University of Alberta

Word Similarity Using Pair Hidden Markov Models

by

Wesley C. Mackay 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Your file Votre référence
ISBN: 0-612-95811-6
Our file Notre référence
ISBN: 0-612-95811-6

To my family for their constant and enthusiastic support

Acknowledgements

I would like to thank the members of the Natural Language Processing Group of the University of Alberta, for their discussions, suggestions, and advice. This research was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Alberta Informatics Circle of Research Excellence (iCORE).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

El gato del blanco muerde los pescados [1] Without any knowledge of Spanish it is difficult for an English speaking person to determine what this sentence might mean. However, information specifically about Spanish is not the only way that the sentence's meaning can be determined. Consider for example Table 1.1 which contains a short list of Italian words along with their equivalent in English. The translations were generated by Babel Fish [2] since it was freely available online. Italian and Spanish are both from the same language family (in this case Romance languages) and are known to have similar words.

When examining Table 1.1 we can begin to see similarities between Italian words and those in the Spanish sentence. For instance, *bianco* (white) has a great deal of similarity to *blanco* in our example sentence. If we look for the Italian words in the list that are most similar to the Spanish words in the sentence we can begin to make educated guesses at various translations. By following this logic we can translate a few of the key words in the Spanish sentence, concluding that *gato* means *cat*, *pescados* might mean *fish*, and *muerde* could be some grammatical form of *bite*. From this we can say the sentence seems to be about a white cat biting a fish, which is not far from what the sentence translates to. Some of our choices may seem like more of a stretch than others. The similarity between *blanco* and *bianco* is much stronger than that between *pescados* and *pesci*. In fact, following this approach can cause

[1] Our apologies if the Spanish grammar is not perfect, the translations were done by machine and this is still an area needing more research.

[2] http://world.altavista.com/babelfish/

Italian	English
uno	one
due	two
piccolo	small
grande	big
bianco	white
rosso	red
orecchio	ear
occhio	eye
gatto	cat
asino	donkey
pesci	fish
latte	milk
burro	butter
morso	bite
ritrovamento	find

Table 1.1: Italian/English translations

problems. Consider this Spanish sentence, *Un burro tiene dos oidos.* Attempting to use the same similarity based method can cause us to make bad choices. The number two seems to be part of the sentence (from *dos* and *due*). However *oidos* could be translated into ears (using *orecchio*) or possibly eyes (using *occhio*). A major problem occurs when using similarity for examining the word *burro* in the Spanish sentence. We have an exact match for this word in our Italian list, the problem is it means butter. We are beginning to construct a sentence about the two ears (or eyes) of butter. This shows that while the similarities between Italian and Spanish can be helpful for translating to English, the process is difficult and can lead to unexpected errors.

This brings us to the purpose of this thesis, which is to examine word similarity in natural language and find some way to determine the useful information that it contains while filtering out the noise and errors. We want to find a way to discover what constitutes true similarity between words, and separate that from the similarities that can occur by chance. Most languages have a relatively small alphabet, restricted even more by patterns and regularities imposed on the words by the language, so it is possible that words may look very similar based on random occurrences. Thus, one of our main goals

is to model a system that can differentiate between similarity that occurs due to a relationship existing between the words from these coincidental matches.

We also have another important goal, and that is to have a system that is applicable to any language pair. To achieve this goal we shall use the techniques of machine learning to automatically create our model from a set of training data. Our hope is that if we have some examples that exhibit the similarity we are looking for, then we can automatically learn exactly what details about these word pairs makes them similar. If we can determine what kinds of transformations are likely between languages then it should help us to determine what pairs show similarity based on the trained criteria.

You may have noticed that the words in our example Spanish sentences and in the Italian/English list all represent very simple concepts. This has happened because the similarity we were using to aid us in translation comes from a very specific source: *cognates*. The study of cognates is the initial motivation for the entire system. More information on cognates can be found in the Chapter 3. For now it is enough to know that cognates are similar because of an evolutionary process. This process represents how languages have developed through history. The idea is that a pair of related languages (like Spanish and Italian) are both descendants of a single root language. This language was the starting point for all of the related languages, but over time the languages began to diverge. Fortunately such divergences often exhibit a great deal of uniformness throughout the evolved language. It is common for a vowel or consonant change to be consistent between the two languages. Table 1.2 shows this correspondence between several Romance languages using counting numbers.

From the table you can begin to see what sort of correspondences there are. English provides a counter-example since it is from a different language family. Obviously the languages from the same family share many common features, but you can also see how they are diverging. Different language prefer variations on vowels and consonant structure. There seems to be a correspondence developing between the tokens "c" and "q". Much is persevered between these languages as well. Letters like "d" and "n" occur in the same location

French	Italian	Spanish	Portuguese	English
un	uno	uno	um	one
deux	due	dos	dois	two
trois	tre	tres	tres	three
quatre	quattro	cuatro	quatro	four
cinq	cinque	cinco	cinco	five
six	sei	seis	seis	six
sept	sette	siete	sete	seven
huit	otto	ocho	oito	eight
neuf	nove	nueve	nove	nine
dix	dieci	diez	dez	ten

Table 1.2: Numbers in various Romance languages

consistently.

Cognates between languages that exist because of a shared history are called genetic cognates. The word "genetic" is used to help illustrate how such similarities form. They are created through a process that is in many ways a parallel of how DNA and other biological components have evolved over the course of life's history. Because of this we looked for inspiration in the field of bioinformatics. This field is appealing for many reasons. First of all, it is well rooted in mathematics and computer science theory. The algorithms that become popular in this field are in frequent use, so they are stable and well understood. One sub-field of bioinformatics that looks especially similar to natural language study is biological sequence analysis [8]. This field has become important lately because of the large scale DNA sequence projects that are being undertaken. These projects require the automation of sequence analysis in a way that is both efficient and accurate.

Some of the more popular methods are based on probabilistic theories, most notably, Hidden Markov Models. A Hidden Markov Model represents a stochastic process that is used to generate a series of observations. The model consists of states that emit the various observations (using a probability distribution) along with transition probabilities for moving between the different states of the model. It differs from a regular Markov Model in that the state sequence cannot be exactly determined by looking at the observation sequence. Instead it must be reconstructed by various algorithms that

examine the likelihood of any state sequence producing such an output.

In fact there is a new form of Hidden Markov Model that matches well with the needs of a word similarity measurement system. This model is called the *Pair Hidden Markov Model.* The idea behind this model is to utilize a pair of observation streams, instead on a single observation sequence. These streams are used to represent alignments between word pairs. The model consists of three states, each representing a different way of processing the two streams. They essentially give the choice of processing the streams together, in which case we are matching up the tokens of the words. We can also process each stream individually, using what are called insertions and deletions. The model can then be used to determine the probability of our sequence of observations. For example we could align the Spanish word *blanco* and the Italian word *bianco* in several possible ways. The two most intuitive are the following:

```
b l a n c o
b i a n c o

b l - a n c o
b - i a n c o
```

The algorithms of the Pair Hidden Markov Model provide a means to measure these alignments and compare them. We can use such alignments as a way to rank how likely two words are to be related. There are various methods to do this, each using a different but valid approach.

We also wanted a system that can be trained automatically, so that we could handle a variety of word similarity tasks. Pair Hidden Markov Models provide us with the methods we need to accomplish this. They use a modified form of the Baum-Welch algorithm, or forward-backward algorithm. This is a type of expectation maximization algorithm common to machine learning. The idea is that you can incrementally modify the parameters of your model to better fit your training data. Each iteration of the forward-backward algorithms creates a model that more accurately represents the training data. It is usually possible to run this algorithm with no knowledge of what the parameters should be, and still learn their values.

Chapter 2 discusses work done in related fields that involves either alignments, Hidden Markov Models, Pair Hidden Markov Models or some combination of them. Chapter 3 provides more details about the problem of word similarity, the tasks we will be examining, and any information that will be needed in order to construct a model to represent our domain. The next two chapters go over the necessary background material, with Chapter 4 concentrating on the mathematical theory and algorithms of Hidden Markov Models, while Chapter 5 gives the details of Pair Hidden Markov Models as well as the implementations and algorithms we will be using for our experiments. Those experiments and the data collected are discussed in Chapter 6, with the details of our tests listed in Appendix A.

Chapter 2

Related Work

Our technique involves the use of Hidden Markov Models, but more generally our main task is the alignment of the various parts of a word. As such, there is much that can be gained by examining other alignment techniques that exist in other fields. One of the most common and in many ways best documented alignment tasks comes from another sub-field of Natural Language Processing: *Machine Translation.* Just as we seek to find the best alignment of tokens within words, a component of machine translation involves a similar goal; the alignment of words within a sentence. This alignment is designed in such a way as to align words that are translations of each other, but looks at the problem of choosing alignments similar to how we look at choosing the alignments between parts of the individual words. This view considers alignments as representing a transformation from one form to another using a series of (for the most part) consistent rules and correspondences. In Machine Translation this first form is the text in one language which is transformed into the correct translation of the text in the second language. Word similarity takes a word in one language and aligns segments that correlate to parts of a word in another related language.

In the literature this transformation is often described using the analogy of the noisy channel. It works by imagining that we are working with only a single language, the *source language.* However when someone attempts to transmit something in the source language, either through writing, or speaking, or whatever method of transmission is most convenient, the language gets

corrupted. This corrupted language is actually our *target language*, but we consider it as a corrupted form of the source language. The task is now to recreate the original form from the corrupted output. The probabilities that arrive from such an examination are useful in alignment tasks in general.

Once computers became powerful enough, the noisy channel model could be learned automatically from existing corpora that was representative of translations between various languages. Some recent and widely used techniques for using such text were introduced by a team of researchers at IBM [4]. The major problem that needs to be solved in both word and token alignment is determining the parameters that will be used for the various transformations. Such correspondences are usually unknown beforehand, so they need to be trained by examining the data. In addition, it is unlikely that the corpora being trained on will have the alignments included. For Machine Translation it is adequate, for a starting point, to have two texts that are known to have the same content, but represented in different languages. Of course for many languages using such a limited approach will not give very good translations, but it is where the approaches for Machine Translation and word similarity measurement are most like each other. It is then possible to apply a machine learning technique, such as the Expectation Maximization algorithm, along with some (arbitrary) starting parameters to begin to learn the alignments from this parallel data.

Machine Translation works by applying an iterative deciphering methodology to the data. If you know (or at least have a reasonable guess for) a few alignments in your data, you can use those alignments as a starting point to find more alignments in your data. By combining this with statistical knowledge of the individual languages, like bigrams for example, it is possible to create a reasonable set of word alignments representing a possible translation. Knight [13] provides a simple, but detailed example of this process at work. He also provides an introduction to the structure of Statistical Machine Translation including the underlying mathematics and probabilities [14]. Machine Translation at its core deals with the combination of two models, a language model and a translation model. For our work we concentrate on the transla-

tion model, since the language model is mostly concerned with word ordering and correct grammar, while our approach provides constraints that make such corrections unnecessary.

An important point about Machine Translation is that it can benefit from the more detailed examination that word similarity evaluation can provide. This is possible because one of the more straightforward applications of a word similarity program is to recognize words between languages that evolved from the same root form. These words, called *cognates*, are more often than not translations of each other, since in evolving from the same word it is reasonable that they would preserve the meaning of the original. As such, word similarity measures can be used to bootstrap a Machine Translation program, providing a better start for the EM algorithms by augmenting (or providing) a translation dictionary to base initial alignments on [19]. Such a use is especially beneficial when no machine readable bilingual dictionaries exist. In addition because this is done as a preprocessing step, cognate information can be added to a variety of Machine Translation programs without modifying the original system.

An interesting experiment with the IBM model was done by Och and Ney [23]. They compared other techniques, such as Hidden Markov Models, to the standard progression of IBM models that were used in the original parameter estimation application [4]. In the IBM model only the first two levels (IBM-1 and IBM-2) can be calculated efficiently enough to allow an examination of all possible alignments. Hidden Markov Model alignment algorithms have the same property of efficient calculations, but with one important difference. Hidden Markov Models have a first order structure, meaning that an alignment position depends on the previous alignment position. The simpler IBM models use a zero order structure where all alignments are independent of each other. They place the complexity of the Hidden Markov Model somewhere between IBM models 2 and 3, making Hidden Markov Models the most complex construct that can still be exhaustively searched, giving exact values for all of the algorithms. The more complex IBM models (3+) need to use heuristics to create sub-sets which are then examined in their entirety. The authors experiment with using a Hidden Markov Model instead of IBM-2 when training

through the IBM progression. The IBM models are normally trained in order, with each one creating a starting point for the next. The Hidden Markov Model helps boost the performance of the overall system, suggesting that it is capable of producing good alignments despite the simplicity that allows for complete and exact calculations. These properties are some of the reasons that a Hidden Markov Model (and in turn a Pair Hidden Markov Model) seemed like a reasonable model to use for the task of word alignment.

The Hidden Markov Model used in the previous paper is based on the idea of using alignment probabilities dependent on relative alignment positions rather than on the absolute position of alignments in the sentence. This idea was previously developed by Vogel, Ney, and Tillman [28]. The authors' motivation was to overcome the independent word positioning assumption used in the IBM models of similar complexity (the ones where exact probability calculations are possible). Their point was that aligned words are not randomly distributed within sentences but instead form clusters. To capture this behavior they use a first order Hidden Markov Model to allow for dependence on a previous alignment. The results with the Hidden Markov Model were comparable with those of the IBM models, with indications that the Hidden Markov Model algorithms, with the smoother alignments that they can produce, would be better able to handle more complex correspondences that may exist between the two sentences.

Hidden Markov Models are a well known technique, but they have not been explored as a way to align the parts of a word. However, there have been many techniques proposed that can examine and align words for various purposes. The most common task is the recognition of cognates, which can be useful as a part of a larger system (such as one for Machine Translation), but also functions as a domain of investigation all on its own. This is especially true when determining the history of languages, an interesting and difficult problem, since records for many languages are few or even non-existent. For example there are languages that pre-date writing and thus have no lasting records once their use has faded. These languages can often be the source of many modern languages that are still in use today. As such, reconstructing

the root languages provides a means to link modern languages together, easing tasks such as translation.

Covington [6] created a program to align phonetic sequences and extended it to allow for alignments between more than two languages [7]. Each language supplies one word or string. He uses a discrete set of eight substitution scores, or match scores, representing how good (low score) or bad (high score) a substitution is by applying a set of comparison criteria. For example a substitution of identical consonants gets a score of 0, while the substitution of unrelated sequences has a score of 100. Two operations in his substitution set are not substitutions in the conventional sense. Since the substitutions are against gaps, they follow the form of insertions and deletions in our model. Covington adopted an affine gap penalty system, where the initial insertion or deletion costs more than subsequent ones. While the algorithm itself is useful, the metric is admittedly weak. The author states that is it just a stand-in for a more sophisticated system.

Kondrak created a program that uses multi-valued features to compute the similarity of phonetic sequences [16]. The program called ALINE represents phonetic sequences as feature vectors, where each feature represents some phonological idea, such as place of articulation. Each feature has some value within the continuous range of 0 to 1, allowing for flexibility and adaptability to different languages (many of which have various phoneme sets). This approach was shown to outperform Covington's method, based on the alignments in several different sets of cognates.

Since word alignment functions more or less (usually more) as a special case of edit distance, it is useful to examine another technique that has a very similar probabilistic flavor when compared to Hidden Markov Models. Ristad and Yianilos [26] created a stochastic model for determining edit distances that uses a Finite State Transducer. It was able to automatically learn string edit distances from a given corpus of examples. They use two methods of alignment, which they call Viterbi and "stochastic" edit distances. Not surprisingly their Viterbi is equivalent to the Viterbi algorithm for Hidden Markov Model, where only the best sequence is considered. The stochastic edit distance cor-

responds to the forward or overall probability, where all possible sequences are considered. Their model consists of a single state, and within that state all operations share the probability mass. Thus, substitutions, insertions, and deletions are all part of the same state. This approach is lacking in that it is memoryless, and hence each edit operation is preformed independently of all others unlike Hidden Markov Models where each operation depends on the one that came before it. Their experiment does suggest that learning based approaches can be quite effective. In some ways the comparison of their transducer model to Hidden Markov Models is similar to the way IBM models 1 and 2 were compared against Hidden Markov Models by Och and Ney [24].

A surprising experiment was preformed by Mann and Yarowsky [20]. They investigated the induction of translation lexicons using bridge languages. Their approach starts with a dictionary between two well studied languages, English and Spanish is one pair provided. They then use cognate pairs to induce a "bridge" between two strongly related languages, such as Spanish and Italian, and from this create a smaller translation dictionary between English and Italian. The related languages used in their experiments all come from the same language family and hence cognates should exhibit a great deal of surface similarity. They compared the performances of three different cognate similarity (or distance) measures; one based on the Levenshtein distance, one based on the stochastic transducers of Ristad and Yianilos [26], and the last comprised of a Hidden Markov Model. However their model is of a distinctly different design than the Pair Hidden Markov Model we are employing for the task of word similarity measurement. For example, the probability of the atomic edit operations sum to one for *each* character. This approach provides a different structure than the transducer where all edit operations share the probability of the model (sum to one). In our approach, described later, we separate the edit operations into distinct states and deal with the output of pairs from each state. The authors' choice of model does seem to be a poor one as it is out-preformed by the transducer model. This is unusual considering that both approaches are strongly grounded in probabilistic theory with, in my opinion, the Hidden Markov Model winning out in complexity and expressibility. Nev-

ertheless their Hidden Markov Model fails to even out-preform the Levenshtein distance, falling well short of that baseline. I believe this is in part due to the high similarity of the languages studied. Since many cognates between these languages are actually identical (or extremely similar) words, they would get the zero Levenshtein score. In more complex language pairs with better hidden and less well studied correspondences, it should be easy for a probabilistic learning method to preform better than a fixed scoring scheme. In addition the fact the similar, but in some ways weaker, learned model gets better results, suggests to me that not enough care was taken in the initial development of their Hidden Markov Model. The success Hidden Markov Models have had in other fields leaves me confident that our approach can work, if done with enough care.

Although our technique comes from the field of bioinformatics, we are not the first in the natural language community to attempt to use Pair Hidden Markov Models as a means to learn alignments. Clark [5] used them as the basis for a system that would learn morphology, using a model of stochastic string transductions. He adopts a rather novel output structure where the pair is either an identical pair, a single token from the "right" stream, or a single token from the "left" stream. His models were mostly concerned with the addition of suffixes to the ends of words. To do a true substitution of one token for another, each token would need to be output independently on each stream. He uses a mixture of different Pair Hidden Markov Models, one for each morphological class. A morphological class represents a single transformation rule in the language in question. For English, there would be one Pair Hidden Markov Model to represent adding an "s" to the end of a word, and another to add "es". This is due to the ambiguity that exist in natural language stems. This technique got reasonable accuracies when tested on tasks in English past tense, German plurals, and Arabic plurals. Since we want our model to work independent of languages, it is promising to see a general approach applied successfully.

Chapter 3

Word Similarity

Word similarity is, at its core, an alignment task. So our word similarity measurement system will essentially be an alignment system. For the system to function properly we require that it have some measure for determining how related two word segments are. Along with this we need a way to determine scores for each pair that can be used to rank a pair of words in order to compare them against each other. Finally since we want our system to be adaptable, we would like to have a method of automatically determining the parameters of our system so it can work on a variety of tasks in many different languages.

As we will show, a Hidden Markov Model will meet all the necessary criteria of our system. But before we begin to discuss how Hidden Markov Models work, it is important to better understand the task they have been recruited to preform, along with the constraints and assumptions that go along with that task.

3.1 Word Representation

The first problem we face is the various representations of words that are used when doing similarity alignment. There are many different alphabets in use in the world and our program must be able to handle any of them. Of course they must be put into some form that is understandable by a computer as well. In addition to this, it is possible to represent the same word in the same language in different ways. The simplest representation is *orthographic*. An orthographic representation of a word is simply the word written in its nat-

ural language, or at least in a representation that corresponds closely with the natural language. The second method in common usage is *phonetic transcription.* When using phonetics, the symbols represent the way the word would sound when spoken. Table 3.1 shows both an orthographic and a corresponding phonetic representation, from ARPAbet, a phonetic alphabet for American English that uses only ASCII characters. Phonetic transcription has the advantage that given a large enough sound alphabet, all languages can be represented in the same way. An example of such an alphabet is the International Phonetic Alphabet which has the goal of representing all of the sounds in human language. More specifically for the problem of cognate recognition and other common word similarity tasks, it is often easier to see correspondences between words by looking at the sounds than the written word. There are disadvantages to the phonetic approach as well. Most important, it is difficult to find data that has been transcribed phonetically, and programs to automatically transform words to phonetics are not yet fully developed. This is mostly due to the ambiguity that exist in the written form. For example, the English letters "ough" can produce different sounds. Consider the words "cough", "rough", "dough", and "through". Each one is pronounced differently, so each would need a different phonetic representation.

Orthographic	Phonetic
sage	s ey jh
raccoon	r ae k uw n
lotus	l ow dx ax s

Table 3.1: Examples of English orthographic and phonetic representations

Once we have settled on a representation, we need to break that representation up into several parts that can be used for alignment. These parts can be thought of as *tokens* and we use that terminology sometimes in our discussion. For the representations we have considered, two possible tokens sets would be the alphabet of the language, or a phonetic alphabet such as ARPAbet. It would also be possible to consider certain combination of elements of those sets as a single token, for example the English letter pair "ch".

3.2 Word Similarity Tasks

Once a representation has been chosen, we can split up each word into tokens based on that representation. Our next job is to take a pair of words and by examining their component tokens, decide if the words exhibit the similarity for which we are looking. The choice of exactly what kind of similarity we are searching for is dependent on the application being used. As an example that is used in our experiments we can consider recognition of *cognates*. Cognates can be defined in a few different ways, but essentially words are cognates they have the following properties:

1. They are similar in form or sound

2. They have the same meaning

There are many ways that cognates can be produced between languages. For discussion of these cognates we will use the language pair English and Russian [22]. Several examples for each type of cognate are shown in Table 3.2. The Russian words are displayed phonetically so that the correspondences are more obvious, using the tables presented with the examples [22]. Russian uses a different alphabet than English, so it exhibits little orthographic similarity. English is part of the Germanic language family, while Russian is a member of the Slavic family. However, both languages are part of the larger Indo-European set. The first source of cognates, sometimes called *genetic* cognates, are words that come from the same root word. This root word, sometimes called a *proto-form* represents a word in some distant, yet common, language for which there may be no record. Genetic cognates are usually words that represent very simple concepts that would have been important to people throughout history. A second source for cognates is *direct borrowing*, where a word from one language is transplanted in the other. These are usually words that are very culture specific, but also contains newer technological words that have only recently come into being, in only one of the two languages. Another occurrence similar to borrowing is *transliteration* that occurs when a proper noun in borrowed. These are usually transformed based on spelling

and are much easier to recognize that normal cognates. Transliteration is normally thought of as a task separate from other word alignments, although the learning approach we are using suggests it may be adaptable to that task as well. An example of a transliteration system can be found in the work of Knight and Graehl [15]. The final source we will discuss is another form of borrowing, *borrowing from a third language*. These occur when both of the languages being studied have had contact with another culture that uses a third language. Often both of the languages of interest will have borrowed the same cultural terms from the third language, with the borrowed words being transformed to fit with the patterns of their new home.

	English	Russian (phonetics)
Genetic cognates	mama	mama
	two	dva
	no	nyet
Direct borrowing	vodka	vodka
	hooligan	xuliygan
	hacker	xakyer
Transliteration	Russia	Rossiyya
	Picasso	Piykasso
Borrowing from a third language	fiesta	fiyyesta
	karma	karma
	bandit	bandiyt

Table 3.2: Examples of cognates between English and Russian

The main obstacle for a cognate recognition system is the ability to differentiate between words that are actually related in one of the ways mentioned above, from words that look similar simply by chance. It is possible to put even stronger restrictions on the system, perhaps wanting to find only the genetic cognates. Other types of similarities exist as well, such as recognizing confusable words within a single language, or determining the correct spelling from a misspelled word. The hope is that our system, if given the right type of training data, will be able to adapt to any word similarity task.

3.3 Constraints on the Model

Before we begin developing the word similarity model and its algorithms, we can simplify the procedure a great deal by using some domain knowledge to restrict the task of word alignment. In doing so we can make our model simpler and our algorithms more efficient.

The first simplification we make comes from an important difference between the structure of words and that of a sentence. When translating sentences, determining word order is a crucial step. Different languages have greatly divergent grammatical structure and hence require different word orderings in order to produce the same concept. This is one of the reasons why a language model is so important in Machine Translation. Words, however, tend to have the opposite behavior. Since we are looking for words that are similar in some way, it is usually safe to assume that the basic ordering of tokens remains the same between languages. This doesn't mean every token has a corresponding one in the other language, but instead that word transformation comes from three basic operations. The first is called *substitution.* This operation represents a transformation from one token to another. It may be as simple as the same sound being represented by different letters in different languages, or a complex change that has evolved over millennia. The other operations are reflections of each other, they are *insertion* and *deletion.* Insertion represents an addition of more tokens as we change from the first word in a pair to the second. A deletion represents the loss of tokens during the same change, but it could be looked at as a gain in the other direction. In fact if we switch the order of our word pair then the insertions and deletions would switch with it. This is why they can in many ways be thought of as the same operation. The ability to handle different token alphabets for each language prevents us from considering them as a single operation, but we shall see later that this symmetry will help us simplify our model, considerably reducing the number of parameters.

More information can be derived from our view of how two words relate to each other. The evolution of a word is normally a long process, with each

step representing a minor change in form or pronunciation. As such, the evolution process is relatively consistent keeping the orderings of tokens similar between languages. Thus we can assume that the segments, or tokens, of the evolving words will not get mixed around but instead, for the most part, only change form. This keeps the "grammar" of a word fairly consistent between related languages. Thus we can allow our system to only consider alignments in which there are *no crossing links.* This assumption is often used in word similarity programs, and allow the algorithms that search through the space of alignments to work faster. However, the number of alignments is still very high.

We will also be assuming a one to one correspondence between alignments. Prefixes and suffixes that are added to a word will be aligned to gaps. If by some chance a token in one word becomes many tokens in another word we would model that change by using one of the many tokens as an anchor point, the point with which the single token is aligned. The other tokens will be taken care of through the use of insertions or deletions. Although this is not ideal, it does simplify the problems caused by having to deal with token fertilities, and removes the need to create some other method to deal with many to one correspondences. If there is a many to one correspondence that is consistent between languages, it would be beneficial to change the word representation so that the many tokens can be considered as a single token. For example a group of tokens in an orthographic representation may form a single sound, but if the word was written phonetically then that single sound would correspond to a single token.

Chapter 4

Hidden Markov Models

This chapter will present the basis for our word similarity alignment and ranking system. We use the concept of a Markov Model (or more specifically a Hidden Markov Model) along with the dynamic programming algorithms synonymous with it. Hidden Markov Model have been applied successfully to other problems in Natural Language Processing, most notable the field of Speech Recognition, where they have become one of the most effective and widely implemented techniques. Additional background on Hidden Markov Models and their uses in Speech Recognition can be found in [25] and [10].

The first few sections deal with some background information that will be useful for the understanding of Hidden Markov Models. The first section contains a description of Markov Models which, as the name suggests, are the mathematical construct on which Hidden Markov Models are based. The second section describes dynamic programming, a simple but powerful programming technique that allows us to work with very large search spaces quite efficiently. The remaining sections go into more details of the mechanics behind Hidden Markov Models with an emphasis on the algorithms and approaches that have proven successful in the past, and show promise for similar success when used with our word similarity model.

4.1 Markov Models

Markov Models were first introduced by Andrei Markov in 1913. They are stochastic processes that retain only the minimum amount of prior knowledge,

or memory. A Markov process can be defined as a stochastic process for which the future depends only on the present, not the past. Essentially we need only to look at the most recent event in order to predict the next event, all other events are unimportant. In more exact terms the process must have the following property.

Given a set of random variables $X_1, X_2, \ldots, X_n$, where each variable is from some finite alphabet $S = \{s_1, s_2, \ldots, s_m\}$ then the *Markov Property* states that

$$\forall t \leq n : P(X_t|X_{t-1}, \ldots, X_1) = P(X_t|X_{t-1}).$$

A stochastic process with the Markov Property is often called a *Markov Chain*. The chain can be specified with a matrix containing transition probabilities for each state in the model. It is often desirable to include a set of initial state probabilities to avoid the need to begin in a unique and predetermined start state. Consider the following simple example of a Markov Model shown in Figure 4.1. The model will produce a binary string of any length. It has two states, state 0 which outputs a "0" and state 1 which (oddly enough) outputs a "1". One possible definition of such a model would be:

$$\mathbf{\Pi} = \begin{pmatrix} 0.5 & 0.5 \end{pmatrix}$$

$$\mathbf{A} = \begin{pmatrix} 0.6 & 0.4 \\ 0.3 & 0.7 \end{pmatrix}$$

In this example, and in a general Markov Model, we know what state we are in simply by looking at the output. So we can calculate the probability of a given observation sequence directly, by using the probabilities given in the transition matrix for the model. For example the observation sequence "01101" would have probability:

$$\begin{aligned} P(01101) &= \pi_0 a_{01} a_{11} a_{10} a_{01} \\ &= (0.5)(0.4)(0.7)(0.3)(0.4) \\ &= 0.0168 \end{aligned}$$

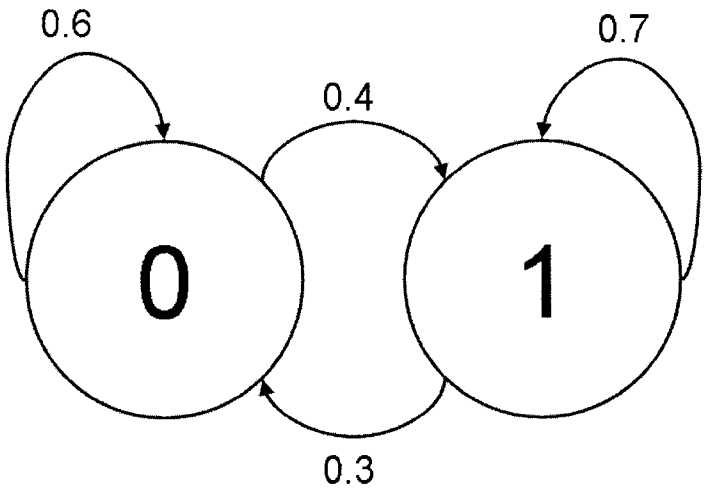

Figure 4.1: A Markov Model to create binary strings

4.2 Hidden Markov Models

Although some problems can be solved by using a visible Markov Model there are many that are too complex to be described within such a restrictive framework. We can add an extra layer of randomness by allow each state to produce various output based on some (possibly unknown) distribution. This second stochastic process is not directly observable, hence the "hidden", but instead must be inferred from the produced observation sequence. Essentially the transitions through the states of the model can no longer be determined directly from the observations. Instead the observations only offer clues from which, using the right set of algorithms, we can attempt to determine the underlying process that created them.

To accomplish this change we need to add more probabilistic information to our model. There are two popular (and equivalent) ways to add this information to a model. The first is to increase the number of transitions, where each transition now includes an emission symbol. The choice of emission symbol and next state are done at the same time, so the emission depends on both the current state and the next state. The second way (the one our word similarity model uses) is to include a new set of emission probabilities at each state,

keeping the transitions between each state the same as in a visible Markov Model. In this way the emissions depend only on the current state. As a simple example we can modify the binary string model as shown in Figure 4.2. Now it will have a 0-preferring state, and a 1-preferring state with these emission probabilities for "0" and "1":

$$\mathbf{E_0} = \begin{pmatrix} 0.8 & 0.2 \end{pmatrix}$$

$$\mathbf{E_1} = \begin{pmatrix} 0.3 & 0.7 \end{pmatrix}$$

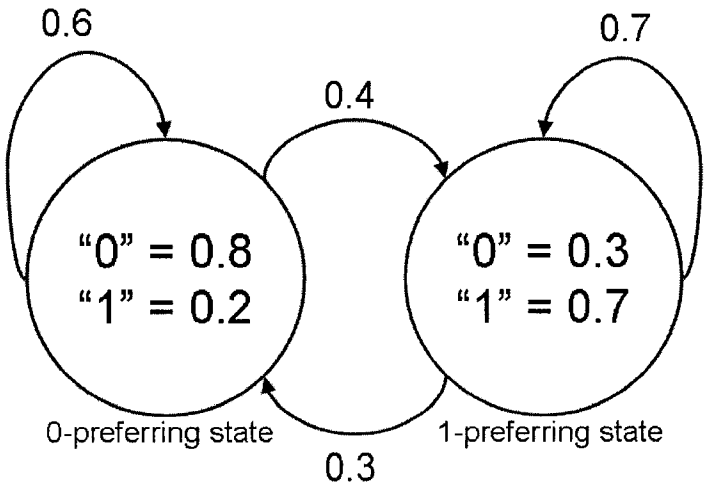

Figure 4.2: A Hidden Markov Model to create binary strings

For such a model we can not determine the state sequence just by looking at an observation. This is because a single string can be produced in a number of different ways. For example, the string "000" could be produced by staying only in the 0-preferring state, or by staying in the 1-preferring state (or any other of the $2^3 = 8$ combination of states). The only difference is the probability of either of those state sequence producing that string. Obviously the string is more likely to be produced by staying in the 0-preferring state, but for more complex models, how do we decide which state sequence is most probable, and how do we determine the probability of a given string as we did for the previous Markov Model representation?

Before we can begin to examine the properties of Hidden Markov Models it will be helpful to more formally define what a Hidden Markov Model is. The following is the definition used by Manning and Schutze [21] modified slightly to represent state emission Hidden Markov Models. A Hidden Markov Model is defined as a five-tuple (S, Σ, Π, A, E) where each symbol is defined as in Table 4.1. An example of a generic Hidden Markov Model state is shown in Figure 4.3.

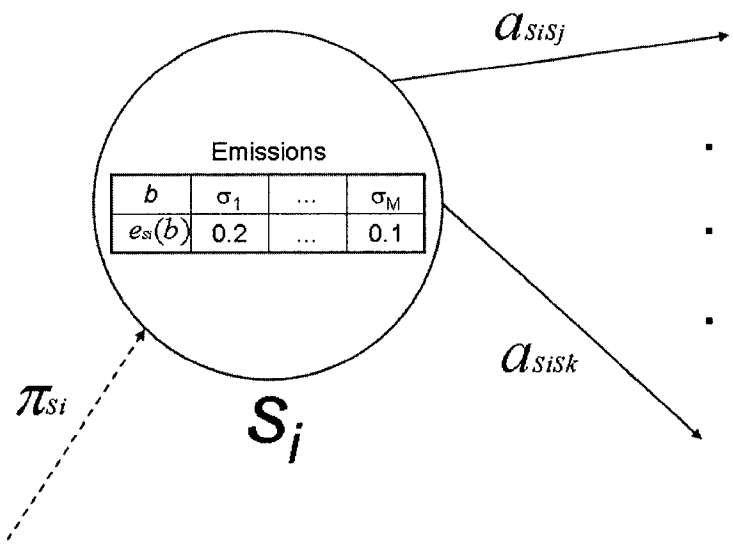

Figure 4.3: The various components of an HMM

Set of (hidden) states	$S = \{s_1, \ldots, s_N\}$
Output (observation) alphabet	$\Sigma = \{\sigma_1, \ldots, \sigma_M\}$
Initial state probabilities	$\Pi = \{\pi_i\}, \quad i \in S$
State transition probabilities	$A = \{a_{ij}\}, \quad i, j \in S$
State emission probabilities	$E = \{e_k(b)\}, \quad k \in S, b \in \Sigma$
State sequence	$\mathbf{X} = (X_1, \ldots, X_T), \quad X_t : S \mapsto 1, \ldots, N$
Output (observation) sequence	$\mathbf{O} = (o_1, \ldots, o_T), \quad o_t \in \Sigma$

Table 4.1: Notation for Hidden Markov Models

Since the set of states and output alphabet often remain constant in many of the later applications and algorithms a more compact notation for a model is often used. A model μ can be defined simply as a triplet, $\mu = (A, E, \Pi)$.

The various probabilities in Table 4.1 are defined as such:

$$\begin{aligned} \pi_i &= P(s_1 = i) \\ a_{ij} &= P(s_t = j | s_{t-1} = i) \\ e_k(b) &= P(o_t = b | s_t = k), \quad b \in \Sigma, k \in S \end{aligned}$$

As was stated before, we can no longer determine the state sequence (and hence the probability) of an observation simply from the observation itself. In fact when dealing with a Hidden Markov Model there are three questions that need to be answered if the model is to be of any use at all.

1. How do we determine the probability of a given observation sequence $\mathbf{O} = (o_1, \ldots, o_T)$ given the model $\mu = (A, E, \Pi)$?

2. How do we determine the "most likely" state sequence $\mathbf{X} = (X_1, \ldots, X_T)$ given an observation sequence $\mathbf{O}$ and a model μ?

3. How do we determine the "best" model μ given a set of observations $\mathbf{O}$, or how do we adjust the parameters of the model μ to maximize $P(\mu|\mathbf{O})$?

4.2.1 Generating Observations with a Hidden Markov Model

There is a simple way to view a Hidden Markov Model that helps to explain how the various probabilities in the model work. Because of its probabilistic nature, the Hidden Markov Model can be thought of as a generator of observation sequences. Although direct generation is rarely of practical value, except perhaps debugging training algorithms, this is how we view the functioning of our model and is the basis for all of the algorithms derived from it. Figure 4.4 shows the generation process using the binary string producing HMM as an example.

Any possible observation sequence $\mathbf{O} = (o_1, \ldots, o_T)$ can be generated by the following procedure [25]:

1. Choose an initial state X_1 based on the initial state distribution Π.

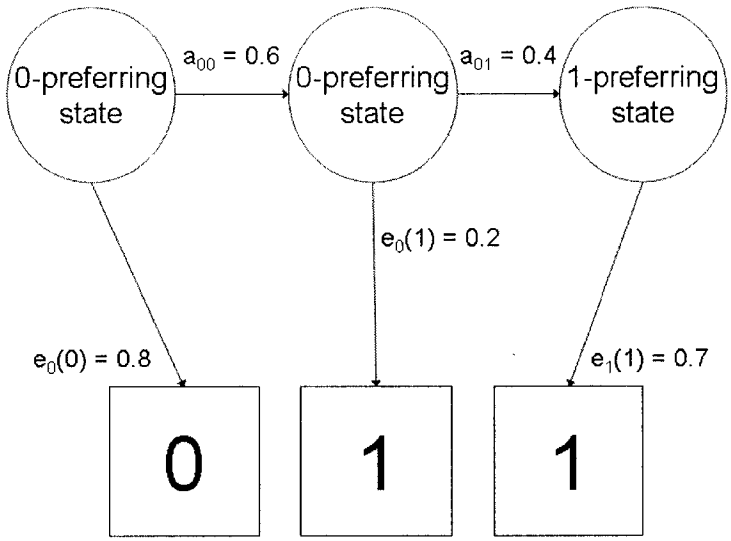

Figure 4.4: Generating observations using an HMM

2. Set $t = 1$.

3. Choose $o_t = b$ according to the emission probabilities $e_{X_t}(b)$.

4. Transit to a new state X_{t+1} based on the transition probabilities $a_{X_t X_{t+1}}$.

5. Set $t = t + 1$, return to step 3 if $t < T$, otherwise end the generation procedure.

For the above procedure we assumed that we could end in any state, and opted to use the number of observations to control when we stopped. Another option is to add transitions to a silent end state from all the other states. This lets the Hidden Markov Model decide when to stop generating an observation sequence. The higher the probability of going to the end state, the shorter the observation sequences will be. The procedure would then become:

1. Choose an initial state X_1 based on the initial state distribution Π, if $X_1 = \text{END}$ end the generation procedure, otherwise proceed to step 2.

2. Set $t = 1$.

3. Choose $o_t = b$ according to the emission probabilities $e_{X_t}(b)$.

4. Transit to a new state X_{t+1} based on the transition probabilities $a_{X_t X_{t+1}}$.

5. If X_{t+1} = END end the generation procedure, otherwise set $t = t + 1$ and return to step 3.

This generation structure shall be examined with our Pair Hidden Markov Model to see if it can give us any useful information about similar words and how they end.

4.3 Dynamic Programming

We will find that it is not enough simply to answer the three fundamental questions for Hidden Markov Models. In addition, we must be able to answer them in a way that can be efficiently and practically implemented. One technique that will prove invaluable to this end was developed by Bellman in 1957 [2]. It works by keeping track of optimal solutions to subproblems of the main problem, usually in some table or matrix. These subproblem solutions can then be used to incrementally calculate the optimal solution to the main problem, without having to re-examine previous results.

To illustrate its effectiveness we will briefly examine a simple dynamic programming algorithm that is also used to align words, the *minimum edit distance* algorithm. Minimum edit distance is defined as the minimum number of edit operations that are required to transform a source string into a target string. Pseudo-code for the algorithm is shown in Figure 4.5; it is a slightly modified version of the code presented in Speech and Language Processing [11]. The operations are usually *substitution*: exchanging one token for another; *insertion*: adding a new token to the string; and *deletion*: removing a token from the string. This algorithm is helpful because the operations it uses will be used to define the states in our word similarity model, and it closely resembles the solution to the first fundamental question of Hidden Markov Models. One of the first things we need to do is assign a cost for each of the operations. The simplest of these is the *Levenshtein* distance [11], which assigns each operation a cost of 1. This will give a measure of the min-

```
INPUT: Two words
  source, target

n = length(source)
m = length(target)
distance[0][0] = 0
for i from 1 to n
  distance[i,0] = distance[i-1,0] + del_cost(source[i])
for j from 1 to m
  distance[0,j] = distance[0,j-1] + insert_cost(target[j])
for i from 1 to n
  for j from 1 to m
    distance[i,j] =
      min(distance[i-1,j-1] + sub_cost(source[i], target[j]),
        distance[i-1,j] + del_cost(source[i]),
        distance[i,j-1] + insert_cost(target[j]))

OUTPUT: The minimum edit distance.
  distance[n,m]
```

Figure 4.5: Pseudo code for the minimum edit distance algorithm

imum number of operations needed to achieve the transformation. With the dynamic programming algorithm it is now possible to develop solutions to the fundamental questions in an efficient way.

4.4 Probability of an Observation Sequence

We now can turn our attention to the first fundamental question; how do we determine the probability of an observation sequence? This will be especially important because the algorithms presented here will form an integral part for answering the third question, which is the most important for the problem we are dealing with.

With a visible Markov Model we knew the path through the model that was used to generate the observations. For a Hidden Markov Model we know only the observations, there are several possible state sequence that could produce then. For the remainder of this discussion it will be helpful to assume that our model is *ergodic*, meaning that there are transitions from every state to

every other state. What we need to consider is the total probability for every possible path through the model that would produce the observation sequence that we are examining. The mathematical development will closely follow that presented by Rabiner and Juang [25].

We want to find the probability of the output sequence $\mathbf{O} = (o_1, \ldots, o_T)$ given a model μ. First consider a single fixed state sequence

$$\mathbf{X} = (X_1, \ldots, X_T)$$

The probability for the observation sequence given this state sequence is

$$P(\mathbf{O}|\mathbf{X}, \mu) = \prod_{t=1}^{T} P(o_t|X_t, \mu).$$

The assumption of statistical independence of observations gives us

$$P(\mathbf{O}|\mathbf{X}, \mu) = e_{X_1}(o_1) \cdots e_{X_T}(o_T).$$

The probability for the fixed state sequence is

$$P(\mathbf{X}|\mu) = \pi_{X_1} a_{X_1 X_2} \cdots a_{X_{T-1} X_T}.$$

We can now calculate the joint probability of the observation sequence and the state sequence occurring together given the model, which is the product of the previous two terms.

$$P(\mathbf{O}, \mathbf{X}|\mu) = P(\mathbf{O}|\mathbf{X}, \mu) P(\mathbf{X}|\mu).$$

The probability of $\mathbf{O}$ given the model μ can be calculated by summing the joint probability over all of the possible state sequences $\mathbf{X}$. In an ergodic model there are N^T such sequences.

$$\begin{aligned} P(\mathbf{O}|\mu) &= \sum_{\text{all } \mathbf{X}} P(\mathbf{O}|\mathbf{X}, \mu) P(\mathbf{X}|\mu) \\ &= \sum_{\text{all } \mathbf{X}} \pi_{X_1} e_{X_1}(o_1) a_{X_1 X_2} e_{X_2}(o_2) \cdots a_{X_{T-1} X_T} e_{X_T}(o_T). \end{aligned}$$

The problem with calculating this value directly is that it is horribly inefficient. For the general model being discussed this gives a calculation of order $O(2TN^T)$. More precisely, the calculation requires $(2T-1)N^T$ multiplications

and $N^T - 1$ additions [25]. Even for very small values of N and T this can mean an unacceptable number of calculations. However there is a way around this obstacle by using dynamic programming techniques. This technique is know as the *forward algorithm* (or forward procedure).

4.4.1 Forward Algorithm

The forward algorithm works like any other dynamic programming technique: it keeps track of optimal solutions to sub-problems and then uses them to calculate the solution to the main problem. In this case the main problem is the probability of an observation sequence and the sub-problems are probabilities for sub-sequences of the observation. These sub-sequences are described by something known as the *forward variable*, defined as

$$\alpha_i(t) = P(o_1 \cdots o_t, X_t = i|\mu).$$

This represents the probability of seeing the partial observation sequence $o_1 \cdots o_t$ and being in state i at time t. We can calculate all of the forward variables recursively using the steps outlined in Table 4.2.

1. Initialization
$$\alpha_i(1) = \pi_i e_i(o_1), \qquad 1 \leq i \leq N.$$

2. Induction
$$\alpha_j(t) = e_j(o_t) \sum_{i=1}^{N} \alpha_i(t-1)a_{ij}, \qquad \begin{matrix} 1 < t \leq T \\ 1 \leq j \leq N \end{matrix}.$$

3. Termination
$$P(\mathbf{O}|\mu) = \sum_{i=1}^{N} \alpha_i(T).$$

Table 4.2: Forward Algorithm for Hidden Markov Models

The forward algorithm is considerable more efficient then naively calculating the probability directly. It has an order of only $O(N^2T)$ calculations as opposed to the $O(2TN^T)$ required by the direct calculation. More exactly it takes $N(N+1)(T-1)+N$ multiplications and $N(N-1)(T-1)$ additions [25].

4.4.2 Backward Algorithm

The forward algorithm represents only one way of examining the probability of an observation sequence, although it is the most intuitive. There is however another method of determining the probability that will prove very useful when attempting to determine the parameters of a Hidden Markov Model (fundamental question 3). This is the *backward algorithm* or backward procedure. It works in much the same way as the forward algorithm only it starts from the end of the observation sequence and builds up the complete probability from the end to the beginning. We shall need to define a *backward variable* similarly to how we defined the forward variable as

$$\beta_i(t) = P(o_{t+1} \cdots o_T | X_t = i, \mu).$$

We can now define the backward algorithm recursively to solve for all of the backward variables as shown in Table 4.3. Just as with the forward algorithm, the backward algorithm requires order $O(N^2T)$ calculations.

1. Initialization
$$\beta_i(T) = 1, \qquad 1 \leq i \leq N.$$

2. Induction
$$\beta_i(t) = \sum_{j=1}^{N} \beta_j(t+1) a_{ij} e_j(o_{t+1}), \qquad \begin{matrix} T-1 \geq t \geq 1 \\ 1 \leq i \leq N \end{matrix}.$$

3. Termination
$$P(\mathbf{O}|\mu) = \sum_{i=1}^{N} \pi_i \beta_i(1).$$

Table 4.3: Backward Algorithm for Hidden Markov Models

4.5 Finding the Best State Sequence

The first obstacle in finding the best state sequence, is determining exactly what is meant by the "best". Unlike finding the probability of the observation sequence where there is a single correct answer, the answer to this question can

change depending on what we use for our optimality criterion. There are several different possible definitions of "best". For example, we could choose the states individually at each step to maximize the expected number of states that will be guessed correctly. However, such an approach may yield a very unlikely state sequence [11]. We will instead adopt the most commonly used definition of "best": maximize the overall probability of the entire path through the Hidden Markov Model, finding the single state sequence (or possibly a set of state sequences) that has the highest probability given the observations and the model. Thus we are to compute

$$\arg\max_{\mathbf{X}} P(\mathbf{X}|\mathbf{O}, \mu)$$

which is equivalent to determining for a fixed observation sequence

$$\arg\max_{\mathbf{X}} P(\mathbf{X}, \mathbf{O}|\mu).$$

4.5.1 Viterbi Algorithm

An efficient algorithm exists to calculate the most probable state sequence for a fixed set of observations. It again uses the ideas of dynamic programming starting with the following definition

$$\delta_i(t) = \max_{X_1 \cdots X_{t-1}} P(X_1 \cdots X_{t-1}, o_1 \cdots o_t, X_t = i|\mu).$$

This variable represents the probability of the single best (highest probability) path through a given Hidden Markov Model, μ, which accounts for the first t observations and ends in state i. It is possible to calculate this variable recursively to get the probability of the single best path that accounts for all of the observations. If we want the state sequence as well, then we only need to keep track of the path through the model by storing a backtrack variable $\psi_j(t)$ at each step. The entire procedure is shown in Table 4.4. Note the similarities between the Viterbi and forward calculations. Except for the backtracking, the only real difference between the two algorithms is that the forward algorithm calculates a sum, while the Viterbi considers only the maximum at any step. This distinction will become more important when we discuss various methods of ranking word pairs using our word similarity model in later chapters.

1. Initialization

$$\delta_i(1) = \pi_i e_i(o_1), \qquad 1 \leq i \leq N$$
$$\psi_i(1) = 0.$$

2. Induction

$$\delta_j(t) = e_j(o_t) \max_{1 \leq i \leq N} \delta_i(t-1)a_{ij}, \qquad \begin{matrix} 2 \leq t \leq T \\ 1 \leq j \leq N \end{matrix}$$

$$\psi_j(t) = \arg \max_{1 \leq i \leq N} \delta_i(t-1)a_{ij}, \qquad \begin{matrix} 2 \leq t \leq T \\ 1 \leq j \leq N \end{matrix}.$$

3. Termination

$$P(X^*) = \max_{1 \leq i \leq N} \delta_i(T).$$
$$X_T^* = \arg \max_{1 \leq i \leq N} \delta_i(T)$$

4. Path backtracking

$$X_t^* = \psi_{X_{t+1}^*}(t+1), \qquad T-1 \geq t \geq 1.$$

Table 4.4: The Viterbi Algorithm for Hidden Markov Models

4.6 Finding the Optimal Model Parameters

The third problem will prove to be both the most difficult to solve and the most important to our word similarity model. We need a way to find an optimum set of model parameters $\mu = (A, E, \Pi)$ for a given observation series. As such we are trying to maximize $P(\mathbf{O}|\mu)$. Unfortunately, there currently exists no method that can find the model parameters μ that will maximize that probability. The best we can accomplish is to use iterative hill-climbing techniques in order to find a local maximum. The hope is that this local maximum will be sufficient to adequately represent our model.

4.6.1 Expectation Maximization Algorithm

To solve the optimal parameter problem we will be using a type of Expectation Maximization algorithm, call the *Baum-Welch* algorithm or *forward-backward* algorithm. The forward-backward algorithm works by utilizing a recursive procedure. If we have a set of parameters we can use those parameters to

calculate the probability of the observation sequence. If we have an aligned observation sequence we can look at what transitions and emissions are used more often and adjust the probabilities of the model to better fit the data. This gives us a circular optimization method, which we can break by using random starting parameters for our model. There are other ways to choose the initial parameters of the model, some of which may be better than others, but for a general discussion of how the forward-backward algorithm works it is enough to know that the initial parameters are chosen.

Before we can describe the algorithm's details we need to define a new variable $\xi_t(i,j)$ which represents the probability of being in state i at time t then moving to state j at time $t+1$, more formally

$$\xi_t(i,j) = P(X_t = i, X_{t+1} = j|\mathbf{O}, \mu)$$

We can use the forward and backward variables to help represent the various probabilities that we are trying to calculate as is shown in Figure 4.6 adapted from Foundations of Statistical Natural Language Processing [21]. Thus our original probability can be re-written as

$$\begin{aligned}
\xi_t(i,j) &= P(X_t = i, X_{t+1} = j|\mathbf{O}, \mu) \\
&= \frac{P(X_t = i, X_{t-1} = j, \mathbf{O}|\mu)}{P(\mathbf{O}|\mu)} \\
&= \frac{\alpha_i(t)a_{ij}e_j(o_{t+1})\beta_j(t+1)}{\sum_{m=1}^{N}\alpha_m(t)\beta_m(t)} \\
&= \frac{\alpha_i(t)a_{ij}e_j(o_{t+1})\beta_j(t+1)}{\sum_{m=1}^{N}\sum_{n=1}^{N}\alpha_m(t)a_{mn}e_n(o_{t+1})\beta_n(t+1)}
\end{aligned}$$

We shall now define the probability of being in state i at time t as

$$\begin{aligned}
\gamma_i(t) &= P(X_t = i|\mathbf{O}, \mu) \\
&= \frac{P(X_t = i, \mathbf{O}|\mu)}{P(\mathbf{O}|\mu)} \\
&= \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N}\alpha_j(t)\beta_j(t)}.
\end{aligned}$$

Note also that

$$\gamma_i(t) = \sum_{j=1}^{N}\xi_t(i,j).$$

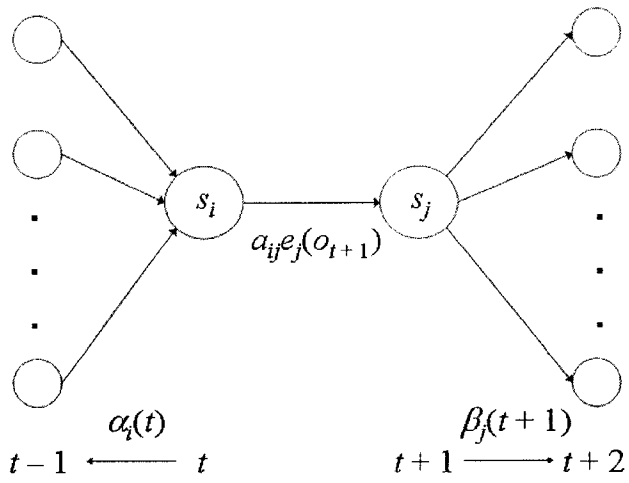

Figure 4.6: The probabilities associated with a transition

Now if we sum over time we get expectations, or counts, that a state is visited. This is equivalent to the number of transitions from that state if we exclude time index T. We can also get counts for the number of transitions from any state to any other state in a similar matter using ξ instead of γ. With these values in hand we can re-calculate our model parameters

$$\begin{aligned}
\bar{\pi}_i &= \text{expected number of time in state } i \text{ at time } t = 1 \\
&= \gamma_i(1) \\
\bar{a}_{ij} &= \frac{\text{expected number of transitions from state } i \text{ to state } j}{\text{expected number of transitions from state } i} \\
&= \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_i(t)} \\
\bar{e}_j(k) &= \frac{\text{expected number of times in state } j \text{ observing symbol } k}{\text{expected number of times in state } j} \\
&= \frac{\sum_{t:o_t=k, 1\leq t\leq T} \gamma_j(t)}{\sum_{t=1}^{T} \gamma_j(t)}.
\end{aligned}$$

Hence, from our initial model parameters $\mu = (A, E, \Pi)$ we can determine new parameters $\bar{\mu} = (\bar{A}, \bar{E}, \bar{\Pi})$. It has also been proved by Baum et al. [1] that

$$P(\mathbf{O}|\bar{\mu}) \geq P(\mathbf{O}|\mu)$$

where equality will occur if the initial model μ represents a critical point, such

as a local maximum, which is any point where all partial derivatives are zero (or some partial derivatives do not exist). Thus through repeated iteration of the forward-backward algorithm, we can get increasingly better models with respect to our training data.

Chapter 5

Pair Hidden Markov Models

Hidden Markov Models have proven to be powerful tools for many tasks within the field of Computer Science. The question remains whether or not they could prove useful for the task of word similarity measurement and ranking. A good indicator of possible success comes from the field of bioinformatics.

One of the more intangible aspects of a Hidden Markov Model is the choice of the model itself. While many algorithms exist to train the parameters of the model so that the model better describes its data, there is no formulaic way to create the model. Instead we must normally rely on our best judgement, and a great deal of trial and error for all but the simplest problems. Fortunately for us, there exists a model that is already in use for a task quite similar to our own. We need only to refine and improve this model to have an excellent starting point from which we can train the parameters for word similarity alignment.

A new type of Hidden Markov Model was developed by Durbin et al. [8] that uses two observation streams in parallel. This model was dubbed the "Pair Hidden Markov Model" and has been used successfully to determine alignments of biological sequences. Since biological sequences are usually represented by a series of alphanumeric characters, it seems possible that similar algorithms could be employed to align the tokens of words. Of course there are differences between word similarity and biological sequence analysis, such as the shorter length of sequences for natural language words, the differences in surrounding context, and the importance of having accurate physical align-

ments between words (which is as important as a good ranking system). These differences shall be part of the work of developing the bioinformatics techniques to a new domain. The Pair Hidden Markov Model is also appealing because all of the major Hidden Markov Model algorithms were developed to go along with the new paired structure, and they also employ the dynamic programming methodology that makes the Hidden Markov Model computationally appealing. This suggests that those algorithms could also be created for a word similarity model. This is an important point considering that exceptionally large corpora are often used in Natural Language Processing, making efficient algorithms a necessity.

What follows in this chapter is the model created by a careful consideration of how to best represent word alignment in a Pair Hidden Markov Model. All of the core Hidden Markov Model algorithms are reinvented to fit within the framework of the model and still retain all of the efficiencies of the originals. We begin with a more detailed discussion of the original Pair Hidden Markov Model and how it differs from the regular Hidden Markov Model. We then discuss the assumptions and weakness of this model and develop a model more suited to the alignment and ranking of words in natural languages. The standard algorithms are presented again with an emphasis on how they differ from the regular Hidden Markov Model algorithms.

5.1 The Biological Model

Pair Hidden Markov Models were developed to align biological sequences but with the same necessities that we have for word similarity alignment. This creation was motivated by the need to differentiate between the similarities between sequences that exist because of a scientific or historical relationship from sequences whose similarity is based solely on chance. The basic idea behind the model is that it emits a pairwise alignment instead of a single observation sequence. This allows a pair of words to be examined as a single entity instead of two separate streams of data. It also adds an extra dimension to the search space, but all of the regular Hidden Markov Model algorithms

can be adapted to work with it.

The model has three states, each corresponding to one of the basic edit operations: substitution, insertion, and deletion. Each state has its own emission probabilities representing the likelihood of producing a pairwise alignment of the type described by the state. The word pairs are represented by x and y, where n and m are the lengths of x and y respectively. Durbin et al. use i and j to represent indexes into the token set for x and y in that order. To represent emission probabilities the model uses the symbol $p_{x_i y_j}$ for emission of pairs from the substitution state. The emissions from the insertion and deletion states are designated by q_{y_j} and q_{x_i}.

Durbin et al. then decided on three transition parameters. The first, δ, represents the probability of going from the substitution state to either the insertion or deletion states. The next, ϵ, represents staying in the insertion or deletion state. This was done to provide facilities for affine gap penalties. Of course this property can be removed by tying these two probabilities together, forcing them to be equal.

The last parameter comes from a reformulation of the model. In order for the model to provide a probability distribution over all the possible sequences, they add a silent begin and a silent end state to the model. The authors chose to tie the probabilities of the start state to those of the substitution state. Thus, the probability of starting in the substitution state is the same as being in the substitution state and staying there, while the probability of starting in an insertion or deletion state equals that of going from the substitution state to the given state. This choice is reasonable and keeps the number of parameters from increasing unnecessarily. They then add a new parameter, τ, that models the probability of ending the sequence, going from any state to the end state. We do some experiments to see how useful such an end state model actually is. When doing alignments the lengths are known ahead of time making such a transition unnecessary except as a way to slightly modify scores when the alignment is done. Now each state has only one other transition, and the value of this transition is determined from the Markov Model property that all transitions sum to one.

5.1.1 Weaknesses in the Biological Model

The main weaknesses of the biological model come from the various simplifying assumptions that are made throughout its development. The core of these simplifications comes from the fact that the sequences being aligned are long, often in the range of dozens of tokens. This allows for many simplifications that (although they can decrease the overall "correctness" of the alignment measure) do not have a large enough impact on the overall calculation to cause any real problems. For us, word lengths are considerable smaller than biological sequence lengths. Usually words are less than 10 tokens long, and depending on the way the words are represented this can be further reduced. For example a phonetic representation will have fewer tokens since the set of tokens is larger than the original alphabet and thus more expressive. Consider how many English sounds are represented by multiple letters (like sh, or th). These would each be represented by a unique token in a phonetic representation, giving us a much smaller number of calculations with which to distinguish good alignments from bad alignments. This means our methods will be less robust to errors that were made on the side of simplification. Words can be as short as a single token, so if we use a simplified version of the model and it introduces errors to facilitate that simplification, we do not have a large number of subsequent calculations to "smooth" those errors out.

Most of the assumptions seem simple on the surface, but can permeate throughout entire algorithms greatly changing their structure. Some of the more simple problems were simplifications to the underlying model. The biological model assumes that an insertion followed by a deletion is the same as a match. This would cause problems when working with natural language because it would say strange things about the alignments created. Covington [7] provides an example of this using the Italian "due" and the Spanish "dos", both of which mean "two". The following alignment is correct,

d u e -
d o - s

If we allowed two successive gaps to be the same, a substitution we would get the alignment

d u e
d o s

which would provide evidence of a connection existing between the Italian "e" and the Spanish "s". Since this is incorrect we do not want such a situation occurring. If we followed the biological model, we would allow and possibly learn many correspondences that should not exist between our languages. Since the alignments produced by our algorithms are just as important as the numerical rankings, we need to keep these two cases separate. These problems can be solved by adjusting our model to more accurately represent the kinds of alignments we would expect between two similar words. The similarity model is discussed in more detail in the following section.

The original biological model also assumes that the probability for the transition to the end state is the same no matter what state we are currently in. If we try to train with this assumption we run into problems. We would need to ignore the actual values we are leaning for the end transition in one state, and instead replace it with the end transition probability learned from the other state. Doing this can cause problems during training, where we are not increasing the overall probability of the data. To overcome this we have split τ into two separate values, τ_M for the match state, and τ_{XY} for the insertion and deletion states. This preserves the symmetry of our model while allowing it to better express how word alignments end. It is often the case that alignments between cognates are more likely to end in a gap, and with this change our system can model this (or similar) properties.

Another problem that is not touched on by the biological model is the rather common problem of word length. Since we are calculating a chain of probabilities, longer words will have a longer chain, which will result in a lower probability. It is difficult to fix these probabilities within the algorithms without invalidating them, so instead we opted to correct for these lengths after the initial calculation by use of a scaling factor. The effectiveness of this approach can be seen in Chapter 6.

5.2 The Word Similarity Model

Our model was developed keeping the basic structure of the biological sequence model intact. As such we start with three main states. A "match" state (represented by state "M") that represents the alignment of two tokens, one from each word. An "insertion" state (represented by "Y") that allows a token in the second word to be aligned against nothing (also called a gap). Finally a "deletion" state (represented by "X") that lets a token in the first word to be aligned to a gap.

A match, or substitution, represents tokens that are being examined for how similar they are. As an example consider the alignment of cognates. Vowels are (usually) more likely to be replaced by other vowels, so the emission probability of two vowels from the match state should be higher than for the substitution of a vowel and a consonant. An insertion or deletion represents a gain or loss of a token from one word to the next in a pair. Some languages add or remove prefixes or suffixes, or there may be some tokens that get transformed into multiple tokens, or a group of tokens that get condensed into a single token. All of these events can be represented by insertions and deletions. Figure 5.1 shows how the Pair Hidden Markov Model can be used to generate pairs of words, with various alignments.

There are many similarities that exist between states "X" and "Y" and as such there are some symmetries in the model with respect to states "X" and "Y". However these states would only truly be the same if the alphabets of both our source and target languages were identical. Nevertheless we opted to maintain the symmetries that exist with respect to the transition probabilities. This is because in general word similarity is more dependent on the values of the substitution, insertion, and deletion costs. The transitions tend to be consistent regardless of language pair and as such, we can save computation time and model complexity while losing very little information about the language pairs. However the necessity of different alphabets for each language requires that the states remain distinct, with various emission probabilities from the insertion and deletion states. The model is shown in Figure 5.2.

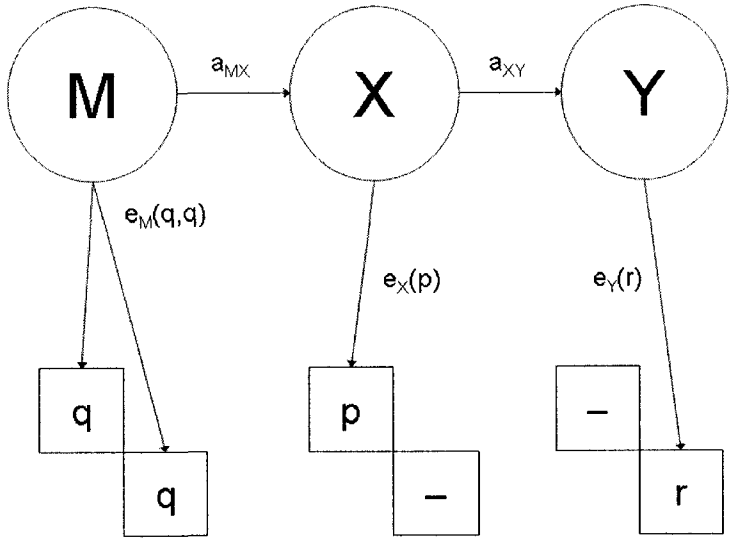

Figure 5.1: Generating an alignment using a PHMM

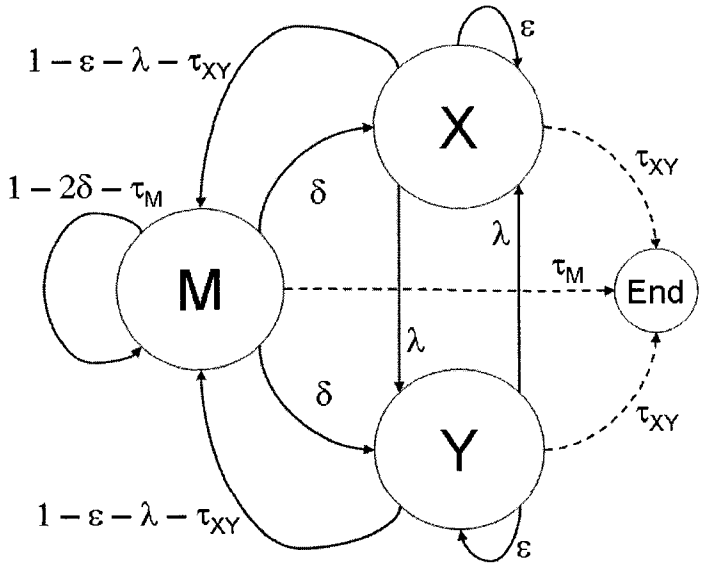

Figure 5.2: The word alignment Pair Hidden Markov Model

5.3 Word Similarity Model Algorithms

Each of the standard Hidden Markov Model algorithms can be created for the word similarity model. This was accomplished by following the basic ideas of the algorithms presented in Chapter 4, but including the ideas and structure of the Pair Hidden Markov Model. What follows are the details of the algorithms as they relate to the task of word similarity measurement. In our case, all of our algorithms deal with the domain of possible alignments between a pair of words. While the method that each algorithm employs to calculate a score for the pair is different, they all must look (at least to some extent) at all of the possible alignments available between the words. Even with the constraints of our model this is still a substantial number of possibilities. Thus it is important to note that each algorithm retains the properties of dynamic programming and as such accomplishes its search through the space of all allowed alignments quite efficiently.

In order to employ a dynamic programming technique the task must have the property that at any point in the calculation, if we have found the "optimal solution" then that solution will stay optimal regardless of what the next step in the solution will be. The key to maintaining the truth of this property are the assumptions that our alignments will contain no crossing links and that each token will only be aligned to one other token. We are finding an optimal path through our Pair Hidden Markov Model, and any sub-path must also be optimal. If there were to be a better sub-path we would use that path instead. Here we use probabilities to measure how good a path is, but the underlying principles remain the same.

5.3.1 Viterbi Algorithm

The purpose of the Viterbi algorithm is to find the best sequence of states, emissions, and transitions for a given observation sequence. The "best" sequence is defined as the sequence that has the highest overall probability. For our domain the observation sequence is the pair of words to be aligned, and the state sequence represents a possible alignment of those two words. Hence

we are trying to find the alignment between those two words that gives the highest probability out of all of the possible alignments.

This can be accomplished by using a dynamic programming algorithm, like the general Viterbi algorithm in Chapter 4. Table 5.1 is the pseudo-code adapted to our problem. In the pseudo-code we use $\bullet$ to represent an action performed for all states M, X, and Y. For the Viterbi algorithm, and all of our word similarity algorithms, the input is a pair of words (the observation sequence). The output would then be a "score" for that pair of words, determined at the termination of the algorithm.

1. Initialization

$$v^M(0,0) = 1 - 2\delta - \tau_M, v^X(0,0) = v^Y(0,0) = \delta.$$

$$\text{All } v^\bullet(i,-1), v^\bullet(-1,j) = 0.$$

2. Induction: for $0 \leq i \leq n, 0 \leq j \leq m$ except $(0,0)$

$$\begin{aligned}
v^M(i,j) &= p_{x_i y_j} \max \left\{ \begin{array}{l} (1-2\delta-\tau_M)v^M(i-1,j-1) \\ (1-\epsilon-\lambda-\tau_{XY})v^X(i-1,j-1) \\ (1-\epsilon-\lambda-\tau_{XY})v^Y(i-1,j-1) \end{array} \right\}, \\
v^X(i,j) &= q_{x_i} \max \left\{ \begin{array}{l} \delta v^M(i-1,j) \\ \epsilon v^X(i-1,j) \\ \lambda v^Y(i-1,j) \end{array} \right\}, \\
v^Y(i,j) &= q_{y_j} \max \left\{ \begin{array}{l} \delta v^M(i,j-1) \\ \epsilon v^Y(i,j-1) \\ \lambda v^X(i,j-1) \end{array} \right\}.
\end{aligned}$$

3. Termination

$$P(X^*) = \max(\tau_M v^M(n,m), \tau_{XY} v^X(n,m), \tau_{XY} v^Y(n,m)).$$

Table 5.1: Viterbi algorithm for Pair Hidden Markov Models

5.3.2 Forward and Backward Algorithms

The problem with the Viterbi algorithm is that it only looks at the most probable alignment between the two words. Some words may have only a single best alignment, while others may have several good alignments, each a

slight variation of the others. While the former may occur simply by chance, it is unlikely that a pair of words that have many high probability alignments could occur just by chance. What the forward algorithm in Table 5.2 will do is examine how similar two words are by looking at all of the possible alignments between them. This should give us a better idea of whether our word pair exhibits true similarity or if an alignment exists simply by chance.

To accomplish this we keep track of every path though our Pair Hidden Markov Model that can produce an alignment for a given word pair. By summing all of these alignments together a better picture of similarity may be obtained. A pair with only one good alignment will now receive a much lower ranking then a pair that has several high probability alignments. Hopefully this will result in a better indication of how similar a pair really is.

Just as with regular Hidden Markov Models, we can do the same calculations as the forward algorithm starting from the end instead of the beginning. The corresponding backward algorithm is given in Table 5.3.

1. Initialization

$$f^M(0,0) = 1 - 2\delta - \tau_M, f^X(0,0) = f^Y(0,0) = \delta.$$

$$\text{All } f^\bullet(i,-1), f^\bullet(-1,j) = 0.$$

2. Induction: for $0 \leq i \leq n, 0 \leq j \leq m$ except $(0,0)$

$$\begin{aligned} f^M(i,j) &= p_{x_i y_j}[(1-2\delta-\tau_M)f^M(i-1,j-1) \\ &\quad + (1-\epsilon-\lambda-\tau_{XY})(f^X(i-1,j-1)+f^Y(i-1,j-1))], \\ f^X(i,j) &= q_{x_i}[\delta f^M(i-1,j)+\epsilon f^X(i-1,j)+\lambda f^Y(i-1,j)], \\ f^Y(i,j) &= q_{y_j}[\delta f^M(i,j-1)+\epsilon f^Y(i,j-1)+\lambda f^X(i,j-1)]. \end{aligned}$$

3. Termination

$$P(\mathbf{O}|\mu) = \tau_M f^M(n,m) + \tau_{XY}(f^X(n,m)+f^Y(n,m)).$$

Table 5.2: Forward Algorithm for Pair Hidden Markov Models

1. Initialization

$$b^M(n,m) = \tau_M, b^X(n,m) = b^Y(n,m) = \tau_{XY}.$$

$$\text{All } b^\bullet(i,m+1), b^\bullet(n+1,j) = 0.$$

2. Induction: for $n \geq i \geq 0, m \geq j \geq 0$ except (n,m)

$$\begin{aligned}
b^M(i,j) &= (1-2\delta-\tau_M)p_{x_{i+1}y_{j+1}}b^M(i+1,j+1) \\
&\quad + \delta(q_{x_{i+1}}b^X(i+1,j) + qy_{j+1}b^Y(i,j+1)), \\
b^X(i,j) &= (1-\epsilon-\lambda-\tau_{XY})p_{x_{i+1}y_{j+1}}b^M(i+1,j+1) \\
&\quad + \epsilon q_{x_{i+1}}b^X(i+1,j) + \lambda q_{y_{j+1}}b^Y(i,j+1), \\
b^Y(i,j) &= (1-\epsilon-\lambda-\tau_{XY})p_{x_{i+1}y_{j+1}}b^M(i+1,j+1) \\
&\quad + \epsilon q_{y_{j+1}}b^Y(i,j+1) + \lambda q_{x_{i+1}}b^X(i+1,j).
\end{aligned}$$

3. Termination

$$P(\mathbf{O}|\mu) = (1-2\delta-\tau_M)b^M(0,0) + \delta(b^X(0,0) + b^Y(0,0)).$$

Table 5.3: Backward Algorithm for Pair Hidden Markov Models

5.3.3 Log Odds Algorithm

One of the biggest problems in recognizing similarity is always distinguishing which words really descended from the same source and which words only seem to be related, often due to randomness or some other factor. One method that could be successful is to create another Pair Hidden Markov Model that can determine how likely a pair of words is to occur randomly. We could then compare the probability of our alignments from the similarity model, to the probability of the words occurring by chance in the two languages. If the probability of them being properly aligned is high and the chance of them existing independently is low, we would give them a higher score than if they have a high probability alignment, but also have a high probability of being in the languages without any common root existing between them.

This property is captured in the log odds algorithm [8]. The probability of the similarity model is normalized by the probability from the random model, to give an overall score to the pair.

The Random Model

The random model represents how likely the pair is to occur together with no underlying relationship. In the sense of an alignment this would be an alignment without any matches (i.e. only insertions and deletions). Thus the random model would only need insertion and deletion states. To keep things simpler one word can be completely generated before the next word. The probabilities would be the same as long as the transitions between states had the same value. Since this is what we want, there is a single new parameter η to represent transitions through our random model.

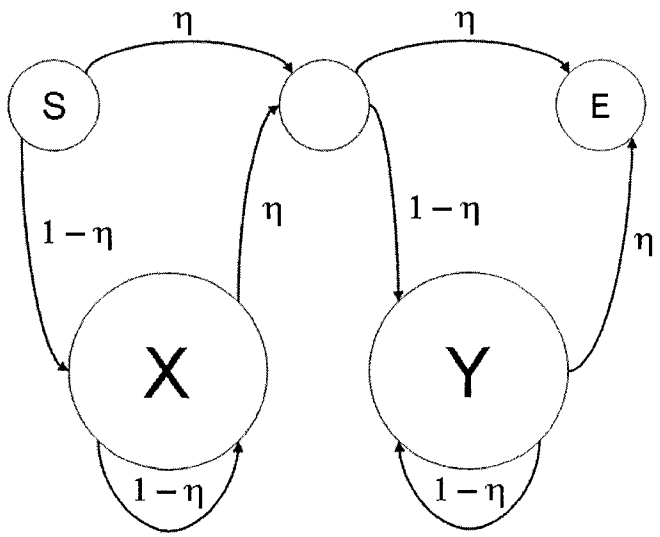

Figure 5.3: The random Pair Hidden Markov Model

Deriving the Log Odds Algorithm

To calculate the scores using the log odds model, we can use an algorithm that takes the same form as the other dynamic programming algorithms we are familiar with. To do this we need to know the probability of our observations given the random model R. The nature of the random model allows only one path for any word pair x and y, we shall use r_{x_i} and r_{y_j} to represent emissions from the corresponding states of the random model. The probability of the

path through the model is

$$\begin{aligned} P(x,y|R) &= \eta(1-\eta)^n \prod_{i=1}^{n} r_{x_i} \eta(1-\eta)^m \prod_{j=1}^{m} r_{y_j} \\ &= \eta^2(1-\eta)^{n+m} \prod_{i=1}^{n} r_{x_i} \prod_{j=1}^{m} r_{y_j}. \end{aligned}$$

The next step is to split up the terms of the two equations, the probability from the word similarity model and the probability from the random model, where the similarity model is being normalized by the random model (i.e. divided). We can compute the terms in an additive model with log odds emission scores and log odds transition scores. These two scores can be combined into a single term using the following equations.

$$\begin{aligned} s(a,b) &= \log \frac{p_{ab}}{r_a r_b} + \log \frac{1-2\delta-\tau_M}{(1-\eta)^2} \\ d(a) &= -\log \frac{q_a \delta(1-\epsilon-\lambda-\tau_{XY})}{r_a(1-\eta)(1-2\delta-\tau_M)} \\ e(a) &= -\log \frac{q_a \epsilon}{r_a(1-\eta)} \\ f(a) &= -\log \frac{q_a \lambda}{r_a(1-\eta)} \\ c &= \log \frac{1-2\delta-\tau_M}{1-\epsilon-\lambda-\tau_{XY}} + \log(\tau_{XY}). \end{aligned}$$

A few points need to be made about these equations. First of all since s is meant to always be used to represent a substitution regardless of what state we were in previously it reflects the assumption that the previous state was also a match state. To compensate for this d has a built in correction to represent going from an insertion or deletion back to the match state. This then requires a final correction c if we end up finishing in an insertion or deletion state. All of this means that although the intermediary steps may not have the correct values the final solution is correct. The log odds algorithm is presented in Table 5.4.

Also it is possible to simplify the above equations if you assume that the states in the random model have the same emission probabilities as the insertion and deletion states in the word similarity model, $r_a = q_a \ \forall a$. This simplified version is the one presented by Durbin et al. [8]. We have also modified the algorithms so that they can be used without this assumption since for

some applications the original version may be correct but it often represents an over-simplification. We have conducted experiments with the different model views to see the effects of this assumption in practice.

1. Initialization

$$V^M(0,0) = -2\log(\eta), V^X(0,0) = V^Y(0,0) = -\infty.$$

2. Induction: for $0 \leq i \leq n, 0 \leq j \leq m$ except $(0,0)$.

$$\begin{aligned} V^M(i,j) &= s(x_i, y_j) + \max\left\{ \begin{array}{l} V^M(i-1,j-1) \\ V^X(i-1,j-1) \\ V^Y(i-1,j-1) \end{array} \right\}, \\ V^X(i,j) &= \max\left\{ \begin{array}{l} V^M(i-1,j) - d(x_i) \\ V^X(i-1,j) - e(x_i) \\ V^Y(i-1,j) - f(x_i) \end{array} \right\}, \\ V^Y(i,j) &= \max\left\{ \begin{array}{l} V^M(i,j-1) - d(y_j) \\ V^Y(i,j-1) - e(y_j) \\ V^X(i,j-1) - f(y_j) \end{array} \right\}. \end{aligned}$$

3. Termination

$$P = \max(V^M(n,m) + \log(\tau_M), V^X(n,m) + c, V^Y(n,m) + c).$$

Table 5.4: Log Odds Algorithm for Pair Hidden Markov Models

Forward Log Odds

We have also created another variation on the regular log odds algorithm. Normally the log odds algorithm is a combination of the Viterbi algorithm with the random model. We have switched from the Viterbi algorithm to the forward algorithm to determine how effective such an approach is when combined with the random model. We use the same variations for the forward log odds algorithm that were used with the regular log odds algorithm. One version makes the assumption that insertions and deletions have the same probabilities in the similarity model as in the random model. The other uses EM algorithms to learn the insertion and deletion probabilities for the similarity model from the training data.

5.3.4 Expectation Maximization Algorithms

All of the previous algorithms require a large set of probabilities to be used. They need the probabilities for state emissions (substitution, insertion, and deletion probabilities), as well as probabilities for transitions between states. The problem is that it is difficult to determine these probabilities using only domain-specific knowledge. Most languages have not been studied in great detail, and those that have are usually not examined in a way that lends itself to a transformation into probabilities. For example is it well known that a vowel is likely to be transformed into another vowel, or that in closely related languages that share (at least part of) an alphabet matches between identical letters occur often. What is not known (at least in general) is exactly how likely it is for one vowel to be substituted for another, or how often identical segments are preserved between languages. Since the Hidden Markov Model has strict mathematical requirements, we need a way to satisfy these requirements using the data that is given to us. Fortunately, Hidden Markov Models already have a method in place that can do just that, the Expectation Maximization algorithms. The main difference between our algorithms and the regular Hidden Markov methods is that we are searching with an extra dimension, to allow us to consider different positions within the two output streams.

For these algorithms we only need to assume some starting probabilities. Uniform and random probabilities are common starting points. Then we check to see how well our training data fits in with our starting probabilities, and adjust the model to increase the overall probability of the data. We then repeat this process getting better and better models until we reach some stopping criteria. Since we are dealing with continuous values, convergence is unlikely to occur, so we need to stop after a certain number of iteration or when the changes that occur are sufficiently small.

Viterbi

The Viterbi version of the algorithm uses the initial probabilities to determine which alignments would be best and then after creating these alignments counts how often each match and gap occurs. It then uses this data to recalculate all of the probabilities and the process can repeat itself. In general, the Viterbi version of the EM algorithm is less powerful, but it remains to be seen how effective it would be for the problem of word similarity.

It is easy to implement a Viterbi Expectation Maximization algorithm, since we only need to align word pairs and then count the different alignments in the data. The advantage of this approach is that it only considers a single alignment. Some tasks, like alignment of cognates, usually have only one alignment that is correct. Although a single alignment says less about how similar words are it could be better for training. If only one alignment is correct then we only want to consider it when determining our parameters. Of course, our training data is not pre-aligned, otherwise a single iteration of Viterbi training would yield our model parameters. This makes this method very sensitive to the initial parameters that are chosen, and could potentially be very easily trapped in an incorrect local maximum.

Forward-Backward

The forward-backward calculation takes much more information into account. It uses partial counts for every possible match or gap weighted by the probability of reaching that alignment starting from both the beginning and end of the word pairs (forward and backward respectfully). While more complicated than the Viterbi algorithm, it is more used in practice and is usually more effective. It has the advantageous property of being more robust and can better deal with noise and errors that may exist in the training data.

The forward-backward algorithm can be used almost as shown in Chapter 4, with a few minor changes and simplifications. First of all we need to search through a 2-dimensional space of possible alignments, over several different

word pairs. The maximum likelihood estimators are:

$$a_{kl} = \frac{A_{kl}}{\sum_{l'} A_{kl'}} \text{ and } e_k(b) = \frac{E_k(b)}{\sum_{b'} E_k(b')}.$$

Where A_{kl} represents the number of transitions from state k to l, and $E_k(b)$ is the number of emissions of b from state k. We now sum over each pair, where h represents the index of the pair we are using.

We can use the following simplification in the equation for $E_k(b)$.

$$\begin{aligned}
\gamma_i(t) &= \sum_{j=1}^{N} \xi_t(i,j) \\
&= \sum_{j=1}^{N} \frac{\alpha_i(t) a_{ij} e_j(o_{t+1}) \beta_j(t+1)}{P(\mathbf{O}|\mu)} \\
&= \frac{1}{P(\mathbf{O}|\mu)} \alpha_i(t) [\sum_{j=1}^{N} a_{ij} e_j(o_{t+1}) \beta_j(t+1)] \\
&= \frac{1}{P(\mathbf{O}|\mu)} \alpha_i(t) \beta_i(t)
\end{aligned}$$

We now have the much simpler equation

$$E_k(b) = \sum_h \frac{1}{P(\mathbf{O}|\mu)} \sum_{i|x_i^h \in b} \sum_{j|y_j^h \in b} f_k^h(i,j) b_k^h(i,j).$$

The equations for the various states have slightly different forms. For example, insertions only need to match y_j with the emission b, since y_j is emitted against a gap.

For transitions this is the general formula for transitions ending in the substitution state. When the transition ends in an insertion or deletion state we only change the index for one of the pairs, that is only one of i or j change. We also use the emission probability for a letter from one word against a gap.

$$A_{kl} = \sum_h \frac{1}{P(\mathbf{O}|\mu)} \sum_i \sum_j f_k^h(i,j) a_{kl} e_l(x_{i+1}^h, y_{j+1}^h) b_l^h(i+1, j+1).$$

By utilizing all of the above equations, with variations for each state, it is possible to construct a forward-backward algorithm that is able to learn all of the parameters of our model. Such a program only requires that we provide it with training data representative of the similarity we want to model. The data does not need to be aligned in any way, although the initial conditions

can help it to converge faster if we have some domain knowledge we can use as a starting point.

Chapter 6

Experiments

The experiments reported in this chapter are mostly concerned with the examination of our Pair Hidden Markov Model as a means to automatically model word similarity. To this end, we examined the effect of various training techniques and variations on several word alignment tasks. Our model is trained using the Expectation Maximization algorithm, in either its Viterbi or forward-backward form. Once the model is trained we have more choices for how to do the alignments and rankings. The Viterbi algorithm was implemented in logarithm form to increase computational speed. We also examined the forward calculation, since it takes into account all alignments, and several variants of the log odds formulation, as they take into account how likely words are to occur at random, and have a normalization step as part of their structure. This chapter begins with the development experiments and what they have taught us about our model. Each section includes formal tests, our focus being on the recognition and ranking of cognates. We also examine the task of identifying confusable drug names.

6.1 Cognate Recognition: Development and Results

An excellent task to measure the abilities of a word similarity system is the recognition of cognates. Thus we have chosen to examine how well our model can learn to rank pairs of words based on how likely they are to be cognates. We also chose this task to be the one on which our system was developed.

We decided on this domain because cognate recognition is a common word similarity task, and it gives us the opportunity to examine both the ranking capabilities of our system and the corresponding alignments it creates.

The task of our system is to examine pairs of words between two languages, some of which are cognates and some of which are not. The data is labeled accordingly, but our program ignores those labels. The labels are used by another program to determine how accurate our rankings are. Our system aligns the two words to optimally fit our model, giving each alignment a score that represents how likely the words are to be cognates. These scores are relative to each other, not to any universal scale. The pairs can then be ordered and if everything worked well, true cognates will be at the top of the list. To measure how well we have ranked cognates we use an evaluation metric developed for Information Retrieval, designed specifically to evaluate rankings. The technique is called *11-point interpolated average precision* [21]. It involves calculating precision at various levels of recall. For 11-point these levels are 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100%. Precision is calculated by using the highest precision that occurs anywhere after a given point of recall is reached. Figure 6.1 shows the interpolation procedure.

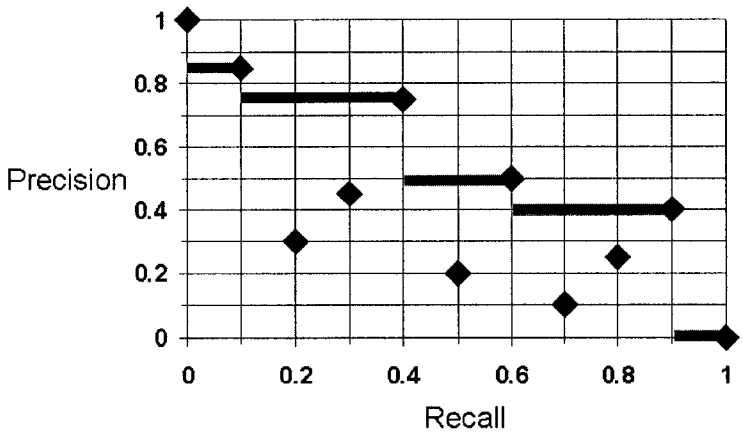

Figure 6.1: An example interpolated precision-recall curve

This domain is challenging because word similarity is not a perfect indicator of whether two words are cognates, since there are other ways that words can be similar. Borrowing is one example, random chance is another possibility. Of course the log odds algorithm is designed to help detect random occurrences and lower their rank accordingly. It is also possible that two words are cognates and yet exhibit no surface similarity. This can be especially true when basing decisions on orthographic similarity, but sometimes such a comparison is the only one available.

6.1.1 Cognate Data

The data for training our cognate recognition model comes from a set of lexicostatistical experiments preformed by Dyen et al. [9]. Lexicostatistics examines the percentage of cognates that exist between two languages. The higher the percentage the more closely related the languages. The words under examination are from the list of 200 basic meanings developed by Morris Swadesh in 1952 [27]. They represent words that are essential to human communication, and should exist within all languages. The data we obtained uses 95 speech varieties all of which are Indoeuropean. The data also has all of the cognate decisions made by Dyen.

The data consists of several small sets of words for each basic meaning. All members of the same set have been judged to be either *cognate* with each other, or *doubtfully cognate* with each other. Doubtfully cognate represents words that show similarity, but it is too difficult to determine whether that similarity arises from being cognates, borrowing, or chance. There are also connections between sets, but these are only vaguely specified. One set would be considered related to another if the words in the first set had a relationship with *at least* one member of the other set. Unfortunately there is no indication given in the data of exactly which member that is.

The data set is not perfect. In addition to the set relationship problem, there is often more than one form given for an individual entry in a set. The forms are considered cognates if one of the forms is a cognate to one of the forms of another member of the set. When using automatic means to extract

the data it can be difficult to capture all of these relationships without error. Luckily, there is a large number of pairs, so any errors that slip into the data should not effect the overall results. A few false pairs out of over two hundred thousand should not have an effect on training. The pairs themselves were created by exhaustively matching up every word in a set. We chose not to try to find matches between the sets, since such a relationship means only that some members of the two sets are related, with no indication which members those are. There are more than enough examples if we only consider the sets themselves so there is no need to deal with the difficulties of determining the cognates that exist between sets. Since we did not want the ordering of the pairs to have an effect on the training data, we also put the reverse of each pair into the data. This creates symmetry in our final parameters, which is good considering we have a single alphabet and are trying to model one language family.

Despite the various languages used in the data, all of the words are represented in English letters. This means the forms are described well enough to be examined, but not well enough to be used for phonetic analysis. The data that we are testing with has the same representation. This brings us to one problem that occurs when using this data to train: the difficulty of separating training and testing data. Normally it is a simple matter to separate out the test examples from the training data. For cognate data this is difficult to do completely. Word forms between languages can be similar, if not identical. This means that if you are testing on a specific language, removing all of the words of that language from the data will not necessarily mean that your test examples do not occur in the data. This occurs because there are several languages that exhibit a large amount of similarity. As an example imagine you were testing on Spanish and Polish, and removed those languages from the training data. There exists other languages, Italian and Russian for example, that would represent most of the same tokens and correspondences as the removed pair. In this case Italian could substitute for Spanish, and Russian for Polish. Instead of trying to remove the testing examples, and possibly presenting results that may be incorrect, we instead allow the testing

and training data to have some overlap during development. We shall still remove languages for formal testing, but we want to bring attention to the effect language overlap has on the training data. For now the sheer size of the training data when compared to the development sets should help prevent the parameters from being tuned to any particular language. Instead we hope to extract general correspondences between Indoeuropean languages and test the succuss those of correspondences in recognizing cognates between very specific language pairs.

Development Data

For development we have chosen to use two language pairs. These are Italian and Serbo-Croatian (abbreviated IK), as well as Polish and Russian (PR). We chose these sets because they represent opposite levels of similarity. Polish and Russian are very similar, possessing a high number of cognates, while Italian and Serbo-Croatian are much more distinct, with fewer correspondences. The percentage of cognates within the data is important, as it provides a simple baseline from which to compare the success of our algorithms. If our selection process were random, then we would expect to get roughly these percentages for our recognition precision (on average). For the Italian and Serbo-Croatian data, the percentage of cognates is 25.3%. The Polish and Russian data set has a much higher cognate density at 73.5%. These would be the expected precision values for any random approach.

Test Data

The data used in testing represents pairing of words from 5 different languages. Each language provides 200 of 1000 total words. The list for each language contains 200 meanings (one word for each meaning), and words are paired if they share the same definition. The languages are Albanian, English, French, German, and Latin. This data comes from the word lists given by Kessler [12]. Although these word lists did not come from the cognate data used for training, there is still some overlap, since the same cognates occur frequently between languages. Four of the languages: Albanian, English, French, and

German, were in the training data, but they were removed before the model was trained for these tests. We did this to keep the testing and training data as separate as possible. The final language, Latin was not part of the set. We still offer the same caveat as before: there is enough similarity between the removed languages and those that still exist in the set that we cannot completely say that all words of a language are removed. There are languages similar to English and there are languages similar to French. Obviously the correspondences between such languages would mirror those between English and French. However, this is a property of natural languages, one which made a learned approach seem more appealing. Even if a language was not studied at all, just as Latin is not considered in our training data, there should be similar languages we can train on to learn the correspondences we want. For comparison with the development set we use the average of all 10 language pairs available for testing. The average cognate percentage for these 10 pairs is 28.4%. Appendix A contains detailed data for every language pair. Also keep in mind that formal tests already have some improvements in them that were discovered during development. The most important of these is the correction for word length added to the Viterbi and forward algorithms (see 6.1.5). Even the formal tests whose results appear in earlier sections have this correction.

Model Input/Output

The input for the training program is a set of example words that exhibit the similarity to be modeled. For these experiments this is the set of cognate pairs within the Indoeuropean languages. The output of the training program will be the three sets of parameters. First the substitution probabilities, represented by a matrix with dimensions equal to the token alphabet size of the corresponding languages. In this case we have a symmetric representation with a token alphabet size of 26. The second set is the insertion and/or deletion probabilities for each token. Finally we get the transition probabilities. The standard model has 5 of these.

For the ranking algorithms, the input is any pair of words. The cognate recognition task uses a mix of cognates and non-cognates. However, all of the

pairs selected have a common meaning, they just are not necessarily cognates. The output of any ranking program is a score for each pair in the input. This score represents (relatively) how much of the learned similarity the pair exhibits. For these experiments words that are very similar should be cognates.

Baseline: Normalized Edit Distance

Normalized edit distance is simply a minimum edit distance calculation normalized by the length of the longest word in the pair. The section on dynamic programming in Chapter 4 has pseudo-code for the minimum edit distance algorithm. If you use a standard approach, where each operation has a cost of 1, except the substitution of identical tokens which has a cost of 0, then we get a precision of 0.568 for the average of the 10 language pairs in our test set. This number can be increased by adding some domain knowledge to our cost structure. First, we set the substitution costs of non-identical vowels to 0.5. This is because vowels are often transformed between languages. We also use a substitution cost of 2, to remove the effect non-identical segments have on the overall calculation. By doing this we can boost the effectiveness of the normalized edit distance score, achieving a test data average precision of 0.624. This number represents how well a simple method could preform in a well studied domain. The earlier precision of 0.568 is more likely for tasks that have little or no domain knowledge available.

6.1.2 Experiments on Trained Parameter Effectiveness

The following experiments represent variations in the way the trained model is used. The purpose is to examine how useful each part of the trained model is to the overall performance of the system. The trained model can be broken up into three sets of parameters: the substitution probabilities, the insertion/deletion probabilities, and the transition probabilities. Obviously, the substitution costs constitute the core of the model, without them we have no truly useful information with which to rank the word pairs. However, how useful are the other parameters, and is it really worth training them? This set of experiments attempts to answer that.

Because of the way our training data was created, it contained an exceptionally large number of pairs. The total pair count for the data set is 235483. This caused the training program to take a long time to run, not normally a problem once the system is in place, since training only needs to be done once. However in order to develop the system, training needed to be done numerous times. Therefore, we created a smaller training set that included only pairs for which both words had at least 4 tokens (length greater than or equal to 4). This effectively cut our training set in half to 118485 pairs, greatly reducing the time needed to train.

For our experiments we need to make a distinction between the two variations of the log odds algorithm available to us. The first (referred to in the tables as L.O.C.) is the model as presented by Durbin et al. [8], while the other (L.O.L.) represents our more complex version of the log odds algorithm, where we separate the insertion and deletion values for the word similarity and random models. Also, whenever we assume that the insertion and deletion costs are constant, the two log odds models become equivalent.

	4+ Data		Full Data	
Algorithm	IK	PR	IK	PR
Vit	0.582	0.923	0.584	0.920
For	0.566	0.911	0.570	0.909
L.O.C	0.764	0.990	0.777	0.990
L.O.L.	0.764	0.990	0.777	0.990
F.L.O.	0.350	0.942	0.358	0.942

Table 6.1: Comparing two training sets using only substitution costs

Table 6.1 shows a comparison of the effectiveness of the model trained on the two data sets. This also represents the performance of our model using only the substitution scores (for log odds and forward log odds) or the substitution probabilities (for Viterbi and forward). In other words, only the substitution costs are used when calculating the final score for a pair. The other model parameters are left at constant values. The constant values for the insertion and deletion probabilities vary depending on the algorithm. For Viterbi and forward they are set uniformly at $\frac{1}{26}$ (since our alphabet size is 26). For our

log odds family of algorithms we follow the design of the original version of the log odds algorithm and choose constant insertion and deletion values set at the corresponding letter frequencies of the languages being examined, allowing the two to cancel out. We do this to ensure that the insertions and deletions have no effect on our results. For the Viterbi and forward algorithms we want uniform values so all the probabilities are equal. In the log odds algorithms we chose letter frequency so that the insertion and deletion probabilities cancel out with the probabilities from the random model. For any algorithm the transitions are each set to 0.3 except for τ which is given a smaller value of 0.1, since it is less essential to the model.

There is very little difference between the model trained with the entire data set and the model trained with the smaller sub-set of the data. The model trained with more data usually outperforms the other but not always, and the differences between the two are minor enough that we can conclude there are no significant differences between the two models. It seems that the correspondences that exist between small words are also contained in larger words, meaning there are no transformations that only exist between small words and nowhere else. Choosing length to cut down our data did not have an adverse effect on the performance of our trained model. However, since we have the model trained on the full set of data, it is those parameters that we will use for most of the experiments, unless training time becomes problematic.

The same table shows us the effectiveness of the log odds approach. Without any corrections or tweaking it does an excellent job on the recognition task. It is possible that the Viterbi and forward algorithms suffer from problems caused by word length and the scores need to be adjusted to help alleviate this deficiency (later experiments will determine if this is true or not). Those algorithms still do a decent job, at least when compared to random selection of cognates. The forward version of the log odds algorithm achieves good performance on the Polish/Russian data set, but does poorly on the Italian/Serbo-Croatian. The lower percentage of cognates makes the Italian/Serbo-Croatian data more difficult and as will be seen later, the forward log odds algorithm works best when given complete information in the form of the entire trained

model.

Parameters Set Constant	IK				
	Vit	For	L.O.C	L.O.L.	F.L.O.
indel, trans	0.584	0.570	0.777		0.350
trans	0.601	0.599	0.777	0.792	0.396
indel	0.610	0.604	0.767		0.490
none	0.612	0.615	0.767	0.734	0.508

Table 6.2: The effect of each set of trained parameters (part 1)

Parameters Set Constant	PR				
	Vit	For	L.O.C.	L.O.L.	F.L.O.
indel, trans	0.920	0.909	0.990		0.942
trans	0.926	0.920	0.990	0.993	0.953
indel	0.937	0.936	0.992		0.976
none	0.938	0.937	0.992	0.991	0.977

Table 6.3: The effect of each set of trained parameters (part 2)

Tables 6.2 and 6.3 show the consequences of using more learned parameters during testing. The second row shows the effect that adding learned insertion and deletion scores has on the algorithms. Since we are training on symmetrical pairs of words the insertion and deletion costs are equal. This is fine when you only have one "language" or alphabet as we do with this data. Not surprisingly this addition improves each of the algorithms to some extent, but not a significant degree. The best performance is obtained by using the modified log odds algorithm with uniform transition probabilities. The next row shows the same small increase in precision when only transition probabilities are added to the algorithms. For this experiment the insertion and deletion costs are returned to their constant values. Again we see a small increase in the accuracy of the rankings, but nothing too significant.

The most unexpected results are shown the final row of the table, which reflects the addition of both the insertion and deletion costs along with the transition probabilities. The standard result occurs for the Viterbi and forward algorithms: a small gain in recognition accuracy. However for Viterbi-based log odds we get a very unintuitive result; the accuracy of the rankings

drops for both data sets. In fact, for the IK set, it drops below the level obtained by just using substitution costs. It seems strange that two additions that improve results on their own would cause a decrease when used together, especially considering they are fairly separate in what they tell us about the model. One possible reason this is occurring is the extra parameter introduced in the random model, η. This parameter, along with the various transitions from the cognate model, is used to determine a penalty structure for insertions and deletions. This makes the algorithm a bit more sensitive to the interplay between the various transition parameters, which can result in larger insertion and deletion costs. Since the algorithm normally assumes that insertion and deletion *probabilities* are constant, combining the learned insertion and deletion probabilities with the log odds algorithm's existing cost structure seems to be exaggerating the penalty for insertions and deletions. This causes the algorithm to be a bit too aggressive when aligning tokens, reducing the overall performance.

Overall, it is the forward log odds algorithm that most benefits from the addition of more trained parameters. There is a significant increase in the precision of this algorithm with the addition of each set of parameters. It is unusual that the forward log odds algorithm behaves like the simpler algorithms that do not contain the random model. It seems that weaker algorithms require more data to be effective, while the more powerful algorithms function well with simplified data that allows them to concentrate on the more important facets of the data. Because of this trend we decided against a constant insertion and deletion version of the forward log odds algorithm. Instead, our forward log odds algorithm will always use the learned insertion and deletion values of the trained word similarity model.

The test data shows a slightly different story. Table 6.4 shows a slight drop in precision whenever insertion/deletion information is added. The exception to this is the forward log odds model, which still benefits from every extra bit of data it receives. Notice that the Viterbi and forward algorithms both profit from the inclusion of transition probabilities. This gives them a more detailed penalty structure. On the other hand, the Viterbi log odds algorithms

Parameters Set Constant	Test Data Average				
	Vit	For	L.O.C.	L.O.L.	F.L.O.
indel, trans	0.569	0.537	0.702		0.536
trans	0.565	0.516	0.702	0.698	0.553
indel	0.630	0.628	0.685		0.634
none	0.619	0.615	0.685	0.662	0.639

Table 6.4: Formal tests of the effect of adding more trained model parameters

get lower performance as more data is added to them. These algorithms work best when they can concentrate on the most important data, in this case substitution costs. So far the best performance is the log odds algorithm with constant insertions, deletions, and transitions.

6.1.3 Viterbi vs. Forward-Backward for EM Training

Initial training was done with two different EM algorithms. The first was based on a Viterbi approach, only looking at the best alignments, the second used the standard forward-backward approach. One of the biggest obstacles with the Viterbi algorithm is its sensitivity to initial parameter settings. The standard forward-backward algorithm works well when given uniform starting values, but this leads to poor results when used as a starting point for the Viterbi algorithm, as the experimental results show. We attempted to overcome this limitation by using some domain specific knowledge. One experiment gave the diagonal entries higher values than the other entries. The other experiments attempt to use phonetic similarity as a means to determine what tokens should get higher initial scores. The first version of phonetic initialization creates positive scores in the initial score table if the pair is phonetically similar. These decisions were made by a separate program, developed by Kondrak as part of his PhD thesis [17], that is able to measure phonetic similarity. This program produces a score for any token pair and that score can be either positive or negative. For example the phonemes "f" and "v" get a positive score, 25. However, "f" and "e" get a score of -30. Remember that our data was not designed to be examined phonetically, but we are only using phonetics for a starting point, not for the actual scores. If the pair of tokens were given

a positive score by this program, then they got a positive initial score in the table. The second phonetic initialization uses the similarity of vowels as a threshold. This was done because vowels tend to always be similar. The score table gets a positive entry if the pair exhibited more similarity then the best pair of vowels. The main problem with the phonetic approach is that our data is represented as English letters, not phonemes, but hopefully there is enough overlap between phonemes and letters in our data that such initial conditions help avoid any of the poor local maximums that plague the Viterbi algorithm is this domain. The Viterbi algorithm used is the log odds algorithm, since it tends to preform better and has a built in length normalizing component.

Initial Conditions	IK	PR	Test Data Average
Uniform	0.455	0.878	0.386
Diagonal	0.565	0.980	0.530
Phonetic (zero-threshold)	0.562	0.972	0.589
Phonetic (vowel-threshold)	0.486	0.945	0.586

Table 6.5: Viterbi based EM algorithms

Table 6.5 shows that the Viterbi algorithm preforms poorly as an EM algorithm for the data sets we are working with. The average for the test data is a bit better than the development results, but this approach still gets precisions well below the mark set by similar forward-backward trained models. The problem mainly lies in the way that the Viterbi algorithm works. Since it can only consider the alignments it initially creates when getting counts, it is difficult to change from the first set of alignments. When there is no good indication of what starting parameters to use with the algorithm it is difficult to get good results when doing Viterbi training. For every token we know nothing about (except for identical tokens, this is most of them) we must either make the choice to align them or not. If we align them, they will always get aligned in the data, causing the algorithm to align every token when comparing two words. Obviously this is not the behavior we are looking for. On the other hand if we do not align the tokens we know nothing about, then we will not get any new information from the data. We align the tokens we "guessed" were good initially and can learn almost nothing from the rest of the data. This all or

nothing problem with the Viterbi algorithm is a problem of our domain. If we had more precise starting conditions that we just need to refine to better match the data we were training on, the Viterbi approach would probably preform much better. Unfortunately we are looking for approaches that can be used in domains that are not well studied and the Viterbi approach does not seem applicable to such tasks. What we need to be able to do is change our insertion and deletion values during training. It might be possible to start by aligning our data aggressively, and then by decreasing the penalty for gaps, remove some of the weaker alignments from being preferred over insertions/deletions. This approach doesn't quite fit with the Hidden Markov Model approach, but it is something that could be examined at a later time. The remainder of our experiments concentrate on the forward-backward training algorithm and how well we can get it to preform.

6.1.4 Experiments with Model Complexity

While the previous set of experiments dealt with how much of an effect each individual parameter component had on the overall model, we still trained the model as a whole in order to get those probabilities. It would be interesting to see if *testing* with only the substitution costs (or whatever else might interest you), preforms better if only the substitution costs are *trained.* The first experiments trained the insertions, deletion, and transition costs, and then sets them to constants during the testing phase. The following experiments keep those parameters constant throughout the training process. Since the trained model more closely resembles the model used for testing, it may allow for better result.

The results show that the less complex the model the worse the overall performance of the model. Keeping transitions constant is worse than training the entire model (see Tables 6.2 and 6.3) and continues to degrade if the insertions and deletions are held at a uniform value as well. There are a few exceptions, mostly occurring when insertions and deletions are kept constant during training. However, the differences are too small to conclude there are any significant improvements in precision over the fully trained model.

Constant Parameters During (After) Training	IK				
	Vit	For	L.O.C	L.O.L	F.L.O
none	0.612	0.615	0.767	0.734	0.508
none (trans)	0.601	0.599	0.777	0.792	0.396
none (indel)	0.610	0.604	0.767		0.490
none (trans, indel)	0.584	0.570	0.777		0.350
trans	0.608	0.612	0.740	0.766	0.474
indel	0.618	0.618	0.777		0.505
trans, indel	0.599	0.582	0.706		0.376

Table 6.6: Training with constant parameters (part 1)

Constant Parameters During (After) Training	PR				
	Vit	For	L.O.C.	L.O.L.	F.L.O.
none	0.938	0.937	0.992	0.991	0.977
none (trans)	0.926	0.920	0.990	0.993	0.953
none (indel)	0.937	0.936	0.992		0.976
none (trans, indel)	0.920	0.909	0.990		0.942
trans	0.940	0.937	0.986	0.992	0.974
indel	0.939	0.938	0.993		0.978
trans, indel	0.931	0.923	0.988		0.952

Table 6.7: Training with constant parameters (part 2)

When the models trained with constant values are compared against the fully trained models whose parameters are later set constant for testing the results are less consistent. For Viterbi, forward, and forward log odds there is an overall increase in performance when training with constant values. However, the Viterbi log odds algorithms get a decrease in performance, instead preferring the fully trained model with constants set afterwards, except in the case of insertion/deletions where the opposite is true for the constant indel version of the Viterbi log odds algorithm. One of the main reasons this may be occurring is that the training algorithm is in fact the forward-backward algorithm. Since we are training with the forward algorithm it makes sense that training the model the same way we are testing it would work well when testing with the forward algorithm. The Viterbi and forward algorithms are in many ways the same, so it is likely that they would share many of the same properties. The log odds algorithm on the other hand uses the data produced by the training algorithm in a transformed way, no matter how the model

is trained. The addition of the random model and the parameter changes that occur when adding that model to the algorithm ensure that the trained model is adapted from what it originally was. Since this adaptation always occurs, the algorithm should prefer to use the most precise versions of the various costs, and would not benefit from a partially trained model as the other algorithms do.

Constant Parameters During (After) Training	Test Data Average				
	Vit	For	L.O.C.	L.O.L.	F.L.O.
none	0.619	0.615	0.662	0.685	0.639
none (trans)	0.565	0.516	0.702	0.698	0.553
none (indel)	0.630	0.628	0.685		0.634
none (trans, indel)	0.569	0.537	0.702		0.536
trans	0.602	0.572	0.686	0.699	0.604
indel	0.622	0.614	0.682		0.627
trans, indel	0.625	0.625	0.672		0.641

Table 6.8: Formal tests after training with constant parameters

For the formal tests the fully trained model does not perform as well as the models with some values kept constant. Keeping only the transitions constant tends to perform worse, except in the case of the Viterbi-based log odds algorithms. All of the algorithms benefit from keeping both indel and transitions constant during training, at least when compared to the unmodified, fully trained model. Table 6.8 shows that keeping parameters constants during training is usually a benefit to Viterbi, forward, and forward log odds algorithms, but is detrimental to the other log odds algorithms. However, when indels are kept constant during training all of the models suffer, although not always significantly. These experiments again show that the forward log odds algorithm behaves more like the Viterbi and forward algorithms. This is another example of the testing algorithm matching the training algorithm, since both the testing and training programs have the forward algorithm as part of their structure. It appears that this similarity is more important then the structural similarity between the log odds algorithms. As such, the forward log odds algorithms benefits from testing with the exact same model that was trained. The best variation is still the log odds algorithm with substitution

scores derived from a fully trained word similarity model and constant insertion, deletion, and transition probabilities set after training.

6.1.5 Correcting for Length

Our next set of experiments deal mostly with changes to the algorithms that occur after training, in order to more effectively complete the cognate recognition task. The first of these is to examine the effect of word length on the precision of the various algorithms. We have already seen that the log odds algorithm is implicitly normalizing since it contains the division of two models. However the Viterbi and forward algorithms have no such mechanism. Since we are dealing with multiplicative probability chains, we know each number in the chain will be at most one. In fact, we do not allow any transitions or emission to get probability one (or zero) since we can never be that certain about any single alignment, and since in the case of a one, it would force all the other probabilities to be zero in order to satisfy the restrictions on Hidden Markov Models. We are continually multiplying numbers smaller than one together every time we process a token (or a token pair). This means that the result of this multiplication will get smaller and smaller the longer the words are. This causes true cognates to get lower scores than non-cognates if the true cognates are a few tokens longer. In order to correct for this we use the following correction for length,

$$\text{Correction} = \frac{1}{C^n},$$

where n is the length of the longest word in the pair, and C is a constant which will need to be determined by experimentation. This gives our forward algorithm a final calculation of

$$\frac{P(\mathbf{O}|\mu)}{C^n},$$

while the final score for the Viterbi calculation becomes

$$\frac{P(X^*)}{C^n}.$$

Constant	IK		PR	
	Vit	For	Vit	For
1.0	0.612	0.615	0.938	0.937
0.5	0.624	0.626	0.944	0.944
0.1	0.660	0.662	0.961	0.961
0.05	0.672	0.671	0.968	0.968
$1/26 \approx 0.0385$	0.673	0.674	0.971	0.971
0.01	0.702	0.695	0.981	0.983
0.005	0.649	0.606	0.981	0.981
0.0001	0.372	0.343	0.928	0.921

Table 6.9: Correcting for word length

The first entry in Table 6.9 represents no correction for length (division by one). As the constant gets smaller, the correction is getting larger. The table also includes the value $\frac{1}{26}$ since this represents the same sort of value that would be used by the log odds algorithm during its implicit normalization. The random model uses a larger variety of values based on letter frequency, but this is the average value for each token. Even the smallest correction begins to have a positive effect of the cognate recognition task. There is a limit to how much correction can be done however. This follow our intuition, if the correction for length becomes too great we will have the opposite problem: long words will be preferred over short ones no matter how good the actual alignments are.

Another common method to correct for length is to take the n^{th} root of the final calculation, where n is the length of the longest word. On the development set the Viterbi algorithm achieved precisions of 0.605 and 0.592 for the Italian/Serbo-Croatian and Polish/Russian data sets respectively. For the forward algorithm the results were 0.971 for both data sets. Since this approach did not preform as well as the previous calculation, it was not explored any further during formal testing.

It is important to note that our formal testing was done after all of the development experiments were finished. As such the formal tests already include the modifications that were done by hand tuning the algorithms on the development sets. This means that any of the results in previous sections that

are done on the *test* set, already have this correction for length as part of the calculation. We used the constant $C = 0.01$, since it got the best performance on the development data.

6.1.6 Reducing the Number of Parameters

One of the more difficult decisions to be made when creating a Pair Hidden Markov Model, or any Hidden Markov Model, is choosing what parameters the model should have. For our model having a separate state for each edit operation seems intuitive, allowing us to divide each operation into costs or probabilities that make sense when you consider token by token alignment. One of the more problematic issues arrives when discussing the transition parameters. The biological model is set up in such a way that it allows for affine gap penalties, something common in the alignment of biological sequences. The question still remains if such a gap policy would be beneficial to our word similarity model. In addition there is also the question of the effect of our end state transition parameter τ. Since our algorithms always know the lengths of the words we are processing ahead of time, we do not need to use τ in order to know when to stop our iterations. The end state and transitions to it are included to allow for the generation of data, but if you do not need a generative model do you need to have this state in the model? It may be that including an end state has no effect on the ranking and alignment of word pairs. Since Hidden Markov Models put a restriction on the transition probabilities (all transitions from a state must sum to one) then including the end state does not just give us a simple constant multiplier at the end of our probability chain. Including it takes away some probability mass from the other transitions between each of our states, since part of that mass must go into the transition to the end state.

It is simple enough to remove τ from our transitions. The remaining probability mass goes to the transitions that lead to the substitution state, since this tends to be the highest out of all of the probabilities. The comparison in Tables 6.10 and 6.11 are against the fully trained model after it has been corrected for word length.

	IK				
	Vit	For	L.O.C	L.O.L	F.L.O
With τ	0.702	0.695	0.767	0.734	0.508
Without τ	0.726	0.699	0.755	0.720	0.466

Table 6.10: The effect of removing the end state (part 1)

	PR				
	Vit	For	L.O.C.	L.O.L.	F.L.O.
With τ	0.981	0.983	0.992	0.991	0.977
Without τ	0.985	0.985	0.992	0.991	0.969

Table 6.11: The effect of removing the end state (part 2)

There is a small increase for the Viterbi and forward algorithms when the end state is removed. The extra probability mass allows the other transitions to more exactly represent the form needed by the cognate model. Log odds, including all variations, on the other hand gets a decrease in precision for the Italian/Serbo-Croatian data. Again, the log odds family of algorithms is more sensitive to transition changes since their entire gap penalty structure relies on it. Keep in mind that the best performance obtained by the log odds algorithm is still from constant, uniform transition parameters. Such a choice allows for a more consistent penalty between the various states.

	Test Data Average				
	Vit	For	L.O.C.	L.O.L.	F.L.O.
With τ	0.619	0.615	0.685	0.662	0.639
Without τ	0.620	0.613	0.666	0.641	0.603

Table 6.12: Formal tests for removing the end state

The formal tests in Table 6.12 show that using the model without the end state decreases the performance of the algorithms. The Viterbi and forward algorithms show no significant difference, but all of the log odds algorithms have a significant lowering of their cognate ranking precision. We can conclude that the end state is important to the word similarity model, and the information it adds, mostly through the transition probabilities used to get to the end state, can increase the performance of our system.

It is possible to reduce the number of parameters even further by allowing

only one transition parameter. This is done by keeping all of the transitions to a state constant, for all the previous states. By combining this with the symmetry we already have between insertion and deletion states and the removal of the end state, we now only need to consider the probability of entering the substitution state and the other transitions will be set by the properties of Hidden Markov Models. The random model still has its parameter η. In the following series of experiments it was found that a lower value of $\eta = .0.1$ preformed best. This was determined by experimenting with different η along with the single word similarity parameter (x in the table). The good thing about having so few parameters is that it is a simple matter to determine during development which value seems to be better, since the search space is so small. Tables 6.13 and 6.14 show some of the values for x and how well each one works. In that table $1 - \epsilon$ represents choosing a value as close to 1 as possible without completely removing the remaining transitions from the model. For these experiments $\epsilon = 0.0001$.

	IK				
x	Vit	For	L.O.C	L.O.L	F.L.O.
0.1	0.518	0.419	0.743	0.750	0.302
0.2	0.616	0.496	0.799	0.789	0.333
0.3	0.650	0.569	0.805	0.792	0.365
1/3	0.663	0.579	0.804	0.790	0.374
0.4	0.683	0.610	0.805	0.787	0.389
0.5	0.705	0.647	0.793	0.774	0.410
0.6	0.721	0.674	0.779	0.757	0.431
0.7	0.725	0.694	0.761	0.735	0.459
0.8	0.735	0.716	0.740	0.704	0.469
0.9	0.726	0.720	0.695	0.670	0.474
$1 - \epsilon$	0.625	0.625	0.566	0.562	0.479

Table 6.13: Using only a single transition parameter (part 1)

When the parameters are tied together like this, we are trying to answer one question; which state are we more likely to be in? For this data set that state is the substitution state. Since our task is to recognize cognates this is what would be expected. A good alignment will mostly contain substitutions, since gaps only occur when there are no good alignments for tokens in the

x	PR				
	Vit	For	L.O.C.	L.O.L.	F.L.O.
0.1	0.919	0.902	0.982	0.988	0.898
0.2	0.951	0.940	0.992	0.992	0.927
0.3	0.964	0.956	0.993	0.993	0.941
1/3	0.968	0.961	0.994	0.993	0.945
0.4	0.972	0.968	0.994	0.993	0.949
0.5	0.978	0.976	0.994	0.993	0.955
0.6	0.982	0.981	0.993	0.992	0.960
0.7	0.985	0.985	0.992	0.992	0.965
0.8	0.986	0.986	0.992	0.990	0.968
0.9	0.986	0.987	0.990	0.987	0.971
$1-\epsilon$	0.963	0.964	0.962	0.957	0.955

Table 6.14: Using only a single transition parameter (part 2)

words. So essentially when x has a high value then gaps are penalized more. Since the forward algorithms (both the regular and log odds forms) look at every possible alignment they benefit the most from a very high probability of going into the substitution state. There is a limit to how high x can go. Any pair whose words are not of equal length requires gaps in order to create an alignment, so having too much of a gap penalty can be detrimental to overall performance. The log odds algorithms preform better when a lower probability is given to the match state. This is again due to the harsher penalties for gaps employed by the log odds algorithms. If we reduce the probability of entering a gap state too much then the penalty becomes too high. Since every state has one transition to the match state and two transitions to gap states, the x value of $\frac{1}{3}$ represents uniform transition probabilities. This is the same behavior we saw in previous experiments, log odds preforming best when the transitions between states are equal, or close to it.

For the formal tests we used only the most promising value of x for each algorithm. In the case of Viterbi, forward, and forward log odds, this is $x = 0.9$. For the remaining log odds algorithms the best choice seems to be $x = 0.3$, since it has better performance on the more difficult of the two development sets (IK). Table 6.15 gives the precision of the best single parameter model for each algorithm.

Transitions	Test Data Average				
	Vit	For	L.O.C.	L.O.L.	F.L.O.
multiple	0.619	0.615	0.685	0.662	0.639
single	0.642	0.648	0.701	0.701	0.602

Table 6.15: Formal tests using only a single transition parameter

The results of this experiment give us the best performance out of all variations of the Viterbi and forward algorithms. This is interesting because it follows a common trend. The best performance tends to involve first training the entire model, then using the trained substitution scores, but simplifying other parts of the model. In this case we are still using the learned insertion and deletion probabilities, but the transitions are being simplified. For the Viterbi-based log odds algorithms we get precision that is nearly equivalent to the best of the previous variations. This is not surprising since the value of the single parameter is such that all transitions are roughly equivalent. Only the forward log odds suffers from this approach. It still works best when all of the parameters of the trained model are used together.

6.1.7 Experiments with Discrete Emission Costs

This experiment follows an approach used by Mann and Yarowsky [20], where they transformed the probabilities of their stochastic transducer into discrete classes that could be used by a simple Levenshtein distance approach. We have tried the same, turning all of our scores from the log odds substitution matrix of the log odds algorithm (fully trained model) into one of three weight classes: 0.5, 0.75, and 1. As was done in their experiment we gave identical tokens a zero cost. We chose to give negative scores a cost of 1, since they represent bad matches in the score table. Moderate matches, those with non-negative values less than one were given the value 0.75. The remaining scores, those that represented learned correspondences within our language family, got the cost of 0.5. Table 6.16 shows the results with various indel (insertion/deletion) costs. All of the entries used a cost based Viterbi algorithm, where lower costs are preferred.

Because of the discrete nature of our values it is easy to describe the behav-

indel cost	IK	PR
2.0	0.634	0.965
1.0	0.672	0.979
0.75	0.683	0.983
0.6	0.700	0.984
0.5	0.701	0.984
0.4	0.687	0.980
0.3	0.674	0.977
0.2	0.655	0.974

Table 6.16: Scores transformed into discrete values

ior of the algorithm based on the indel cost. For example, if the indel cost is 1, then it is just as bad to align a token to a gap then to the worst possible choice of token (which also has a cost of one). An indel cost of 0.4 would always gap two tokens whose alignment cost is 1, since the two gaps would only cost 0.8. These two perspectives are useful to see why values around 0.5 or 0.6 get the best performance. Such a value makes gapping a bad choice, either the same or slightly worse than any alignment. This gives the algorithm more freedom for determining alignments since it can gap or align bad sequences at about the same cost, allowing the similar subsections of the words to have a greater effect on overall cost. They also keep a minimum penalty for single token gaps which often occur in cognates, when one word has an extra token. Such extra tokens will not be greatly penalized. However, the overall performance of this technique is significantly lower than that of the fully trained model from which the costs are calculated. Even the other simplified models that are based on the fully trained model (those with some parameters set to constant, uniform values) get better precision. Our results differ from those of Mann and Yarowsky, but these results seem more intuitive. We are taking a finely tuned set of parameters and ignoring much of the detail that was learned through training. If our parameters were overtrained, then this approach might be better, but the results suggest that the more detailed model is preferable for cognate recognition.

For formal testing we used only the best indel cost, 0.5. With this value the average of all of the test runs was 0.664. This is higher than the best

of the regular Viterbi or forward algorithms, but still falls well short of the precision achieved by the log odds algorithms. This is as expected, since we use the scores of the log odds algorithm to derive the discrete costs. Since we are simplifying the main component of our model, it should have a negative effect on the models performance, and it does.

6.1.8 Removing Multiple Paths

Multiple paths can be a problem when your model considers them as separate, but in practice it may be better to consider the paths the same [3]. Consider the following two fictional alignments

```
x  y  -  x
x  -  z  x

x  -  y  x
x  z  -  x
```

In our model these two alignments would be treated as different alignments, each with the same probability. If we want to consider them as equivalent alignments then we get inconsistencies between Viterbi alignments and the corresponding forward calculations. This happens because in the forward algorithm the two alignments would contribute equally to the calculation, while in the Viterbi only one of them would ever be considered. For the Viterbi algorithm to function correctly we would need to allow only one such alignment to be legal, and it would need to have twice the probability. It is also possible just to consider the alignments as separate, in which case no changes are needed.

To determine the effect of such inconsistencies we have trained two separate similarity models. The first allows no transitions between insertion and deletion states. This is the original bioinformatics model, which we have already discussed is not well suited for cognate alignment. The second allows a transition from the deletion state to the insertion state, but does not have a transition in the opposite direction. This better preserves the needs of the alignment functionality of our model, but it does remove the symmetry we have been preserving between insertion and deletion states.

	IK			PR		
	No λ	One λ	Both λ	No λ	One λ	Both λ
Vit	0.700	0.700	0.702	0.981	0.981	0.981
For	0.694	0.694	0.695	0.983	0.983	0.983
L.O.C.	0.768	0.768	0.767	0.992	0.992	0.992
L.O.L.	0.734	0.734	0.734	0.991	0.991	0.991
F.L.O.	0.508	0.508	0.508	0.977	0.977	0.976

Table 6.17: Various transition structures between X and Y states

There seems to be no discernable difference between the models with or without multiple paths. This comes from examining exactly what our model does. It is a model to describe word similarity. This means we are training it on words we believe to be similar, and as such, there should be few gaps when a pair of similar words is aligned. The transition between insertion and deletion states is the least used, because we rarely need to do a large series of insertions and deletions at a single time. For dissimilar words that are still cognates, it is more often the case that one word has many extra tokens, usually in the form of prefixes or suffixes. These extra tokens are handled by a single state, insertion or deletion depending on the order the words are paired in. We have discovered a property of our model that is interesting and could be useful in the future: the transition between insertion and deletion states has little to no effect on the rankings of the word pairs. However, it is important to remember that the alignments produced by such models would be significantly different, so choosing either the model with one or two transitions between insertion and deletion states would be preferred for word similarity.

	Test Data Average		
	No λ	One λ	Both λ
Vit	0.619	0.619	0.619
For	0.615	0.615	0.615
L.O.C.	0.685	0.685	0.685
L.O.L.	0.662	0.662	0.662
F.L.O.	0.638	0.639	0.639

Table 6.18: Formal tests with various transition structures between X and Y states

Our formal tests confirm that there is no difference between these models. All of the results are almost identical as can be seen in Table 6.18. In fact it is not just the averages that are identical. Each language pair gets essentially the same precision no matter which structure we use for transition between the insertion and deletion states. Multiple combined insertions and deletions just don't occur much in this domain, and as such we do not need to worry about multiple paths in order to achieve good performance.

6.2 Phonetic Experiments

Since phonetic representations are often a much better indication of cognate pairs then the more abstract orthographic representations it would be useful to examine how well our system works when given phonetic training data. Unfortunately, such data is not often available, certainly not to the extent that was used for training the previous word similarity models. Instead we have tried to develop experiments that could be done with our system using only a limited amount of phonetic data.

6.2.1 Development

Our solution was to use an experiment that involved filtering out cognates from a list that contained both cognates and non-cognates. The idea is that for a single pair of languages, the correspondences that exist between cognates should remain relatively consistent for each example pair. On the other hand, the pairs that are not cognates should create a random set of correspondences. Thus, if the model is trained on this mixed data, then the cognate correspondences should occur more regularly than any other correspondences that exist solely by chance. If this is true then our model can get the best performance on the training data by emphasizing the correspondences in the cognates, giving them higher scores in the rankings. Essentially we are filtering out correspondences that occur by chance, since they should not occur in any great number relative to the true cognate correspondences. The frequency that the letter occurs in the data should not have an effect, since the counts for

all correspondences with that letter will be larger. The training data consisted of 200 pairs of words between languages, each with a varying percentage of cognates. The data used in these experiments are the Italian/Serbo-Croatian and Polish/Russian sets used in previous experiments.

We used both the Viterbi and forward-backward training algorithms for this task. We had to restrict the models since we did not have enough information to accurately estimate all of the parameters. For the forward-backward algorithm we assume uniform, constant insertion and deletion values based on the observed alphabet size for each language in the pair. The observed alphabet is the alphabet created by examining the tokens that occurred in the training data. Since we are testing and training on the same data this assumption is adequate. Our Viterbi log odds algorithm requires that we have frequency probabilities for each token. We had to use uniform token frequencies for the random model, since when we tried getting frequency information from the data as an experiment the results were terrible (often below the cognate percentages of the data sets, our lower bound). The Viterbi-based log odds training algorithm required us to use a number of different initial conditions in an attempt to increase the model's performance. These are the same initializations that are described in 6.1.3.

Training	Testing	IK	PR
F.B.	Vit	0.534	0.990
F.B.	For	0.531	0.990
F.B.	L.O.C.	0.441	0.987
Vit L.O.(uniform init)	L.O.C.	0.278	0.723
Vit L.O.(diagonal init)	L.O.C.	0.410	0.987
Vit L.O.(phon init, normal thresh)	L.O.C.	0.380	0.979
Vit L.O.(phon init, vowel thresh)	L.O.C.	0.386	0.980

Table 6.19: Cognate filtering experiment development

Table 6.19 shows the precision achieved by the various training and testing algorithms. The results show two general trends. First, since the Polish/Russian data has a denser percentage of cognates it is possible to achieve high precision in the filtering task. This happens because the data contains very little noise to confuse the model during training. The second result of

these experiments is the failure of the log odds algorithm to outperform the Viterbi and forward algorithms. The reason this occurs is the greater need for data that the log odds algorithm has. To function properly it needs more information then just the probabilities for token substitutions. It also relies heavily on the random model, which requires some understanding of the languages being studied. Specifically, the random model needs the frequencies that each token occurs in the language. Normally, it is enough to get these frequencies and use them as constants during the training of the model. Unfortunately, without any additional data for each language, we can only assume a *uniform* token distribution. This is almost never the case in a natural language, but it is the best we can do under the circumstances. To make matters worse, we not only need data for each language, but that data has to be in a phonetic form. If nothing else this experiment has exposed a small weakness in the log odds model that does not exist in the other algorithms, the need for more domain information then just token substitution costs.

6.2.2 Results

We also tried the filtering experiment with phonetic versions of the data we used for our formal cognate identification tests. Each language pair was tested separately, and we also calculated the average of all of the possible pairs between all 5 languages. The results of these experiments are shown in Table 6.20. The first entry of each row shows the pair under examination, using the first letter from the names of the languages.

There is not much difference between the various algorithms. The only exception is that the log odds algorithm performs poorly when using the forward-backward trained probabilities. We do however, get good performance with the log odds algorithm if we also *train* with the log odds algorithm. This is occurring despite having to use uniform values for all of the token frequencies. Again, the languages with a higher percentage of cognates are easier to filter than those with a low cognate density. The results for English/German are very encouraging, since the Viterbi and forward algorithms are achieving near perfect cognate recognition, even though the percentage of cognates in

	Training / Testing			
Data	F.B. / Vit	F.B. / For.	F.B. / L.O.C.	Vit L.O. / L.O.C.
EG	0.970	0.970	0.944	0.901
FL	0.860	0.866	0.798	0.805
EL	0.732	0.734	0.694	0.583
GL	0.645	0.649	0.648	0.562
EF	0.451	0.451	0.524	0.581
FG	0.471	0.464	0.459	0.527
AL	0.488	0.500	0.493	0.571
AF	0.410	0.403	0.358	0.406
AG	0.232	0.229	0.190	0.281
AE	0.192	0.195	0.157	0.235
Average	0.545	0.546	0.527	0.545

Table 6.20: Filtering experiment results

the data is less then 60%. These successes show that the algorithms are well suited to working with phonetic data. If enough phonetic data becomes available our word similarity system should be able to increase the precision of its rankings.

6.3 Drug Name Similarity

This experiment follows the ideas presented by Kondrak and Dorr [18], as they examined ways to determine confusable drug names. We have trained our system using a list of drug name pairs that are considered confusable, and then applied four of our ranking algorithms to a larger set containing pairs of names where some are considered confusable and the rest are not. We have used the same evaluation metric that was used in the original set of experiments, and have graphed the results along with one of the approaches used in the paper, *normalized edit distance* (NED). This gives us a point of comparison with the results obtained in the original experiments as well as a baseline to compare our algorithms to each other. Figure 6.2 shows each of our approaches.

The graph shows the same results that occurred on most of our cognate test data. The log odds algorithm with constant insertion and deletion costs gets the best recall, followed closely by the log odds algorithm that was mod-

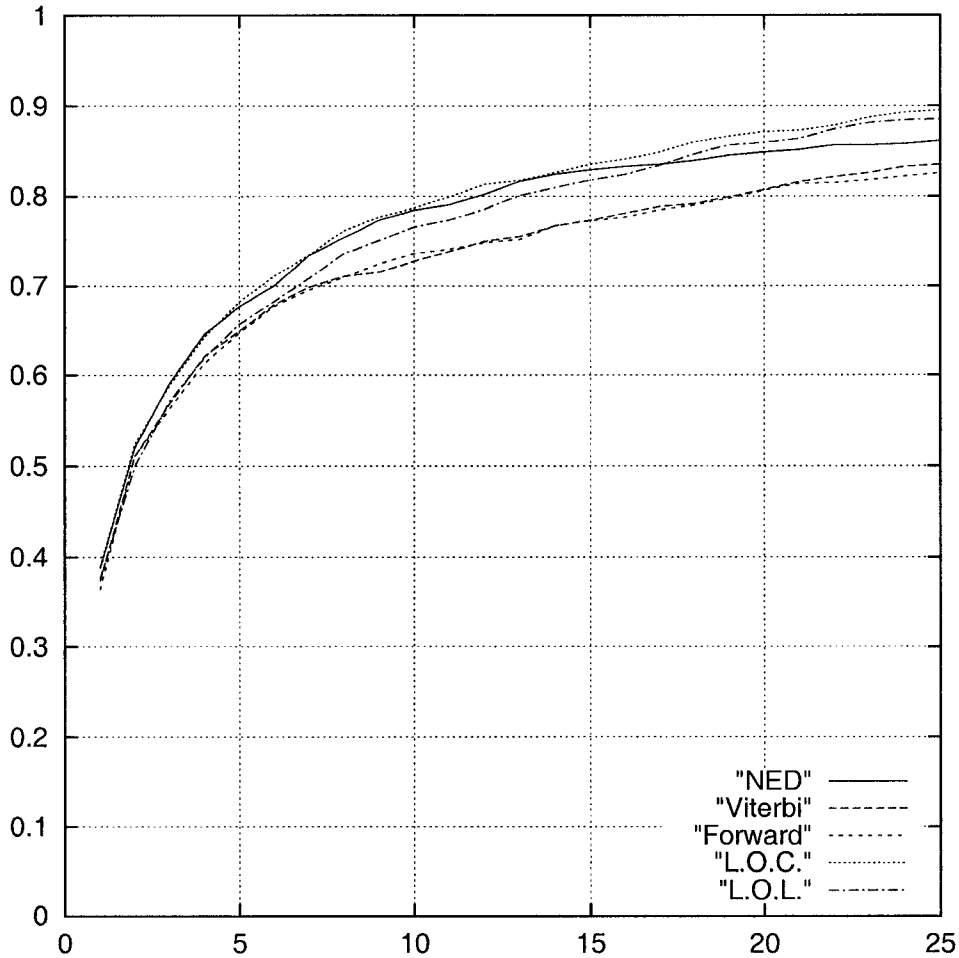

Figure 6.2: Recall at various thresholds using the UPS test set

ified to use the learned insertion and deletion costs. The Viterbi and forward algorithms get the lowest recall, falling short of the mark set by the normalized edit distance. It is good to see that our algorithms perform as well as some of the better techniques used in the original experiments. Our system was not changed in any way before using it with this task. We did no development, instead using the system as it was developed for cognate ranking. This experiment shows the adaptability of our machine learning based approach. Even without any domain knowledge or any changes to the model we were able to get good recall on this task. In addition, this experiment speaks well for our underlying Pair Hidden Markov Model. Having each state represent one of the three core edit operations keeps the model general enough to handle any alignment task. It has proven itself as a useful model for biological sequence alignment, and we have also shown that it can be used (with a few changes) successfully in natural language based tasks. This system would make an excellent starting point for any problem that can be dealt with from the view of word alignments, or rankings based on the corresponding alignment scores.

Chapter 7

Conclusion

Our goal was to create a system that could automatically learn to recognize words that were similar based on some criteria provided during training, and separate such words from those that did not exhibit such similarity or whose similarity exists solely by chance. To this end, we have successfully adapted techniques from the field of bioinformatics by using a Pair Hidden Markov Model.

Our system consists of a variety of algorithms and variations for testing and training that have proved themselves, as a whole, to be useful for recognizing similarity between cognates. The best overall algorithm for cognate recognition was the *Viterbi-based log odds* algorithm. Both the original version, and our modification that uses learned insertion and deletion costs, got about the same precision, working best when the transition probabilities are set to uniform, constant values. Our model with only a single transition parameter, along with all of the learned emission probabilities also performed well. By using more domain knowledge in the form of the random model, the log odds algorithm was able to better separate true similarity from chance similarity. This kind of information is easy to obtain and can usually be learned directly from the training data. In addition the log odds algorithms automatically normalize the results based on the lengths of the words under examination.

The algorithms also seem to work well given the much more available orthographic data. Yet, the system shows promise for working well with phonetic data. We had such data in mind when we designed the system, as it tends

to be more powerful when aligning words in natural languages. It remains to be seen if phonetic training data in the same amount as the orthographic data we used would further boost the performance of the system. Creating phonetic data is a difficult procedure however, and a useful task to study on its own. Of course, such data becomes a necessity if we wish to study languages that have complex alphabets, such as Asian languages. We have also shown how our system can be useful for other tasks, not just cognate identification, since the system was used as is to determine similarity between drug names. Switching to a new domain provided no difficulties for the algorithms, and the results were good considering that no domain knowledge was added, and no development was done for that task.

Our approach also represents a push further into the field of machine learning, without the need for the domain specific knowledge often associated with natural language tasks of this type. This allows our system to be adapted to any number of tasks, even those that are not well studied, as long as we have examples of what would be considered similar words for the job in question. Of course this is still just a first step, machine learning is a very deep and well studied field of research. There are always more approaches to try, more variations that give a different way of using the data available to us. We believe our system gives an excellent starting point from which further research can continue. Our system could be used as a comparison to measure other machine learning techniques, or as the basis for an even larger system that incorporates more approaches.

Bibliography

[1] Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic function of Markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.

[2] Richard E. Bellman. *Dynamic Programming.* Prinston University Press, 1957.

[3] Brona Brejova, Daniel G. Brown, and Tomas Vinar. The most probable labeling problem in HMMs and its applications to bioinformatics. In *Algorithms in Bioinformatics (WABI 2004)*, 2004. To appear.

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

[5] Alexander Clark. Learning morphology with Pair Hidden Markov Models. In *Proceedings of the Student Workshop at ACL 2001*, Toulouse, France, July 2001. ACL.

[6] Michael A. Covington. An algorithm to align words for historical comparision. *Computational Linguistics*, 22(4):481–496, 1996.

[7] Michael A. Covington. Alignment of multiple languages for historical comparison. In *The 17th International Conference on Computation Linguistics*, 1998.

[8] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Protein and Nucleic Acids.* Cambridge University Press, 2000.

[9] Isidore Dyen, Joseph B. Kruskal, and Paul Black. An Indoeuropean classification: A lexicostatistical experiment. *Transactions of the American Philosophical Society*, 82(5), 1992.

[10] Frederick Jelinek. *Statistical Methods for Speech Recognition.* The Massachusetts Institute of Technology Press, 1999.

[11] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice Hall, 2000.

[12] Brett Kessler. *The Significance of Word Lists.* Center for the Study of Language and Information Publications, 2001.

[13] Kevin Knight. Automating knowledge acquisition for machine translation. *AI Magazine*, 18(4), 1997.

[14] Kevin Knight. A statistical MT tutorial workbook. Prepared in connection with the JHU summer workshop, April 1999.

[15] Kevin Knight and Jonathan Graehl. Machine transliteration. *Computational Linguistics*, 24(4):599–612, 1998.

[16] Grzegorz Kondrak. A new algorithm for the alignment of phonetic sequences. In *Proceedings of the First Meeting of the North American Chapter of the Association for Computational Linguistics*, pages 288–295, Seattle, April 2000. NAACL.

[17] Grzegorz Kondrak. *Algorithms for Language Reconstruction.* PhD thesis, University of Toronto, 2002.

[18] Grzegorz Kondrak and Bonnie Dorr. Identification of confusable drug names: A new approach and evaluation metric. To be presented at the Twentieth International Conference on Computational Linguistics (COLING 2004), August 2004.

[19] Grzegorz Kondrak, Daniel Marcu, and Kevin Knight. Cognates can improve statistical translation models. In *Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: companion volume*, pages 46–48, Edmonton, May 2003. HLT-NAACL.

[20] Gideon S. Mann and David Yarowsky. Multipath translation lexicon induction via bridge languages. In *Proceedings of the Second Conference of the North American Association for Computational Linguistics*, Pittsburg, 2001. NAACL.

[21] Christopher D. Manning and Hinrich Schutze. *Foundations of Statistical Natural Language Processing.* The Massachusetts Institute of Technology Press, 2001.

[22] Rose Nash. *NTC's Dictionary of Russian Cognates Thematically Organized.* NTC Publishing Group, 2000.

[23] Franz Josef Och and Hermann Ney. Improved statistical alignment models. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics*, pages 440–447, Hongkong, China, October 2000. ACL.

[24] Franz Josef Och and Hermann Ney. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51, March 203.

[25] Lawrence Rabiner and Biing-Hwang Juang. *Fundamentals of Speech Recognition.* Prentice Hall PTR, 1993.

[26] Eric Sven Ristad and Peter N. Yianilos. Learning string edit distance. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 20 of *5*, pages 522–532. IEEE PAMI, May 1998.

[27] Morris Swadesh. Lexicostatistic dating of prehistoric ethnic contacts. *Proceedings of the American Philosophical Society*, 1952.

[28] Stephen Vogel, Hermann Ney, and Christoph Tillman. HMM-based word alignment in statistical translation. In *Proceedings of 16th International Conference on Computational Linguistics*, pages 836–841, 1996.

Appendix A

Cognate Recognition Test Results

These tables list the results for each language pair used as part of our formal tests. The data is broken up into different tables, such that each table represents one of the 5 main algorithms we experimented with. These are: Viterbi, forward, log odds (constant indel), log odds (learned indel), and forward log odds. There are a few exceptions. Tables A.1 and A.2 give the percentage of cognates in each data set. This is useful as a lower bound on ranking precision. Tables A.5 and A.6 give the results for the experiment with discrete substitution costs. This is separate because it required a different algorithm, essentially a minimum edit distance algorithm that functions as a Viterbi algorithm. Tables A.7 and A.8 give the results on the test data when the log odds version of Viterbi EM training is used. A few abbreviations are used in the experiment column. "cT" and "cQ" represent the use of constant transitions and constant insertions/deletions respectively. "DEC" is used to label the experiments that use discrete emission costs, determined from the trained model parameters. Each row corresponds to one of the experiments discussed in Chapter 6.

Experiment	EG	FL	EL	GL	EF
% Cognates	0.590	0.560	0.290	0.290	0.275

Table A.1: Cognate percentages (part 1)

Experiment	FG	AL	AF	AG	AE	Average
% Cognates	0.245	0.195	0.165	0.125	0.100	0.284

Table A.2: Cognate percentages (part 2)

Experiment	EG	FL	EL	GL	EF
Basic Costs	0.911	0.930	0.698	0.574	0.652
Domain Knowledge Costs	0.907	0.946	0.737	0.630	0.709

Table A.3: Normalized edit distance (part 1)

Experiment	FG	AL	AF	AG	AE	Average
Basic Costs	0.487	0.523	0.471	0.187	0.242	0.568
Domain Knowledge Costs	0.561	0.592	0.512	0.341	0.305	0.624

Table A.4: Normalized edit distance (part 2)

Experiment	EG	FL	EL	GL	EF
DEC	0.928	0.913	0.753	0.687	0.762

Table A.5: Discrete emission costs (part 1)

Experiment	FG	AL	AF	AG	AE	Average
DEC	0.650	0.582	0.523	0.438	0.403	0.664

Table A.6: Discrete emission costs (part 2)

Initial Conditions	EG	FL	EL	GL	EF
Uniform	0.720	0.596	0.379	0.386	0.388
Diagonal	0.888	0.871	0.590	0.465	0.661
Phonetic (zero-threshold)	0.900	0.888	0.706	0.543	0.707
Phonetic (vowel-threshold)	0.904	0.867	0.624	0.520	0.732

Table A.7: Viterbi log odds training (part 1)

Initial Conditions	FG	AL	AF	AG	AE	Avg
Uniform	0.344	0.341	0.229	0.264	0.209	0.386
Diagonal	0.507	0.483	0.440	0.179	0.213	0.530
Phonetic (zero-threshold)	0.564	0.500	0.433	0.267	0.377	0.589
Phonetic (vowel-threshold)	0.574	0.556	0.533	0.243	0.307	0.586

Table A.8: Viterbi log odds training (part 2)

Experiment	EG	FL	EL	GL	EF
Single Trans. Parameter	0.928	0.915	0.788	0.669	0.748
Testing with cQcT	0.893	0.875	0.747	0.625	0.656
Testing with cQ	0.922	0.910	0.803	0.705	0.747
Testing with cT	0.897	0.861	0.721	0.622	0.614
Training with cQcT	0.907	0.907	0.789	0.676	0.762
Training with cQ	0.925	0.906	0.789	0.711	0.735
Training with cT	0.912	0.889	0.747	0.662	0.682
Full Model	0.920	0.901	0.793	0.715	0.718
No τ	0.922	0.898	0.790	0.679	0.710
No λ	0.920	0.901	0.793	0.716	0.718
One λ	0.920	0.901	0.793	0.715	0.718

Table A.9: Viterbi results (part 1)

Experiment	FG	AL	AF	AG	AE	Average
Single Trans. Parameter	0.547	0.560	0.513	0.359	0.392	***0.642***
Testing with cQcT	0.472	0.478	0.348	0.347	0.248	0.569
Testing with cQ	0.552	0.538	0.439	0.356	0.328	0.630
Testing with cT	0.442	0.509	0.366	0.364	0.255	0.565
Training with cQcT	0.575	0.537	0.408	0.342	0.342	0.625
Training with cQ	0.513	0.539	0.445	0.346	0.316	0.622
Training with cT	0.482	0.548	0.399	0.396	0.305	0.602
Full Model	0.476	0.551	0.446	0.362	0.308	0.619
No τ	0.479	0.566	0.440	0.382	0.335	0.620
No λ	0.477	0.551	0.446	0.362	0.308	0.619
One λ	0.476	0.551	0.446	0.362	0.308	0.619

Table A.10: Viterbi results (part 2)

Experiment	EG	FL	EL	GL	EF
Single Trans. Parameter	0.929	0.914	0.793	0.670	0.755
Testing with cQcT	0.871	0.851	0.733	0.584	0.610
Testing with cQ	0.922	0.907	0.808	0.711	0.728
Testing with cT	0.867	0.830	0.661	0.534	0.546
Training with cQcT	0.907	0.906	0.787	0.669	0.756
Training with cQ	0.921	0.899	0.788	0.695	0.721
Training with cT	0.903	0.860	0.732	0.622	0.616
Full Model	0.919	0.895	0.795	0.704	0.705
No τ	0.919	0.889	0.790	0.664	0.689
No λ	0.919	0.895	0.795	0.704	0.705
One λ	0.919	0.895	0.795	0.704	0.705

Table A.11: Forward results (part 1)

Experiment	FG	AL	AF	AG	AE	Average
Single Trans. Parameter	0.555	0.563	0.530	0.382	0.390	***0.648***
Testing with cQcT	0.432	0.440	0.310	0.319	0.225	0.537
Testing with cQ	0.550	0.537	0.437	0.359	0.325	0.628
Testing with cT	0.395	0.459	0.313	0.308	0.245	0.516
Training with cQcT	0.577	0.536	0.413	0.344	0.357	0.625
Training with cQ	0.499	0.536	0.419	0.341	0.317	0.614
Training with cT	0.455	0.529	0.358	0.370	0.277	0.572
Full Model	0.477	0.561	0.419	0.367	0.309	0.615
No τ	0.487	0.562	0.428	0.383	0.316	0.613
No λ	0.477	0.561	0.419	0.367	0.309	0.615
One λ	0.477	0.561	0.419	0.367	0.309	0.615

Table A.12: Forward results (part 2)

Experiment	EG	FL	EL	GL	EF
Single Trans. Parameter	0.925	0.937	0.798	0.718	0.811
Testing with cQcT	0.927	0.936	0.800	0.724	0.812
Testing with cQ	0.911	0.941	0.807	0.762	0.807
Testing with cT	0.927	0.936	0.800	0.724	0.812
Training with cQcT	0.898	0.924	0.777	0.698	0.813
Training with cQ	0.914	0.942	0.803	0.742	0.804
Training with cT	0.933	0.938	0.794	0.679	0.795
Full Model	0.911	0.941	0.807	0.762	0.807
No τ	0.914	0.940	0.800	0.677	0.790
No λ	0.911	0.941	0.807	0.760	0.807
One λ	0.911	0.941	0.807	0.760	0.807

Table A.13: Log odds (constant indel) results (part 1)

Experiment	FG	AL	AF	AG	AE	Average
Single Trans. Parameter	0.754	0.683	0.667	0.355	0.358	0.701
Testing with cQcT	0.744	0.677	0.667	0.364	0.371	***0.702***
Testing with cQ	0.691	0.598	0.609	0.316	0.408	0.685
Testing with cT	0.744	0.677	0.667	0.364	0.371	***0.702***
Training with cQcT	0.689	0.566	0.556	0.348	0.448	0.672
Training with cQ	0.685	0.586	0.622	0.315	0.406	0.682
Training with cT	0.733	0.664	0.651	0.303	0.375	0.686
Full Model	0.691	0.598	0.609	0.316	0.408	0.685
No τ	0.679	0.579	0.588	0.305	0.393	0.666
No λ	0.691	0.598	0.611	0.316	0.408	0.685
One λ	0.690	0.598	0.609	0.316	0.408	0.685

Table A.14: Log odds (constant indel) results (part 2)

Experiment	EG	FL	EL	GL	EF
Single Trans. Parameter	0.925	0.925	0.792	0.744	0.801
Testing with cQcT	0.927	0.936	0.800	0.724	0.812
Testing with cQ	0.911	0.941	0.807	0.762	0.807
Testing with cT	0.924	0.924	0.791	0.741	0.797
Training with cQcT	0.898	0.924	0.777	0.698	0.813
Training with cQ	0.914	0.942	0.803	0.742	0.804
Training with cT	0.937	0.924	0.808	0.728	0.799
Full Model	0.905	0.923	0.779	0.728	0.777
No τ	0.909	0.924	0.763	0.662	0.767
No λ	0.905	0.923	0.779	0.728	0.777
One λ	0.905	0.923	0.779	0.728	0.777

Table A.15: Log odds (learned indel) results (part 1)

Experiment	FG	AL	AF	AG	AE	Average
Single Trans. Parameter	0.684	0.663	0.698	0.351	0.423	0.701
Testing with cQcT	0.744	0.677	0.667	0.364	0.371	***0.702***
Testing with cQ	0.691	0.598	0.609	0.316	0.408	0.685
Testing with cT	0.683	0.649	0.694	0.355	0.424	0.698
Training with cQcT	0.689	0.566	0.556	0.348	0.448	0.672
Training with cQ	0.685	0.586	0.622	0.315	0.406	0.682
Training with cT	0.669	0.671	0.669	0.367	0.417	0.699
Full Model	0.641	0.559	0.588	0.324	0.396	0.662
No τ	0.633	0.547	0.578	0.281	0.351	0.641
No λ	0.641	0.559	0.591	0.324	0.396	0.662
One λ	0.641	0.558	0.588	0.324	0.396	0.662

Table A.16: Log odds (learned indel) results (part 2)

Experiment	EG	FL	EL	GL	EF
Single Trans. Parameter	0.921	0.891	0.719	0.562	0.708
Testing with cQcT	0.887	0.842	0.628	0.507	0.608
Testing with cQ	0.924	0.910	0.760	0.648	0.725
Testing with cT	0.899	0.842	0.653	0.534	0.622
Training with cQcT	0.909	0.906	0.756	0.643	0.783
Training with cQ	0.925	0.905	0.748	0.638	0.720
Training with cT	0.922	0.877	0.702	0.596	0.689
Full Model	0.926	0.905	0.763	0.652	0.726
No τ	0.916	0.893	0.738	0.582	0.698
No λ	0.926	0.905	0.762	0.652	0.726
One λ	0.926	0.905	0.763	0.652	0.726

Table A.17: Forward log odds results (part 1)

Experiment	FG	AL	AF	AG	AE	Average
Single Trans. Parameter	0.556	0.502	0.514	0.232	0.416	0.602
Testing with cQcT	0.456	0.482	0.427	0.244	0.276	0.536
Testing with cQ	0.594	0.544	0.537	0.276	0.427	0.634
Testing with cT	0.472	0.524	0.427	0.265	0.294	0.553
Training with cQcT	0.639	0.537	0.527	0.309	0.403	***0.641***
Training with cQ	0.575	0.537	0.540	0.269	0.415	0.627
Training with cT	0.552	0.572	0.491	0.297	0.341	0.604
Full Model	0.595	0.567	0.542	0.283	0.427	0.639
No τ	0.554	0.523	0.507	0.254	0.364	0.603
No λ	0.595	0.567	0.542	0.283	0.427	0.638
One λ	0.595	0.567	0.542	0.283	0.427	0.639

Table A.18: Forward log odds results (part 2)

Appendix B

Glossary

AE

The language pair Albanian and English.

AF

The language pair Albanian and French.

AG

The language pair Albanian and German.

AL

The language pair Albanian and Latin.

ARPAbet

A phonetic alphabet for American English that uses only ASCII characters.

borrowing

One possible source of cognates, where a word is taken from one language and added to another. These words are often cultural specific terms.

cognates

Words between languages that have a similar form or sound, and a similar meaning. See also *genetic cognates.*

deletion

One of the three basic edit operations. It involves removing one of the

tokens of the first word. This is equivalent to the alignment of that token to a *gap*.

dynamic programming

A class of algorithms that use table-based calculations to solve problems by combining solutions to sub-problems.

EF

The language pair English and French.

EG

The language pair English and German.

EL

The language pair English and Latin.

FG

The language pair French and German.

FL

The language pair French and Latin.

F.L.O.

An abbreviation for the forward-based log odds algorithm.

For

An abbreviation for the forward algorithm.

gap

A series of either continuous *insertions* or *deletions*. Each gap contains only one of *insertions* or *deletions*, not both. It is shown in alignments with "-" and represents alignment to nothing.

genetic cognates

A more specific type of cognate, especially useful in historical linguistics. The words in the pair must have both evolved from the same root word, called a *proto-form*.

GL

The language pair German and Latin.

Hidden Markov Model

A Markov Model that uses a second distribution to produce its output. This extra layer of randomness makes the state sequence hidden, since it cannot be determined directly from the observed output of the model.

HMM

See *Hidden Markov Model.*

IK

The language pair Italian and Serbo-Croatian.

indel

This term represents both insertions and deletions. It is normally used when both operations are being considered in the same way, due to the symmetry that often exists between them.

insertion

One of the three basic edit operations. It involves adding a token to the second word. This is equivalent to the alignment of that token to a *gap.*

International Phonetic Alphabet

A phonetic alphabet produced by the International Phonetic Association (IPA). The goal of the IPA is to be able to represent *all* spoken languages.

L.O.C.

An abbreviation for the Viterbi-based log odds algorithm that assumes the insertion and deletion probabilities of the word similarity model are equal to the emission probabilities of the random model.

L.O.L.

An abbreviation for the Viterbi-based log odds algorithm that uses the learned insertion and deletion probabilities for the word similarity model and letter frequencies for the emission probabilities of the random model.

NED

See *Normalized Edit Distance.*

Normalized Edit Distance

A simple process that determines how many edit operations are required to transform one word into another. The final solution is normalized by the length of the longest of the two words.

Markov Model

A model for a stochastic process that only retains the minimum amount of prior knowledge. Only the current event is needed in order to predict (or generate) the next event.

match

See *substitution.*

orthographic

One possible way to represent words. The word is shown as it would be written in its natural language (or some approximation of it).

Pair Hidden Markov Model

A variation of a *Hidden Markov Model* that produces two output streams in parallel. Each output stream is accessed independently.

PHMM

See *Pair Hidden Markov Model.*

phonetic

One possible way to represent words. The word is shown using symbols to represent the sounds produced when the word is spoken.

PR

The language pair Polish and Russian.

proto-form

A word in an older (possibly pre-historic) language, that is the source of a word (or words) in one or more modern languages.

random model

A formal description of the coincidental patterns that exist within languages. The model can generate pairs of words that are likely to exist within languages but are not related to each other.

substitution

One of the three basic edit operations. It involves transforming a token in the first word into another token in the second word. For alignment it is sometimes referred to as a *match*.

tokens

The parts of a word after it is broken up according to its representation. The tokens are the components that are used in alignment.

transliteration

The transformation of a word from one language into another language based on the spelling of that word. It is similar to *borrowing*, but more often used for proper nouns.

Vit

An abbreviation for the Viterbi algorithm.

word similarity

An abstract concept that represents the strength of the relationship between words. The relationship can be anything, including surface or sound similarity. It is also possible for words to exhibit such a similarity by chance, making the problem of recognizing related words more difficult.

word similarity model

A formal description for a specific choice of *word similarity*. The model can generate similar pairs of words, or calculate how similar a given pair of words is.