see any visible success against some of the problems highlighted above.

Some of the corpora used, particularly those other than the UNPC, may require pre-processing, such as word alignment. The standard tool in current use for this task is *GIZA++*, a program based on statistical models developed by IBM's work on Machine Translation in the 1980s. Sentence alignment shouldn't be necessary for the UNPC as it

is already aligned, however any parallel corpora that do require it can be aligned using *hunalign*, a widely used tool based on the Gale-Church alignment algorithm (which itself operates based on sentence length).

As for the SMT system itself, I will be using Moses. This is currently one of the most widely used open source SMT systems in research, and increasingly in industry too. Moses' two primary components are its training pipeline, and the decoder. The training pipeline is a collection of tools that turn both parallel texts and monolingual (source language) texts into a Machine Translation Model. The decoder is a single C++ application that takes a source sentence, an MT model, and a language model, and produces a target language sentence. The last key part of the system comes from external tools, which are used to create this language model from monolingual data in the target language. Details on where these components come from and how they are used will be discussed in the following section.

Project Description

Concepts

The SMT system I will be implementing will use phrase-based translation. For the sake of description, we shall consider a system built to translate from some foreign sentence $\mathbf{f}$ to some English sentence $\mathbf{e}$. Under the phrase-based MT model, the input sentence $\mathbf{f}$ is split into sequences of consecutive word(s), each of which is translated into English (with some possible reordering). To calculate the probability of sentence $\mathbf{e}$ being a translation for $\mathbf{f}$ we use Bayes' rule as follows [8]:

$$\text{argmax}_{\mathbf{e}} p(\mathbf{e}|\mathbf{f}) = \text{argmax}_{\mathbf{e}} p(\mathbf{f}|\mathbf{e}) p(\mathbf{e})$$

The two terms on the right are the source of the language model and the translation model. The language model, $p(\mathbf{e})$, is the probability that the resulting target sentence is a reasonable sentence in the target language, and as a result this trigram language model is created from monolingual data, as it is a representation of how a well formed sentence appears. Bayes' rule allows us to now have an entirely separate translation model, $p(\mathbf{e}|\mathbf{f})$, this is the model that Moses builds as it is trained on parallel corpora. It is further broken down into two more elements; a probability distribution for phrase translation, and a distortion probability distribution to model the reordering of the English output phrases.

The final element introduced is a word cost factor, to help calibrate the output length. It is introduced for each generated English word, and it is usually greater than 1 to favour longer output, but can be changed during tuning in situations that require it.

Work items and algorithms

Once the corpora have been transformed into a processable format, it will be used to train Moses to create a "baseline" set of translation results that will be scored using the BLEU system, an algorithm that evalutes machine translated text to estimate the correspondence

between it and that of a human translator (the details of which will be discussed in the following section). These measurements will be used as a point of reference to determine the results of later system combination and tuning. The parallel corpora will be used to train a translation model, and monolingual text will be used with the external tool to create a language model.

There are multiple current techniques for system combination.

Sentence Pivoting As described by [5] and [3], this is the form of pivoting thus described, and will be the main focus for system combination considered in this project. It is the translation of a sentence in language A to a sentence in language C by using a direct translation system from A to a third language B, then using this best estimate of a translation with another direct translation system from B to C. This allows you to bypass weak A-C corpora via training done using strong A-B and B-C corpora.

The main issue with this method is the compounding of errors introduced in the initial source-pivot translation; when it is used in the pivot-target translation any mistakes present can push the final translation further from the desired result. However, for sufficiently resource poor language pairs, the lack of a strong corpus usually outweighs these issues.

Choosing appropriate languages for any kind of pivoting will have a large impact on performance. Experimentation will be the best way to determine the appropriateness of different pivots for different source-target language pairs.

Phrase pivoting As described by [5] and [3], this technique, also described as triangulation, involves the merging of phrase tables generated from the source-pivot and pivot-target tables. This can then be used to train a new source-target translation system. This has the benefit of reducing the compounding errors found when using sentence pivoting, and as a result usually benefits from a performance increase compared to that technique. However, table merging is certainly not a trivial problem, and so this method is considered as an extension.

Multi-pivot Described as a novel approach by [2] based on an earlier similar method, multi-pivot system combination utilises direct translation systems with several pivot systems. The multiple translations are then compiled into a word lattice, which is then searched for the most likely translation, with the aid of a language model. The similarity between the pivot language and the source and target languages will impact the quality of the translation, so this technique could help with the translation of linguistically distant languages.

This is a computationally expensive technique that relies on very large corpora across many languages. As a result, I consider it as a further extension.

Success criteria

The metric I will use to judge the success of my system will be BLEU scores. The BLEU algorithm is widely used in research as an effective automated means to measure the quality of a machine translation output, with quality being measured by an approximation of "closeness" to that of a human translator's output. Scores are given per sentence in comparison to reference translations. These scores are averaged over the whole corpus, giving an estimation of the translation's quality. The scores are given as a number between 0-1, although they are often quoted as a percentage.

The two metrics typically used for classification problems such as MT evaluation are precision and recall. Precision measures the fraction of the selected items that were relevant, namely the number of true positives divided by the number of items selected. Recall measures the fraction of relevant items that were selected, namely the number of true positives selected divided by all the possible true positives.

The issue with using a simple precision metric for evaluating MT is that it is possible to have a very high or even perfect precision with a very poor translation. The example given by [9] is the sentence "the the the the the the the" as a candidate for two references "the cat is on the mat" and "there is a cat on the mat". Whilst this translation is obviously nonsense, all 7 words appear in both translations, giving it a precision of 7/7 - a perfect precision for a useless translation.

BLEU circumvents this problem by using a modified form of precision. It does this by taking a maximum total count in any of the reference translations for each word in the candidate translation, and uses this to cap the word count for that word. In our above example, this max would be 2, giving the nonsense candidate a much fairer precision of 2/7. This method is generalised to more optimal n-grams.

When averaging over the whole corpus, sentence scores are combined with a geometric mean, weighted by a brevity penalty to prevent very short candidates from receiving an unfairly high score.

Whilst BLEU is far from a perfect measurement system, when working with large corpora, its inaccuracies tend to average out and as a result, even slight increases in BLEU scores is usually indicative of a successful improvement.

Resources required

I will mainly use my own laptops (MacBook Pro (2014, OS X) 2.8GHz, 8GB; ASUS S200E (Fedora 22) 1.8GHz, 4GB) simply for convenience; I plan on using my MCS drive as a network disk for back up, as well as regular backups onto USB drives. No extra resources should be required.

Timetable

Planned starting date is 24/10/16.

1. **Michaelmas weeks 3-4: 20 Oct - 3 Nov**

Initial preparation

This will involve setting up the appropriate work environment, including version control (I plan to use git), the development environment (C++ as it is common with most of the tools I will be using), and backup as mentioned in the previous section.

The aforementioned tools and corpora will be downloaded and installed. I will prepare the corpora for Moses training.

2. **Michaelmas weeks 5-6: 3 - 17 Nov**

   Baseline measurements

   After familiarising myself with Moses and its toolkit, I will begin training MT systems in various language pairs. This will involve selection of the appropriate parameters and tools. Once I have some working translation systems, I can make some slight tuning adjustments and begin scoring the systems to make my baseline measurements (BLEU scores) for future comparisons.

3. **Michaelmas weeks 7-8: 17 Nov - 1 Dec**

   Pivot planning

   With baseline measurements achieved, I will complete the necessary detailed research into the intricacies of sentence pivoting and consider appropriate language choices based on the results of the measurements already obtained. I will build an outline plan of the implementation of the pivoting system itself.

4. **Christmas vacation: 2 weeks from 2 Dec - 17 Jan**

   Begin pivot implementation

   Using this planning, I will start to write and test the pivoting system. This will involve the training of many MT systems, which is a time consuming process. In order to have something to show for the progress report, I will not focus on language choice at this stage, instead trying to get a workable implementation of the pivot system that can demonstrate the applied techniques.

5. **Lent weeks 1-2: 19 Jan - 2 Feb**

   Continued implementation, progress report

   I will continue to test and improve the pivot system, and in the second week of Lent term I will write my progress report.

6. **Lent weeks 3-4: 2 - 16 Feb**

   Finish pivot system, begin results analysis

   I will finish the writing and testing my pivot system, and begin work on results analysis. This will involve careful inspection of BLEU scores and analysis of which language are most appropriate for pivoting between certain language pairs. This will be the first point that I actively concentrate on maximising the BLEU scores that my system can produce.

7. **Lent weeks 5-6: 16 Feb - 2 March**

   Results analysis and system improvement

   At this point I will be iteratively improving my system, similar to as described above.

8. **Lent week 7 to Easter week 2: 2 March - 4 May**

   Dissertation planning and writing

   I will ensure that I keep good notes throughout the project to assist with writing the dissertation, planning to finish with ample time to prepare for my final examinations.

Extensions will be considered depending on progress, with the most likely point for the project to fork being after a working sentence pivot system has been successfully written and tested and shows improvements over the baseline system. If I have ample time, I can instead work on an alternative combination technique rather than on refinement of the current one.

Bibliography

[1] Mauro Cettolo, Nicola Bertoldi, Marcello Federico, 2011. *Bootstrapping Arabic-Italian SMT through Comparable Texts and Pivot Translation.*

[2] Philipp Koehn, Alexandra Birch, Ralf Steinberger, 2009. *462 Machine Translation Systems for Europe.*

[3] Masao Utiyama, Hitoshi Isahara, 2007. *A Comparison of Pivot Methods for Phrase-based Statistical Machine Translation.*

[4] Hua Wu, Haifeng Wang, 2007. *Pivot Language Approach for Phrase-Based Statistical Machine Translation.*

[5] Nizar Habash, Jun Hu, 2009. *Improving Arabic-Chinese Statistical Machine Translation using English as Pivot Language.*

[6] Michal Ziemski, Marcin Junczys-Dowmunt, Bruno Pouliquen, 2016. *The United Nations Parallel Corpus v1.0.*

[7] Pierre Lison, Jorg Tiedemann, 2016. *OpenSubtitles2016: Extracting Large Parallel Corpora from Movie and TV Subtitles.* 10th International Conference on Language Resources and Evaluation (LREC 2016)

[8] Philipp Koehn, 2016. *MOSES Statistical Machine Translation System, User Manual and Code Guide.*

[9] Kishore Papineni, Salim Roukos, Todd Ward, Wei-Jing Zhu, 2001. *Bleu: a Method for Automatic Evaluation of Machine Translation.*